## Contents

Abstract ................................. 10

Declaration ............................ 11

Copyright ................................ 12

Acknowledgements ....................... 13

1 Introduction ............................ 14
   1.1 Rise of Multi-cores .................. 14
   1.2 Parallel Programming ............... 16
       1.2.1 Parallel Programming Models .... 16
           1.2.1.1 Thread-Level Parallelism .... 16
           1.2.1.2 Data Parallelism ........... 16
           1.2.1.3 Message Passing ............ 17
           1.2.1.4 Hybrid ..................... 17
       1.2.2 Software Challenges ............. 18
           1.2.2.1 Locks ....................... 18
           1.2.2.2 Transactional Memory ........ 19
   1.3 Research Objectives ................. 20
   1.4 Contributions ........................ 21
   1.5 Thesis Organization .................. 22
   1.6 Publications ......................... 24

2 Background ............................ 25
   2.1 High Performance Java ............... 25
       2.1.1 Distributed JVMs ............... 26
       2.1.2 Library Assisted Distributed Java 27
4.2.2 Remote Communication ........................................... 49
4.3 Memory Management .............................................. 50
  4.3.1 Centralized approach ........................................... 51
  4.3.2 Decentralized approach ....................................... 52
  4.3.3 Distributed Atomic Collection Classes ....................... 53
    4.3.3.1 Distributed Atomic Singleton Objects .................. 53
    4.3.3.2 Distributed Atomic Arrays ............................... 54
    4.3.3.3 Distributed Atomic LinkedList ......................... 54
    4.3.3.4 Distributed Atomic HashMap ......................... 56
4.4 Transactional Coherence Protocols ............................... 57
  4.4.1 Centralized Protocols ....................................... 58
    4.4.1.1 TCC .................................................. 58
    4.4.1.2 Serialization Lease .................................. 62
    4.4.1.3 Multiple Leases ..................................... 64
  4.4.2 Decentralized Protocol ..................................... 66
    4.4.2.1 Anaconda Data Structures ............................. 66
    4.4.2.2 Anaconda Commit Procedure ............................ 67
    4.4.2.3 Commit Process ...................................... 69
    4.4.2.4 Simple Commit Example ............................... 70
    4.4.2.5 Contention Cases .................................... 72
4.5 Summary .......................................................... 74
5 Evaluation .......................................................... 75
  5.1 Introduction .................................................... 75
  5.2 Experimental Platform .......................................... 75
    5.2.1 Hardware ............................................... 75
    5.2.2 Benchmarks ............................................. 76
    5.2.3 Lee-TM ............................................... 78
    5.2.4 KMeans .............................................. 80
    5.2.5 Genome ............................................... 80
    5.2.6 Vacation ............................................. 82
    5.2.7 Game of Life ........................................... 83
  5.3 Transactional Profiles ......................................... 83
  5.4 Speedups and Execution Times .................................. 85
    5.4.1 Speedups ............................................. 85
      5.4.1.1 TCC ............................................... 85

# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>High Performance Java systems classification.</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>Distributed Software Transactional Memory systems classification.</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Active Objects per Protocol</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>DiSTM’s transactional coherence protocols characteristics.</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Benchmarks’ parameters</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Average readset size of committed transactions (number of transactional objects).</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>Average writset size of committed transactions (number of transactional objects).</td>
<td>86</td>
</tr>
<tr>
<td>5.4</td>
<td>Benchmarks’ execution times (seconds) - TCC protocol.</td>
<td>88</td>
</tr>
<tr>
<td>5.5</td>
<td>Benchmarks’ execution times (seconds) - SL protocol</td>
<td>93</td>
</tr>
<tr>
<td>5.6</td>
<td>Benchmarks’ execution times - ML protocol</td>
<td>99</td>
</tr>
<tr>
<td>5.7</td>
<td>Benchmarks’ execution times - Anaconda protocol</td>
<td>105</td>
</tr>
<tr>
<td>B.1</td>
<td>LeeTM Commits/Aborts - TCC protocol</td>
<td>141</td>
</tr>
<tr>
<td>B.2</td>
<td>LeeTM Commits/Aborts - SL protocol</td>
<td>142</td>
</tr>
<tr>
<td>B.3</td>
<td>LeeTM Commits/Aborts - ML protocol</td>
<td>142</td>
</tr>
<tr>
<td>B.4</td>
<td>LeeTM Commits/Aborts - Anaconda protocol</td>
<td>142</td>
</tr>
<tr>
<td>B.5</td>
<td>LeeTM-ER Commits/Aborts - TCC protocol</td>
<td>143</td>
</tr>
<tr>
<td>B.6</td>
<td>LeeTM-ER Commits/Aborts - SL protocol</td>
<td>143</td>
</tr>
<tr>
<td>B.7</td>
<td>LeeTM-ER Commits/Aborts - ML protocol</td>
<td>143</td>
</tr>
<tr>
<td>B.8</td>
<td>LeeTM-ER Commits/Aborts - Anaconda protocol</td>
<td>144</td>
</tr>
<tr>
<td>B.9</td>
<td>GLife Commits/Aborts - TCC protocol</td>
<td>144</td>
</tr>
<tr>
<td>B.10</td>
<td>GLife Commits/Aborts - SL protocol</td>
<td>144</td>
</tr>
<tr>
<td>B.11</td>
<td>GLife Commits/Aborts - ML protocol</td>
<td>145</td>
</tr>
<tr>
<td>B.12</td>
<td>GLife Commits/Aborts - Anaconda protocol</td>
<td>145</td>
</tr>
<tr>
<td>B.13</td>
<td>Vacation Commits/Aborts - TCC protocol</td>
<td>145</td>
</tr>
</tbody>
</table>
B.14 Vacation Commits/Aborts - SL protocol . . . . . . . . . . . 146
B.15 Vacation Commits/Aborts - ML protocol . . . . . . . . . . . 146
B.16 Vacation Commits/Aborts - Anaconda protocol . . . . . . . 146
B.17 Genome Commits/Aborts - TCC protocol . . . . . . . . . . . 146
B.18 Genome Commits/Aborts - SL protocol . . . . . . . . . . . 147
B.19 Genome Commits/Aborts - ML protocol . . . . . . . . . . . 147
B.20 Genome Commits/Aborts - Anaconda protocol . . . . . . . 147
B.21 KMeansLow Commits/Aborts - TCC protocol . . . . . . . . . 148
B.22 KMeansLow Commits/Aborts - SL protocol . . . . . . . . . 148
B.23 KMeansLow Commits/Aborts - ML protocol . . . . . . . . . 148
B.24 KMeansLow Commits/Aborts - Anaconda protocol . . . . . 149
B.25 KMeansHigh Commits/Aborts - TCC protocol . . . . . . . . 149
B.26 KMeansHigh Commits/Aborts - SL protocol . . . . . . . . . 149
B.27 KMeansHigh Commits/Aborts - ML protocol . . . . . . . . . 150
B.28 KMeansHigh Commits/Aborts - Anaconda protocol . . . . 150
List of Figures

4.1 DiSTM generic architecture ........................................... 48
4.2 Active Object .......................................................... 49
4.3 DiSTM in centralized mode ............................................ 51
4.4 DiSTM in decentralized mode ......................................... 52
4.5 Partitioning types of distributed atomic arrays ................... 54
4.6 State diagrams of distributed linked list operations (decentralized mode) .......................... 56
4.7 Distributed atomic hashmap structure ................................ 57
4.8 Stages of the TCC protocol ........................................... 59
4.9 TCC protocol state diagram .......................................... 59
4.10 Stages of the Single Lease protocol .................................. 63
4.11 Single Lease protocol state diagram .................................. 63
4.12 Stages of the Multiple Leases protocol ............................... 64
4.13 Multiple Leases protocol state diagram ............................... 65
4.14 Transactional Object Cache (TOC) structure ......................... 67
4.15 Transactional Object Buffer (TOB) structure ......................... 67
4.16 Stages of the Anaconda protocol ..................................... 68
4.17 Anaconda state diagram .............................................. 68
4.18 Anaconda protocol simple commit example ............................ 71

5.1 Expansion phase of Lee-TM. ........................................... 77
5.2 Backtracking phase of Lee-TM. ....................................... 78
5.3 Sample routing of the mainboard.txt circuit, a realistic microcode processor .............................. 79
5.4 The aim of gene sequencing is to construct a unique gene sequence from a pool of overlapping gene segments. .......................... 81
5.5 Ratio of ReadSet to WriteSet (RSstoWS) ............................... 85
5.6 Benchmarks’ speedups - TCC protocol .................................. 90
5.7 Breakdown by transaction stage of the execution times. - TCC protocol ................................. 91
5.8 Benchmarks’ speedups - Single Lease protocol .............. 96
5.9 Breakdown by transaction stage of the execution times. - SL protocol 97
5.10 Benchmarks’ speedups - Multiple Leases protocol .............. 102
5.11 Breakdown by transaction stage of the execution times. - ML protocol 103
5.12 Benchmarks’ speedups - Anaconda protocol .............. 108
5.13 Breakdown by transaction stage of the execution times. - ANA protocol 109
5.14 Lee execution times. .................................................. 111
5.15 Lee-ER execution times. .............................................. 112
5.16 KMeansHigh execution times. ..................................... 112
5.17 KMeansLow execution times. ...................................... 113
5.18 GLife execution times. ................................................ 113
5.19 Vacation execution times. .......................................... 114
5.20 Genome execution times. .......................................... 114
5.21 Percentage of time spent in GC. .................................. 117
5.22 Best performing transactional protocols. ..................... 118
5.23 Best performing protocols (including lock-based). ............. 119

B.1 Aborts per Commit of LeeTM ....................................... 151
B.2 Aborts per Commit of LeeTM-ER .................................. 151
B.3 Aborts per Commit of KMeansHigh ............................... 152
B.4 Aborts per Commit of KMeansLow ............................... 152
B.5 Aborts per Commit of Vacation ................................... 153
B.6 Aborts per Commit of Genome ................................... 153
B.7 Aborts per Commit of GLife ....................................... 154
Abstract

Over the past years research and development on computer architecture has shifted from uni-processor systems to multi-core architectures. This transition has created new incentives in software development because in order for the software to scale it has to be highly parallel. Traditional synchronization primitives based on mutual exclusion locking are challenging to use and therefore are only efficiently employed by a minority of expert programmers.

Transactional Memory (TM) is a new alternative parallel programming model aiming to alleviate the problems that arise from the use of explicit synchronization mechanisms. In TM, lock guarded code is replaced by memory transactions which comply with the ACI (atomicity, consistency, isolation) principles. The simplicity of the programming model that TM proposes has led to major research efforts by academia and industry to produce high-performance TM implementations. The majority of these TM systems, however, focus on shared-memory Chip MultiProcessors (CMPs) leaving the area of distributed systems unexplored.

This thesis explores Transactional Memory in the distributed systems domain and more specifically on small-scale clusters. A variety of novel distributed transactional coherence protocols are proposed and evaluated, against complex TM oriented benchmarks, in the context of distributed Java Virtual Machines (JVMs) - an area that has received much attention over the last decade due its perfect applicability into the enterprise domain.

The implemented Distributed Software Transactional Memory (DiSTM) system, proposed in this thesis, is a JVM clustering solution that employs software transactional memory as its synchronization mechanism. Due to its modular design and ease in programming, it allows the addition of new protocols in a fairly easy manner. Finally, DiSTM is highly portable as it runs on top of off-the-shelf JVMs and requires no changes to existing Java source code.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of School of Computer Science (or the Vice-President).
First of all, I would like to thank my supervisor Dr. Chris Kirkham. His guidance and continuous encouragement have been invaluable to the conduction of this research project.

Furthermore, I would like to thank Dr. Mikel Luján for guiding me into the research path. I will never forget the hours we spent late at night trying to meet conference deadlines.

Moreover, I would like to thank all the members of the Jamaica group (Behram, Kim, Simon, Ian, Mohammed, Mohammad) for their help and company throughout my studies at the University of Manchester. Especially, I would like to thank Mohammad Ansari for being my research partner and for his continuous assistance. The time we spend together from debugging to discussing out of topic subjects for hours will always remain in my mind.

I thank all my friends, those I met in Manchester and my loyal friends back home. Their comfort and support during my life have been my driving force. A “thank you” will never be enough.

Finally, I would like to dedicate this thesis to my family and Fiona. To my wonderful parents (Konstantinos and Anastasia), my brother (Panagiotis) and my grandparents (Christos, Xarikleia and Vasileia), whom I could not have done this without. I am waiting for the day I will be able to give back to them even a small percentage of what they so generously offered to me all these years. I am blessed for having such a wonderful family.

Special thanks goes to Fiona Schwab for her support during these, often, difficult years. Her kindness, patience and understanding keep astonishing me until now. I am grateful to Fiona for being the reason for pursuing this PhD. Thank you Fi!
Chapter 1

Introduction

The shift from uni-processor to multi-core chips is currently underway. Nowadays, the majority of vendors (Intel, AMD, IBM, Sun etc.) invest in multi-core product lines. Similarly, computer clusters will undergo this transition. In future, computer nodes of clusters will incorporate tens or hundreds of cores. The programmability of such systems will become even more challenging, as traditional synchronization mechanisms will not be able to cope with future highly parallel workloads.

The remainder of this chapter is organized as follows: Section 1.1 analyzes the reasons behind the transition to multi-core architectures. Section 1.2 explains the software challenge deriving from the rise of multi-core. Section 1.2.1 discusses the current parallel programming models, while Section 1.2.2.2 introduces the Transactional Memory (TM) parallel programming model. Section 1.3 presents the research objectives of the proposed research, while Section 1.4 states the contributions of this thesis. Finally, Section 1.5 outlines the organization of the presentation.

1.1 Rise of Multi-cores

Microprocessors have been increasing their speed and performance exponentially during the last decades. Following the Von Neumann model and complying with Moore’s law [78], CPUs’ speed has been improving drastically on a yearly basis. This increase in performance, however, seems to stall lately [81, 80, 74, 42] due to significant power consumption and overheating.

Traditionally, in order to increase the performance of modern CPUs, more
transistors are being incorporated in them. The rationale behind that is the maximum exploitation of Instruction Level Parallelism (ILP). With ILP, a set of many instructions is divided to a number of smaller segments (transparently to the user). Hence, more instructions can be pipelined increasing the throughput of the system (more instructions are being executed per clock cycle). The increase of the number of transistors placed on a chip, however, requires the usage of more power to operate and is limited due to overheating. The amount of power required and the expensive cooling systems needed in order to operate CPUs with billions of transistors prohibit their use. The afore-mentioned problems have led to a search for an alternative solution in order to increase computers throughput - the development of Chip MultiProcessors (CMPs).

Chip MultiProcessors (CMPs) (multi-core or many-core) achieve high performance by incorporating numerous processors on a single chip. Although the clock speed of the processors (cores) used is not higher compared to those of single-processor systems, the fact that they can execute tasks in parallel results in their increased performance. A safe conclusion that can be drawn is that multi-core computing is becoming mainstream creating new research challenges. Until now, the majority of the programs written were specifically designed for single-processor architectures (i.e. to run serially). With the introduction of multi-core architectures, existing or new parallel programming models have to be used in order to exploit the available parallelism to its maximum.

Computer clusters, which are groups of network interconnected commodity computers working together as a single system, have been widely used over the past decades in high-performance computing. Since computer systems are becoming multi-core, inevitably, computer clusters will consist of network interconnected CMPs. In such clustered environments, the available parallelism is two-dimensional. The first dimension concerns the exploitable parallelism within a single node of the cluster while the second concerns the parallelism available from executing applications in a distributed manner. The programmability of such systems is becoming a vital issue. Traditional parallel programming models have to be re-examined or redesigned in order to provide a parallel programming model capable of dealing with such an abundance of available parallelism in the most efficient manner. The next section (Section 1.2) describes the current prevailing parallel programming models, highlighting their advantages and disadvantages along with software challenges derived from the multi-core era.
1.2 Parallel Programming

Parallel programming is known to be a challenging and difficult task. Over decades of research a variety of parallel programming models have emerged with each one of them having distinct characteristics. Section 1.2.1 categorizes and analyzes the existing parallel programming models while Section 1.2.2 pinpoints the weaknesses of the synchronization mechanisms used in thread-level parallelism, which is the programming model tackled in this thesis. Furthermore, it introduces Transactional Memory, a new parallel programming model which aims to alleviate many of the problems that current shared-memory programming models have.

1.2.1 Parallel Programming Models

Existing parallel programming models fall broadly into four categories: Thread-Level Parallelism, Data Parallelism, Message Passing and Hybrid Parallelism.

1.2.1.1 Thread-Level Parallelism

Thread-based implementations, such as Java [46], are widely used for parallel programming. Each thread performs a number of assigned tasks, reading or writing from/to the shared main memory. Memory consistency is achieved by employing synchronization primitives such as locks and barriers. When concurrent threads attempt to modify the same memory data (hereafter referred to as objects), mutual exclusion locks are employed in order to serialize the execution of the competing threads. Synchronization is achieved by explicitly guarding code segments, that access shared variables and hence they should be protected, with locks. The drawback of using such an approach is the programming complexity added in order to achieve correct implementations or resignation to low degrees of concurrency between critical sections.

1.2.1.2 Data Parallelism

Data parallelism and other similar traditional High Performance Computing (HPC) approaches are specifically designed for splitting and assigning large datasets to
a number of computing resources. Usually, a master node assigns \(^2\) portions of a large data structure (e.g. an array) to a worker node. Each node performs a number of calculations to its personal portion of the dataset and returns the result to the master node (Single Process Multiple Data (SPMD) model). When all nodes complete their tasks, the master node gathers and exports the results. Such implementations, such as High Performance Fortran (HPF) [59] or HPJava [27], although they perform well on static data structures, face difficulties in the presence of complex operations on dynamic data structures. Furthermore, the programmers need to master the specific syntax of those languages.

1.2.1.3 Message Passing

Message Passing implementations, such as MPI [43], mpiJava [70] and MPJ Express [69], have been widely adopted in high performance scientific computing. Programs written in this fashion adhere to a message-exchange methodology in order to communicate data between nodes. Programmers, through explicit code, communicate messages across the cluster via send and receive functions. In the Java world this kind of programming model can be implemented in various ways varying from custom socket interfaces (which support TCP/IP communication), to high level mechanisms such as Remote Method Invocation (RMI) [76] and Common Object Request Broker Architecture (CORBA) [75]. Message passing style algorithms suit ideally this kind of programming model. However, the message passing programming style makes it difficult for programmers to use in a generic manner.

1.2.1.4 Hybrid

Hybrid models combine the Message Passing model with Thread-Level parallelism. As stated in Section 1.1, the advent of multi-cores will result in having clusters of Chip MultiProcessors (CMPs). Hybrid models employ thread-level parallelism in the context of a single node of the cluster and message passing style in order to parallelize and coordinate processes of the same application running on remote nodes.

The research proposed in this thesis targets hybrid parallel programming models in the context of Java Virtual Machines (JVMs). In this way we try to exploit

\(^2\)The term node in this section is used in an abstract way denoting either a single processor or a computer node.
thread-level parallelism with distributed processing. The following section discusses the problems of shared-memory programming, giving an introduction to a novel alternative parallel programming model called Transactional Memory.

1.2.2 Software Challenges

Synchronization of shared variables is achieved, traditionally, by the use of mutual exclusion lock-based primitives. The pathologies of lock-based programming (Section 1.2.2.1), along with the rise of multi-cores, drove the community to invest in exploring new parallel programming models. The outcome from these efforts is Transactional Memory (TM), a new programming model aiming to alleviate the complexities of lock-based programming (Section 1.2.2.2).

1.2.2.1 Locks

The traditional synchronization primitives in multi-threaded shared-memory programming are lock-based primitives. Critical sections of code (i.e. pieces of code that access a shared resource which must not be concurrently accessed by more than one thread) are guarded by software variables called locks. The responsibility of locks is to prohibit the guarded piece of code from being accessed concurrently by multiple threads.

Each thread trying to access a guarded piece of code has, firstly, to acquire the lock associated with this code. Only after a thread has acquired the lock, may it continue in executing the critical section. The successful acquisition of the lock provides the implicit guarantee that no other thread can execute the critical section while the current thread owns the lock; the current thread has been granted exclusive ownership of the critical section. Upon finishing executing the critical section, the thread releases the lock which can then be acquired by other waiting threads.

Inherently, lock-based synchronization has some limitations [58]. Coarse-grain locking, i.e. guarding large segments of data with a single lock, limits scalability. While threads contend to acquire the lock of a critical section, some of them will block upon failed acquisition waiting for the lock to be released. The waiting time is proportional to the size of the critical section. Therefore, coarse-grain locking will eventually lead to poor scalability. Trying to minimize the critical
sections while employing more locks results in fine-grain locking. Fine-grain locking, although more scalable than coarse-grain locking, can either result in higher overhead due to multiple locks acquisition or lead to deadlocks.

Deadlock is a common pathology of lock-based programming. It results from the actions of different threads trying to acquire locks of the same set of objects in different orders. Imagine Thread A (Ta) which already acquired the lock of object X (Ox) attempting to acquire the lock of object y (Oy). At the same time Thread B (Tb) which already acquired the lock of object y attempts to acquire Ox. The two threads will eventually deadlock waiting for each other to release their locks in order to progress. Deadlock avoidance is a challenging task especially in environments where either the actions of the threads or the set of locks are not known in advance. In addition to deadlocks, the usage of locks can also result in other problems. Priority inversion, in which a lower-priority thread is pre-empted while holding a lock needed by a higher-priority thread, is an example.

Many of the weaknesses mentioned above can be solved; however, it requires good software development practices rather than transparent solutions provided from the runtime. As a result, lock-based concurrent programming remains challenging.

1.2.2.2 Transactional Memory

Transactional Memory (TM) [65] is a new promising approach for parallel programming. Originating from database theory, TM uses transactional semantics in order to perform memory operations. The traditional mutual exclusion primitives used in the lock-based thread-level parallelism model are replaced by atomic{...} like constructs. The code included in the atomic statement is executed transactionally. If the transaction commits, it makes its changes visible to the rest of the system. Otherwise (in case of conflict) it aborts and restarts execution fetching the latest values from the memory. Section 2.2 explains Transactional Memory in detail in the context of this research.

The atomicity of the memory transactions is guaranteed by the underlying TM runtime system. The TM runtime system records the objects accessed by transactions in their correspondent read/write sets which will be used later in conflict detection and conflict resolution. Typically transactions execute optimistically and, usually, upon commit they validate themselves against concurrent executing
transactions. If a conflict is detected, depending on the conflict resolution policy, only one transaction is committed while the other conflicting ones abort. Upon successful commit, a memory transaction makes its speculative changes globally visible. On the contrary, in case of abort, the transaction either “rolls back” its changes or just discards its speculative changes.

In summary, transactions are inherently optimistic: all transactions are permitted to access any critical section. In this way, all memory transactions can run in parallel and at the end of their execution conflicts are detected. In addition, transactional execution is non-blocking: a thread running a transaction is guaranteed forward-progress\(^3\) even if all competing transactions suspend their execution or fail. Consequently, transactional memory can overcome two major problems of lock-based programming: scalability and deadlocks.

### 1.3 Research Objectives

As stated in Section 1.2.1, the advent of multi-core architectures will eventually lead to clusters of multi-core chips. The hybrid (message passing/thread level parallelism) parallel model inherits both the pathologies of lock-based programming (Section 1.2.2) and the exposed programming abstractions of message passing libraries.

The present work attempts to replace the traditional lock-based synchronization primitives in a clustered environment. The majority of the research efforts on Transactional Memory focuses on shared-memory architectures leaving the area of distributed systems relatively unexplored. Employing TM on clustered environments is a promising approach for reducing complexity and improving scalability of cluster-aware applications.

Cluster-aware applications broadly fall into two categories: scientific applications and enterprise applications. Scientific applications suit the message-passing model as the majority of them can be modeled according to the SPMD model\(^4\). Thus, this thesis focuses on generic enterprise applications that can run in a distributed manner. The majority of enterprise applications are written in Java and run on top of Java Virtual Machines (JVMs). The research presented here ex-

---

\(^3\)Depending on the implementation.

\(^4\)According to the Single Process Multiple Data (SPMD) model, data are partitioned amongst nodes where the same processes perform calculations on them with different inputs.
plquets Transactional Memory on distributed systems. More specifically, it exploits software transactional memory execution on clustered JVMs. The contributions of this thesis are stated in the following section.

1.4 Contributions

The presented work makes the following contributions:

- It introduces the first Java Virtual Machine clustering solution with software transactional memory support. The Distributed Software Transactional Memory (DiSTM) system provides a Single System Image (SSI) view to the user, where Java applications can run transparently on clusters. The traditional lock-based synchronization primitives of Java (synchronized{...} statements) have been replaced by memory transactions (@distatomic{...} annotations). Threads executing memory transactions validate against locally and remotely running transactions ensuring memory consistency. Java programs require no modifications in order to run on DiSTM. Furthermore, DiSTM can run on top of off-the-shelf JVMs which makes it highly portable.\(^5\)

- DiSTM, due to its flexible and modular design, allows easy experimentation. Different distributed transactional coherence protocols can be implemented and studied in the DiSTM context. Four novel distributed transactional coherence protocols have been designed and implemented in DiSTM. The proposed distributed transactional coherence protocols fall into two categories: centralized (with three instances) and decentralized (with one).

- DiSTM supports two modes of execution: master-centric and fully decentralized. In the master-centric approach, the execution is coordinated by a master node which is responsible for maintaining the consistency of the data in the cluster. The transactional data are maintained by the master node while the remaining worker nodes hold cached copies of it. In contrast, the decentralized approach does not necessitate the existence of the master node in order to run. The transactional data are partitioned amongst the

\(^5\)DiSTM does not utilize any form of distributed garbage collection. However, it ensures that no vital for the execution data are collected by declaring them as static.
worker nodes while the decentralized protocol ensures its consistency. Object caching and replication along with the communication mechanisms of DiSTM are abstracted away from the user.

• DiSTM is evaluated against complex TM oriented benchmarks providing an insight into transactional memory application on distributed environments. A variety of recently established TM oriented benchmarks have been ported and used for the evaluation of DiSTM. The various benchmarks exhibit diverse characteristics stretching DiSTM over various workloads and contention levels.

• Results indicate that depending on the nature of the benchmarks different protocols perform better. Although some benchmarks benefit from distributed transactional execution, others suffer performance degradation due to the overheads of remote execution and Garbage Collection.

1.5 Thesis Organization

Chapter 2 presents the background work and is divided into two sections. The first section concerns the work carried out in the distributed JVM domain, introducing and categorizing that relevant to this research. The second section introduces TM. The design and implementation space of TM research is discussed and analyzed. Finally, after having introduced the vital terminology and techniques of both distributed JVMs and TM domains, the closely related systems to DiSTM are analyzed and compared.

Chapter 3 presents the programming model employed by DiSTM. Furthermore, it entails source code examples of applications ported to DiSTM.

Chapter 4 presents the architecture of DiSTM. Initially the generic core parts of DiSTM are explained; the transactional engine and the remote communication system. The chapter continues by explaining the memory management of DiSTM and the distributed atomic collection classes implemented. Finally, the four novel distributed transactional coherence protocols are described.

Chapter 5 begins with the description of the benchmarks used for the evaluation along with the hardware platform used and continues with the evaluation of DiSTM. The transactional protocols are comparatively evaluated to each other.
and to lock-based implementations ported to a commercial state of the art clustering JVM solution.

Chapter 6 summarizes the contributions of this thesis, and indicates directions for future research.
1.6 Publications

Papers

- Christos Kotselidis, Mikel Luján, Mohammad Ansari, Konstantinos Malakasis, Behram Kahn, Chris Kirkham and Ian Watson

  Clustering JVMs with Software Transactional Memory Support. In the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS ’10), Atlanta, USA, 2010.

- Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham and Ian Watson


- Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham and Ian Watson

  Investigating Software Transactional Memory on Clusters. In the 10th IEEE International Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJPDC ’08), Miami, USA, 2008.

Abstracts/Posters

- Christos Kotselidis


- Christos Kotselidis

  Exploiting Distributed Transactional Memory. In the IPDPS ’08 TCPP PhD Forum, Miami, USA, 2008.
Chapter 2

Background

This chapter discusses the background work related to the research. The background work is divided into two parts. Section 2.1 introduces the research conducted in the High Performance Java (HPJava) domain. The relevant systems are introduced, categorized and compared against DiSTM. Section 2.2 describes the internals of Transactional Memory (TM) systems. Both the design and the implementation spaces are analyzed in order to provide a comprehensive introduction to the internals of the TM engine of DiSTM. Finally, Section 2.3 presents the work directly related to this thesis. The recently emerged, relevant distributed systems, which employ TM are discussed and compared against DiSTM.

2.1 High Performance Java

The Java programming language with its inherent multi-threaded support and automatic memory management has been widely adopted in various areas of parallel computing. Due to the shared-memory model that Java provides it is much easier to use compared to other solutions such as Parallel Virtual Machine (PVM) [1], MPI [43] and Distributed Shared Memory (DSM) [61, 54]. During the last decade there has been a considerable increase in the use of the Java programming language in the enterprise domain. The first reason is the ease in programming, while the second is that it ideally fits the logic of enterprise applications. Multiple requests from clients are assigned on distinct threads running on the server, while synchronization is achieved through explicit locking. Consequently, the majority of application and web servers, such as JBoss [41], WebSphere [60] and Tomcat [44], have been implemented in Java. Clustering JVMs in a transparent
way adds another level of parallelism to Java applications. Especially nowadays, where multi-core architectures are emerging, future clusters will consist of boards of CMPs with hundreds of nodes. Working towards this a number of distributed or clustered JVMs emerged during the last decade. The following subsections categorize the various implementations related to the work described in this thesis.

2.1.1 Distributed JVMs

A number of Distributed JVMs has been developed in order to scale Java applications on clusters. cJVM [14], developed by IBM, is a distributed JVM that provides a Single System Image (SSI) to the user and targets multithreaded Java server applications. cJVM requires no modifications of Java programs in order for them to run. Synchronization is achieved through mutual exclusion locks.

Jessica [71] and its successor Jessica2 [101] are also distributed JVM solutions similar to cJVM. Where they differ is in the way they tackle remote objects. cJVM follows an optimized “master-proxy” approach. When an object is created in a node, the object on its creator node is called the master copy of the object. Other nodes access the object via a proxy stored in their heap. On the contrary, Jessica runs on top of a DSM. Jessica2 implements a Global Object Space (GOS) scheme that supports objects’ migration. In addition, Jessica2 uses a just-in-time (JIT) compiler whereas cJVM is interpreter based. Finally, Jessica2 implements a novel thread migration technique.

CoJVM [5] is similar to Jessica, as it runs on top of a DSM. CoJVM distinguishes itself from Jessica in that it performs synchronization on shared objects. CoJVM employs distributed locks whereas in Jessica all synchronization is performed on the master node thereby significantly slowing down execution time.

Java/DSM [99] is a distributed JVM which runs on top of TreadMarks [61] page-based DSM. It requires from the user to specify the location of the node that a Java thread should run, in contrast to the aforementioned systems. Finally, it uses an interpreter instead of a JIT compiler slowing down execution significantly.

Jackal [83] provides a distributed shared memory for Java programs. It does not require any modifications of the programs in order for them to run on a distributed environment. However, Jackal employs a static compiler in order to statically compile the Java programs to assembly code suitable to run through its runtime environment. Furthermore, Jackal’s runtime system employs a custom
implementation of a distributed shared memory. Synchronization is achieved through lock acquisition resulting in expensive messages amongst the nodes of the clusters. Hyperion [12] also follows a static-compile approach by compiling the Java bytecode produced to parallel C code which in turn runs on top of an object based DSM.

JavaSplit [40], in contrast to Jackal, compiles Java source code to bytecode appropriate to run in a distributed environment and thus gaining in portability. It employs an object based DSM with a novel protocol called \textit{Multithreaded Scalable Release Consistency (MTS-HLRC)} which is inspired by the \textit{Home-based Lazy Release Consistency (HLRC)} [100] protocol. Synchronization is achieved with the use of distributed locks and associated queues which are passed along with lock ownership.

JavaParty [82] is built on top of Java RMI [76] and provides both a pre-compiler and a runtime for clustering programs. It requires explicit definition of local and remote operations (through the \texttt{remote} keyword). It runs on top of off-the-shelf JVMs and memory consistency is achieved by the use of Java’s synchronization mechanisms.

Terracotta [26] is a commercial state-of-the-art JVM clustering solution which targets mainly enterprise applications. It runs on top of off-the-shelf JVMs and it requires no modifications to the Java source code. The programmer must define the “root” shared object in a xml descriptor before starting the Terracotta server. All referenced objects from the “root” object, including the “root” object, are clustered (called \textit{Distributed Shared Objects (DSOs)}). The synchronization of DSOs is achieved by the use of modified locks. Objects accessed inside a \texttt{synchronized} statement block are logged by the Terracotta runtime and upon exiting the monitor, all modified objects are sent to the rest of the distributed JVM instances.

### 2.1.2 Library Assisted Distributed Java

Besides the various distributed JVM implementations described above, a number of libraries exist which enable Java programs to run in a distributed environment. mpiJava [70] and MPJ Express [69] extend the Java programming language with message passing API which in combination with the inherent multi-threading of Java results in a hybrid model. In addition, a number of libraries which are high-level API wrappers of Java RMI exist. The ProActive [16] suite is an example
of those libraries. ProActive is used as the core communication mechanism of DiSTM and therefore it is explained in detail in Section 4.2.2.

### 2.1.3 Classification

Table 2.1 classifies the various High Performance Java (HPJava) systems. The parameters under which the existing systems have been classified are: additional API needed, runtime system, memory management and synchronization mechanisms. Other parameters such as execution engine or thread management could also be used. However, they have been omitted since they are not directly related to the research described in this thesis.

As shown in Table 2.1, all current HPJava systems employ the traditional lock-based synchronization mechanisms of Java in order to coordinate concurrent accesses to shared objects. On the contrary, DiSTM employs transactional semantics by incorporating a transactional memory execution engine. Furthermore, similar to Terracotta, DiSTM runs on off-the-shelf JVMs making it highly portable and platform independent. Concerning the memory management, DiSTM uses a custom distributed heap which supports object caching and replication (Section 4.3). DiSTM does not employ any additional API or syntax besides the \texttt{@distatomic} annotation. The \texttt{@distatomic} annotations are used, instead of the traditional \texttt{synchronized} statements, in order to denote the boundaries of the memory transactions. Finally, the remote communication mechanisms of DiSTM are based on ProActive’s active objects. The exposed API of ProActive, however, is hidden inside DiSTM and the user is not exposed to it.

### 2.2 Transactional Memory

Transactional Memory (TM) \cite{65} is a new alternative parallel programming model aiming to alleviate the difficulties of the existing explicit synchronization mechanisms (locks, mutexes, semaphores, barriers etc.) by introducing memory transactions. The underlying runtime TM systems automatically detect and resolve concurrent accesses to shared data (objects), abstracting away from the programmers the explicit synchronization constructs. With multi-core becoming mainstream, the need to achieve high performance TM systems lead academia and industry to invest a significant amount of research effort in TM. The majority of the TM research focuses on shared-memory Chip MultiProcessors (CMPs) as the
<table>
<thead>
<tr>
<th>System</th>
<th>Additional API</th>
<th>Runtime System</th>
<th>Memory Management</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>cJVM</td>
<td>No</td>
<td>SSI VM</td>
<td>Custom master-proxy</td>
<td>Lock-based</td>
</tr>
<tr>
<td>Jessica</td>
<td>No</td>
<td>SSI VM</td>
<td>TreadMarks page-based DSM</td>
<td>Lock-based, DSM</td>
</tr>
<tr>
<td>Jessica2</td>
<td>No</td>
<td>SSI VM</td>
<td>Custom Global Object Space</td>
<td>Lock-based</td>
</tr>
<tr>
<td>Java/DSM</td>
<td>Additional API</td>
<td>VM</td>
<td>TreadMarks page-based DSM</td>
<td>Lock-based, DSM</td>
</tr>
<tr>
<td>JavaSplit</td>
<td>No</td>
<td>Static compilation/Parallel binary</td>
<td>Custom object-based DSM</td>
<td>Lock-based, DSM</td>
</tr>
<tr>
<td>Hyperion</td>
<td>No</td>
<td>Static compilation/VM</td>
<td>Custom object-based DSM</td>
<td>Lock-based, DSM</td>
</tr>
<tr>
<td>JavaParty</td>
<td>Additional API</td>
<td>Pre-compilation/VM</td>
<td>RMI</td>
<td>Lock-based</td>
</tr>
<tr>
<td>CoVJM</td>
<td>No</td>
<td>SSI VM</td>
<td>Custom page-based DSM</td>
<td>Lock-based</td>
</tr>
<tr>
<td>mpiJava/MPJ Express</td>
<td>Additional API</td>
<td>Clustered VMs</td>
<td>Message Passing</td>
<td>Lock-based</td>
</tr>
<tr>
<td>ProActive</td>
<td>Additional API</td>
<td>Clustered VMs</td>
<td>Active Objects (RMI)</td>
<td>Lock-based</td>
</tr>
<tr>
<td>Terracotta</td>
<td>No</td>
<td>Clustered off-the-shelf VMs (SSI)</td>
<td>Custom Distributed Shared Objects</td>
<td>Lock-based</td>
</tr>
<tr>
<td>DiSTM</td>
<td>@distatomic annotations</td>
<td>Clustered off-the-shelf VMs (SSI)</td>
<td>Custom Distributed Heap</td>
<td>Software TM</td>
</tr>
</tbody>
</table>

Table 2.1: High Performance Java systems classification.
main current target is to conclude on the TM semantics as well as to investigate and pinpoint the factors that influence the performance of TM systems.

Distributed systems and clusters are widely used in high performance parallel computing for both scientific and enterprise applications. Clusters of multi-cores can benefit from parallel execution both on a single multi-core node and in the cluster as a whole. Currently, the programming of such systems is very similar to those of chip multiprocessors. The investigation of the role TM can play in this domain has just started with the development of prototype distributed transactional memory systems, like the one described in this thesis.

This section introduces the basic ideas and implementation mechanisms of TM systems. Section 2.2.1 introduces the basic TM execution and programming model, including a simple example comparing it to lock-based concurrent programming. Section 2.2.2 explores the design space of TM, while Section 2.2.3 outlines the TM implementation space. Section 2.3 discusses related distributed transactional memory systems, comparing their solutions against the goals of this thesis.

### 2.2.1 Background

Transactional memory (TM) is a parallel programming model which promises to abstract away from the developers the need for explicit memory synchronization. The basic idea, introduced by Lomet [68], concerned the use of database transactions as a means of achieving thread-safe object sharing. The properties of database transactions comply with the **ACID principles** [51]: Atomicity, Consistency, Isolation and Durability.

- **Atomicity**: The property of atomicity defines the execution outcome of transactions. It is required for transactions either to complete execution or, in case of failure, to appear that they have never executed. In case of successful completion, their results become visible to the system.

- **Consistency**: The property of consistency refers to the system’s state during transactions’ execution. Every transaction should transform the system (database, program) from one consistent state to another. Programmers should ensure that transactions maintain the consistency of their systems by properly defining their boundaries.
CHAPTER 2. BACKGROUND

• Isolation: Isolation requires that an executing transaction does not interfere with other concurrently executing transactions. In this way, transactions executed concurrently should produce the same results as if they were executed serially.

• Durability: Durability concerns the persistence of committed transactions to the system. In case of a system failure, committed transactions will survive. Database systems achieve that by maintaining transaction logs for recovery purposes.

Transactional memory applies the aforementioned transactional principles to the main memory level. TM language constructs denote sections of code that constitute the boundaries of memory transactions. The underlying TM runtime system is responsible for maintaining the ACI (Durability is excluded as we refer to main memory which is erased in case of a system failure) principles by detecting and resolving conflicts upon concurrent accesses to shared objects.

In order to compare and contrast transactional memory with lock-based programming a simple shared counter example is presented in Listings 2.1, 2.2.

```java
while(true) {
    try_lock(alock);
    if(have_lock(alock)) {
        a = a + 1;
        release_lock(alock);
        break;
    }
}
```

Listing 2.1: Lock-based increment of shared variable “a”.

```java
atomic {
    a = a + 1;
}
```

Listing 2.2: TM-based increment of shared variable “a”.

The lock-based code requires the programmer to indicate a lock associated with the shared variable (alock). The programmer must check if the lock is acquired, and if so to perform the update. Furthermore, the lock-based example of Listing 2.1 suffers from a pathology associated with lock-based programming
known as deadlock. If for any reason the thread owning the lock fails, no other thread can acquire the lock and will spin indefinitely. The deadlock problem can be solved by adding a time-out mechanism in the process of acquiring the lock, adding more complexity to the development of the shared counter.

In TM, the update operation is placed inside the boundaries of a memory transaction, shown here using the keyword `atomic`, Listing 2.2. The actions of lock acquisition, lock release and deadlock prevention are not necessary anymore since the underlying TM runtime ensures the ACI principles of the memory transactions.

In order for a memory transaction to commit safely a number of actions have to be performed by the underlying TM system:

1. **Data Book-keeping:** The TM implementation logs the objects accessed by every transaction. The objects read by a transaction are added to its *readset* while the objects written are added to its *writeset*.

2. **Conflict Detection:** During execution, conflicts occur when a transaction has a) read an object and another transaction has written to it, or b) written to an object and another transaction read or written to it.

3. **Conflict Resolution:** In case of a conflict being detected, a *Contention Manager (CM)* is invoked in order to resolve the conflict. The conflict resolution between two conflicting transactions results in aborting one of them. The decision of which one to abort is based on the *Contention Management Policy* employed by the CM.

If a transaction successfully completes its execution, i.e. it has not been aborted by any other transaction, then it makes changes to shared objects visible to the whole program. A commit can be a single step or a phase of execution; it may only change a single value (e.g. a transaction status object or Compare-AndSwap (CAS) an indirection object pointer), or it may involve updating all the shared objects if the transaction made updates to copies of the original shared objects. Similarly, an abort operation can either only change a single value, or involve a rolling back mechanism.
2.2.2 TM Design Space

This section presents the design space of the current TM systems. The design choices regard different aspects or parts of TM implementations. The following subsections discuss those aspects.

2.2.2.1 Granularity

The term granularity refers to the memory unit upon which conflicting transactions are detected. With object granularity, conflict detection is performed on data objects. Usually TM implementations that extend object-oriented languages (Java, C#) employ this kind of granularity. Other TM implementations perform conflict detection on words or blocks of words. Word granularity is mostly employed on hardware TM implementations by storing TM oriented meta-data into cache lines.

2.2.2.2 Isolation

TM systems may employ either strong or weak isolation. With weak isolation, the TM runtime does not guarantee transactional semantics for code executed outside the atomic blocks which can potentially lead to data races. On the other hand, when the TM runtime employs strong isolation, code executed outside atomic blocks complies with transactional semantics. However, inconsistent states of the program may appear if the barriers between atomic and non-atomic blocks are incorrectly defined.

2.2.2.3 Access Visibility

Access visibility concerns the visibility of transactions’ bookkeeping information (read/write sets). A transaction’s access is said to be invisible if it is recorded in such a way that it cannot be seen by other concurrent executing transactions. Conversely, if the access is recorded in a way that other transactions can see the access, then it is said to be visible. Both read and write accesses can be either visible or invisible.

Invisible accesses allow transactions to continue executing concurrently even if conflicts exist (as they can not be detected since transactions “hide” their datasets) with a potential cost of time and resources of doomed transactions. Visible accesses can result in less waste as transactions are aborted earlier, but
can instead lead to premature aborts; e.g. transaction A is aborted by transaction B, but later transaction B is aborted by transaction C, and thus transaction A was aborted prematurely. Visibility can be mixed, for example, visible writes and invisible reads.

2.2.2.4 Updating Shared Objects

A transaction upon commit has to make its changes visible to the rest of the system. There are two main approaches to perform updates: deferred and direct updates.

**Deferred Update**  Deferred update TM systems do not directly modify the original objects but private cloned versions (copies) of them. When a transaction attempts to modify an object, a copy of the original object is created and accessed. Subsequent read operations to that object from the same transaction are redirected to the cloned version. Upon commit, if no conflicts are detected, the original object is replaced by the transaction’s private object. Deferred update systems have the advantage of fast abort operations since they only have to discard the privately owned cloned versions of accessed objects. However, commit operations require copying the values of the cloned versions back to memory. This can be achieved in many ways such as changing a status word associated with a transactional object or by CompareAndSwapping (CASing) an indirection pointer of that object.

**Direct Update**  Direct update TM systems modify directly the objects that transactions access. A single copy of any object is preserved and all transactions modify it. Previous versions of the objects are maintained in order to support rollback operations. Every executing transaction, normally, maintains a private log of the genuine objects (previously consistent data) it accesses. Upon successful commit, the log is just discarded. If the transaction is aborted, it copies back to memory the previously consistent data from its log. The advantage of direct update TM systems is that they are fast while committing transactions. On the other hand, in the case of abort, the cost is high as the original objects stored in the log have to be copied back to memory.
2.2.2.5 Conflict Detection

Conflict detection, or validation, is the process of conflict discovery. Access conflicts occur upon read/write or write/write concurrent accesses to the same shared object by two transactions. If conflict detection is performed on every access, it is known as eager validation (as the potential conflict will be revealed instantly). By contrast, if the validation phase is performed only at the end of the transaction’s lifecycle, it is known as lazy validation.

**Eager Validation** If early validation is utilized, conflicts are detected upon accessing an object for a write operation. Eager validation is normally combined with visible writes and visible reads. When a transaction attempts to write to a transactional object it checks for the existence of another writer, and performs conflict resolution if there is one. If the shared object has not been accessed by any other writing transaction, it has to perform conflict resolution for all transactions that may be reading the shared object (may be expensive depending on the size of the readsets as well as the number of the transactions). Using visible writes and visible reads ensures that either a single writer, or multiple readers, exist for each shared object.

The benefits of detecting the conflicts early are the execution time and resources saved on doomed transactions. Hence, if conflicts are dominant, eager validation is preferable. The disadvantage is that validating on every access deteriorates performance. When conflicts are rare, this overhead may be redundant.

**Lazy Validation** In lazy validation, conflicts are detected during the late stages of the transaction. After the transaction has finished executing the code contained in its boundaries, it validates itself against the concurrent running transactions only once. Lazy validation is typically used with invisible writes and invisible reads, which allows a transaction to only detect conflicts with those transactions that have committed before it.

The advantage of lazy validation is that transactions can execute faster as they do not have the overhead of validating in every access. Furthermore, the number of checks performed at the end is identical to the number of distinct object accesses, whereas in eager validation the number of checks may be higher since all accesses are checked as they occur, even for repeatedly accessed objects. The disadvantage of lazy validation is that doomed transactions will spend
unnecessary execution time leading to a waste of resources.

**Early Release** Early release is a technique used in order to improve performance by releasing one or more items from a transaction’s readset before validation. The release of the readset results in minimizing the likelihood of conflicts, and thus reducing the number of aborts. However in order to employ this technique, application-specific knowledge is required to ensure that it will not lead to inconsistent states. Therefore, it has to be manually employed by the programmers.

### 2.2.2.6 Conflict Resolution

After a conflict has been detected, the contention manager (CM) is employed in order to resolve the conflict. Depending on the TM implementation, the CM policy employed may be pluggable or fixed. The flexibility with which conflict resolution can make decisions is governed by the conflict detection and access visibility combination in use. In some cases an active transaction can only detect conflicts with another transaction that has already committed, in which case its only option is to abort. This is the case when all accesses are invisible. If two active transactions conflict then there is a difference between eager and lazy validation. With eager validation, the calling transaction can continue executing after the opponent commits or aborts, but with lazy validation, the calling transaction can only continue executing if the opponent aborts. This is because eager validation checks for a conflict before it occurs, and thus it is safe for the calling transaction to continue even if the opponent has committed, because the calling transaction has not yet accessed the shared object. With lazy validation, the conflict is detected after the conflicting access has been performed, and if the opponent commits then the calling transaction needs to access the new value of shared object on which the conflict occurred, which requires it to abort and restart.

A number of conflict resolution policies have been developed in recent years for eager validation with visible writes, and either visible or invisible reads, that attempt to make decisions that improve performance [57, 88, 89, 49, 48].
CHAPTER 2. BACKGROUND

2.2.3 TM Implementation Space

The majority of the TM implementations target Chip MultiProcessors (CMPs). This section categorizes and explains the current implementations. Section 2.3 elaborates on the distributed TM systems which are directly related to the research of this thesis.

2.2.3.1 Hardware TM

Hardware TM (HTM) implementations [4, 23, 52, 58, 98] extend the architectural support of processors in order to support the execution of memory transactions. This is typically achieved by utilizing a local cache per core that stores transactions’ speculative data. Conflict detection is achieved by modifying the cache coherence protocol. HTM implementations outperform the TM systems of the other two categories. One of their limitations is that transactions cannot access more data than can be stored in the small local cache of a core. Some implementations have improved upon this by adding virtualization support so that main memory can be used to store a transaction’s speculative state, although this decreases performance due to memory latency.

2.2.3.2 Software TM

Software TM (STM) implementations [36, 38, 45, 53, 56, 57, 73, 87, 90] support the execution of transactions solely in software through libraries, or extensions to compilers or runtime environments (e.g. JVM, or CLR). The advantage of STMs is that transactions are unbounded; the number of objects each transaction can access is limited only by physical addressing and storage capacity. However, STMs add execution overhead that result in significantly decreased performance compared to HTMs.

2.2.3.3 Hybrid TM

Hybrid TM (HyTM) implementations [18, 33, 66, 67, 77, 94] attempt to combine the performance advantage of HTMs and the flexibility of STMs. The majority of the HyTM implementations initially execute the transactions in hardware. In case of a failure, they fall back into software. The motivation is that most transactions will be short and execute very quickly in HTM. Furthermore, it would be acceptable to execute the few large ones slower in STM. HyTM research
has mostly focused on how to efficiently support detection of conflicts between transactions executing in HTM and those executing in STM.

2.3 Related Work

This section describes the limited research conducted in the domain of distributed transactional memory systems which is closely related to the research presented in this thesis. Apart from Herlihy’s theoretical paper [55] on distributed transactional memory for metric-space networks and Romano’s theoretical paper [85] on distributed transactional memory for the Cloud, several related systems have been implemented and are described in the following subsections.

2.3.1 PGAS languages

In addition to TM research literature, new parallel programming languages are emerging to enable efficient parallel programming on clusters. The traditional Partitioned Global Address Space (PGAS) languages UPC [37], Co-Array Fortran [79], Titanium [97], as well as the PGAS-based DARPA HPCS languages X10 [31], Chapel [29] and Fortress [3], allow parallel programming while providing a global address space. The PGAS model treats the distributed memory as a single-unified globally addressable memory space. The programmer is required to ensure memory consistency when reading or writing memory locations that may be local or global by dedicated API constructs such as locks or barriers. The newer set of PGAS-based languages (X10\(^1\), Chapel, Fortress) attempt to take advantage of the benefits of TM and thus, they include in their specification constructs like \texttt{atomic\{...\}}. Currently, they do not support a distributed transactional runtime and this is the area of research addressed by this thesis.

2.3.2 Distributed STMs

The distributed transactional memory systems existing in the literature execute memory transactions purely in software and thus they are categorized in the STM domain.

\(^1\)X10’s \texttt{atomic} construct, in contrast to Chapel and Fortress, is not optimistic and does not necessitate a STM runtime.
CHAPTER 2. BACKGROUND

2.3.2.1 Distributed MultiVersioning

Distributed MultiVersioning (DMV) [72] was the first effort to produce a distributed transactional memory implementation. It has been implemented as a modified protocol of a Distributed Shared Memory (DSM) system (TreadMarks SDSM [61]). Relying on a DSM results in page-based granularity of conflict detection and resolution. Its prototype implementation supports two modes: a decentralized “update-anywhere” scheme where update transactions can execute anywhere in the cluster and a centralized “master-centric” approach where write transactions can execute only on the master node. The centralized approach is supported by a scheduler which has a priori knowledge of the nature of the transactions while distributing them. DMV utilizes data multi-versioning [20] allowing read-transactions (i.e. transactions that only read data) to commit concurrently with conflicting write transactions (i.e. transactions that both read and write data) if they have been scheduled before them and of course they have read a valid snapshot of the datum. Each node maintains only one copy of a replicated data item. Every time a commit takes place, a patch with the new values is sent to the rest of nodes which store the patches and apply them lazily upon a page request. In the “update-anywhere” protocol, a transaction that wishes to commit acquires a ticket which serves the role of a global serialization number. Furthermore, the committing transaction broadcasts its pages to the rest of the cluster aborting any conflicting local transactions before committing. In the “master-centric” approach, the scheduler schedules all write transactions on the master node and distributes the read transactions to the rest of the worker nodes. Since all write transactions are executed on the master node, there are no conflicts between write transactions. Concerning read transactions, they may block waiting for a previously scheduled transaction to release a lower version of a page in order to create their own. Finally, the scheduler is version-aware of the data that exist or are about to be created at each replica and schedules the read-only transactions accordingly, in order to reduce the possibility of conflicts. Local conflicts within a node are resolved with the help of a variant of the two-phase locking protocol [20].

2.3.2.2 Cluster-STM

Cluster-STM [24] is a distributed software transactional memory implementation targeting large-scale clusters. It mainly focuses on data partitioning techniques
of transactional meta data. It neither employs multi-versioning nor relies on any DSM system (whereas DMV relies on DSM). Each data item has a home node which is responsible for maintaining its consistency. Data caching is not transparent and has been delegated to the application user level. Furthermore, access on transactional data is exclusive, disallowing a single-writer/multiple-readers approach. Cluster-STM, does not provide any forward guarantees, besides the absence of deadlock. This may result in livelock\(^2\). Finally, it does not support multithreaded execution on a single node.

2.3.2.3 D2STM

D2STM [32] is a distributed multi-versioning [20] software transactional memory implementation built on top of JVSTM [28]. It utilizes the notion of “Virtual Boxes” in order to maintain multiple versions of transactional data. Unlike DMV, in D2STM multiple versions of the same object may exist in replicas. Local conflicts are resolved before remote validation via JVSTM’s internal TM engine. If a transaction successfully passes its local validation phase, it Atomic Broadcasts (ABcasts) [35] its writeset and its readset (encoded in a Bloom Filter [22]). If the remote validation phases succeed, the committing transaction successfully updates its data (locally or remotely) atomically.

2.3.2.4 Sinfonia

Sinfonia [2] employs memory transactions (so called “minitransactions”) in order to add transactional memory functionality on distributed systems. The commit protocol is a two-phase protocol [20] and it has been used in building a cluster file system as well as group communication service. Sinfonia however assumes static datasets (i.e. the transactions’ datasets are a priori known) thus limiting its generality.

Besides the published related work already discussed, a number of technical reports exist describing work in progress on distributed transactional memories outlined in the following sections.

\(^2\)A pathology in which conflicting transactions keep aborting each other thwarting forward progress.
2.3.2.5 GTM

Global Transactional Memory [93] targets large scale distributed systems of thousands of nodes. It is being implemented as a STM library for the Chapel language [29]. It supports a dynamic programming model combining: a) thread-level parallelism, b) Single Process Multiple Data (SPMD) parallelism and c) remote procedure call invocation (RPC). The algorithm used to support TM has the following characteristics: weak isolation, read versioning and deferred update. GTM is currently work in progress in collaboration with Cray.

2.3.2.6 ST-DSM

Software Transactional Distributed Shared Memory [34] focuses mainly on algorithms for object prefetching and caching. The compiler has been modified in order to calculate statically path expression prefetches. In this way, batch object requests are sent to fetch remote objects, thereby minimizing network traffic. The transactional coherence protocol used is based on the standard two-phase commit.

2.3.2.7 DSTM

Decentralized STM (DSTM) [21] introduces a decentralized object-based algorithm that utilizes multi-versioning as described in [84, 28]. The commit protocol is a two-phase randomized consensus protocol. DSTM is currently being incorporated within a distributed Java Virtual Machine.

2.3.3 Classification

Table 2.2 classifies the various Distributed Software Transactional Memory systems\(^3\). As shown in Table 2.2, DiSTM is the only system that supports multiple transactional coherence protocols. This is due to its flexible design and enhances its modularity. Similar to D2STM, ST-DSM and DSTM, DiSTM supports object granularity. Furthermore, DiSTM, unlike DMV, does not require any DSM implementation. Finally, multithreading support is also provided by DiSTM unlike Cluster-STM.

\(^3\)The Sinfonia system as well as the PGAS languages have been omitted from the classification as they do not directly relate to DiSTM.
### 2.4 Summary

DiSTM is a Java clustering solution that supports software transactional memory. The system is placed between the High Performance Java (HPJava) domain and the Transactional Memory (TM) domain. Hence, this chapter has introduced both the relevant, closely related, HPJava systems and the applicable research on TM. In addition, the design and implementation space of TM has been introduced. Finally, all the relevant systems have been classified and compared against DiSTM. The next chapter delves into the internals of DiSTM, describing both the architecture of the system and the transactional coherence protocols implemented.

<table>
<thead>
<tr>
<th>System</th>
<th>Granularity</th>
<th>Protocol</th>
<th>Memory Management</th>
<th>Communication</th>
<th>Multithreading (Within a node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMV</td>
<td>Page</td>
<td>2PL Broadcast</td>
<td>TreadMarks DSM</td>
<td>Unix standard libs</td>
<td>yes</td>
</tr>
<tr>
<td>Cluster-STM</td>
<td>Block of words</td>
<td>2PL</td>
<td>Custom (Exclusive write access)</td>
<td>GASNet [25]</td>
<td>no</td>
</tr>
<tr>
<td>D2STM</td>
<td>Object</td>
<td>Multi-versioning</td>
<td>“Virtual Boxes”</td>
<td>GCS [30]</td>
<td>yes</td>
</tr>
<tr>
<td>GTM</td>
<td>Word</td>
<td>2PL</td>
<td>Read versioning</td>
<td>GASNet</td>
<td>yes</td>
</tr>
<tr>
<td>ST-DSM</td>
<td>Object</td>
<td>2PL</td>
<td>PGAS style DSM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSTM</td>
<td>Object</td>
<td>Multi-versioning</td>
<td>Global Accessible Objects</td>
<td>Unix standard libs</td>
<td>yes</td>
</tr>
<tr>
<td>DiSTM</td>
<td>Object</td>
<td>Three Centralized</td>
<td>Distributed Heap</td>
<td>ProActive (RMI)</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decentralized</td>
<td>Master Heap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Distributed Software Transactional Memory systems classification.
Chapter 3

Programming Model

3.1 Introduction

This chapter discusses the programming model used in DiSTM in two sections. The first section discusses the methods employed by the programmer in order to transactify applications to run on DiSTM as well as the usability of the distributed atomic collection classes. The second section explains how DiSTM clusters the Java applications as well as the work queues used in order to distribute the transactional jobs.

3.2 Transactification of generic applications

DiSTM employs a strategy similar to [56] for denoting transactional objects. In order to implement a transactional object, its interface has to be implemented and annotated with the @distatomic annotation. The runtime system detects that the interface refers to a transactional object and dynamically constructs a class by injecting the transactional code transparently to the user. For example, in order to implement the transactional version of a shared counter, the following interface has to be implemented (Listing 3.1).

```java
@distatomic
public interface SharedCounter {
    public int getValue();
    public void setValue(int value);
}
```

Listing 3.1: Shared Counter Atomic Integer
As depicted in Listing 3.1, the programmer has only to provide the interface of the atomic integer counter. Consequently, the runtime system will construct the transactional class.

The @distatomic interface has a number of methods that are inherited by all transactional objects. These methods are used by DiSTM in order to inject necessary information such as cluster-aware objects IDs, node-owner IDs, etc.

DiSTM employs TM semantics in object level and does not allow non-transactional accesses of transactional objects. If at any time the program attempts to access a transactional object outside the scope of a transaction, an exception will be raised and the program will exit. Although this fact may constrain the programmability of applications running on DiSTM, it provides strong memory consistency guarantees. In order to specify the boundaries of a memory transaction, the technique shown in Listing 3.2 has to be employed.

```java
public class GenericThread extends DiSTMThread {

    @Override
    public void run() {
        DiSTMThread.doIt(new Callable<Void>() {
            @Override
            public Void call() {
                ...
            }
        });
    }
}
```

Listing 3.2: Define Transactions’ Boundaries

As shown in Listing 3.2, the Callable object passed as parameter in the DiSTMThread’s static doit method denotes the boundaries of DiSTM’s transactions. The code included in the run method of the callable will run transactionally on the cluster. Concerning the example of the shared integer counter, DiSTM’s execution threads are similar to the one shown in Listing A.4.

The current programming model used in DiSTM utilizes “transactional jobs” [6]. Instead of having transactional code enclosed in atomic blocks (as used in other TM systems), the transactions are wrapped into abstract execution objects, called transactional jobs. In turn, these transactional jobs are enqueued in each node’s work queues. A concrete implementation of a transactional job can be
thought of as a list of parameters, and a pointer to the method that invokes the appropriate **atomic** block with the supplied parameters.

During bootstrap, the node which is responsible for loading all JVM instances of the cluster creates and distributes in a round robin fashion the transactional jobs to the nodes of the cluster. After the distribution of the jobs completes, the nodes start processing the transactional jobs from their local work queues and the worker threads pass the transactional objects to the `DiSTMThread.doIt()` methods which execute the transactions pointed by their correspondent transactional objects. The benefit of using such a skeleton based approach in jobs’ creation, distribution and execution is that numerous orthogonal optimization techniques such as work-stealing [7] or concurrency control [8] can be applied transparently to the user.

### 3.3 Porting applications to DiSTM

In order for a generic Java application to run on DiSTM, two parts have to be modified. The first part concerns the main function of the program while the second concerns the creation of the transactional jobs.

Regarding the main function the following alterations should be effective. First, the class which implements the main function of the program has to extend DiSTM’s `distm.Main` class and call the superclass’ main function from the application’s main function (Appendix A presents a complete example of transforming the shared counter example from single node lock-based application to its DiSTM version.). The inherited main function contains all the necessary calls for bootstrapping DiSTM, creating the jobs, distributing the jobs, starting execution and shutting down the system at the end of the execution. All the aforementioned tasks excluding the creation of the transactional jobs are transparent to the user and are handled by DiSTM.

Concerning the creation of the transactional jobs, applications that are ported to DiSTM have to provide an implementation of the `createTxJobs` method. The user should provide an implementation for the creation of transactional jobs. Transactional jobs should extend the `TransactionalJob` class which implements the `java.io.Serializable` interface in order to enable the distribution of the tasks over the network. The transactional jobs for the shared atomic integer example is illustrated in Listing A.3.
Appendix A contains the complete source code examples for converting a single-node multithreaded program to a transactional clustered version running on DiSTM.
Chapter 4

DiSTM Architecture

4.1 Introduction

This chapter describes the architecture of the Distributed Software Transactional Memory (DiSTM) [62, 63, 64] system. DiSTM is a software solution for clustering multiple off-the-self Java Virtual Machine (JVM) instances on commodity clusters. It provides a Single System Image (SSI) view of the system to the user, on which programs run transparently in parallel on multiple nodes. The novelty of DiSTM stems from the nature of the synchronization mechanisms employed when threads access concurrently shared objects. Traditional constructs such as locks, semaphores, barriers etc. have been replaced by protocols which employ transactional memory semantics in order to resolve the conflicts.

DiSTM provides a model in which transactional objects are defined fairly easily by annotating them with the @distatomic annotation, ensuring transactional coherence of all atomic objects. The algorithms employed upon transactions’ commit phases depend on the protocol used. DiSTM offers four different transactional coherence protocols which have different characteristics. Broadly, the implemented protocols fall into two categories; centralized and decentralized. Generally the categorization of the implemented protocols apply to the memory management of DiSTM as well, with one exception (Section 4.4).

The rest of the chapter is organized as follows: Section 4.2 describes the generic design of DiSTM, while Section 4.3 describes in detail the memory management of the system as well as the implemented distributed atomic collection classes. Section 4.4 presents the internals of the transactional coherence protocols. Finally, Section 4.5 summarizes the chapter.
4.2 DiSTM Generic Design

This section presents DiSTM’s generic design entailing the backbone of the system which is common in all protocols described later. Figure 4.1 shows the basic components of a single instance of DiSTM on a node.

As shown in Figure 4.1, the three core parts are: the transactional engine that transforms the \texttt{@distatomic} annotated interfaces to transactional objects (Section 4.2.1), the remote communication layer which is responsible for the remote method invocations on the cluster (Section 4.2.2) and finally the transactional coherence protocols (Section 4.4).

4.2.1 Transactional Engine

The transactional engine of DiSTM is responsible for transforming plain objects into transactional objects. The technique employed in DiSTM is similar to the one employed in DSTM2 [57]. Transactional objects are declared as \texttt{@distatomic} annotated interfaces. A simple example of a transactional Integer object declaration is shown in Listing 4.1.

```java
@distatomic
public interface AtomicInteger {
    public int getValue();
    public void setValue(int value);
}
```

Listing 4.1: Atomic Integer

Figure 4.1: DiSTM generic architecture
As shown in Listing 4.1, a typical AtomicInteger object is defined by simply creating an annotated interface providing the getter and setter methods. A restriction of this model is the type of the input parameters and return values of the declared methods. DiSTM currently permits only scalar values or atomic objects.

DiSTM's transactional engine at runtime parses the annotated interface and creates on-the-fly a bytecode-written Java class which employs transactional semantics. The process of the bytecode writing is assisted by the BCEL library [13]. The internals of the transactional coherence protocol, specified during bootstrap, are injected in the transactional objects and therefore their life-cycles adhere to the logic of the protocols.

4.2.2 Remote Communication

The communication amongst the nodes is achieved by the use of the ProActive [16] framework (high level API wrapper of RMI). The remote requests are invoked on the active objects. The active objects have their own thread of execution, can be distributed, and constitute the basic building blocks of the ProActive framework. Figure 4.2 depicts a typical active object.

As shown in Figure 4.2, a node that references an active object can perform a remote request on it. The request is placed in the body queue of the active object. In turn, each request is served serially by the active object. Each node in
the DiSTM framework has a number of active objects (a fixed number for dedicated purposes) serving various requests. Those requests have been decoupled and logically assembled in different active objects in order to avoid bottlenecks. Generally, active objects serve one request at a time and hence congestion may occur. The decoupling of the remote requests in the DiSTM framework resulted in the creation of various active objects per node (Table 4.1 summarizes the active objects created at each node per protocol). Furthermore, a remote method invocation can either be synchronous or asynchronous. Upon a synchronous request the caller node’s thread waits until the active object returns the value. In the case of an asynchronous request, the caller node’s thread continues execution and will block, in order to wait for the return value, only when the return value is read. Depending on the protocol’s stages, DiSTM can utilize either synchronous or asynchronous requests to achieve higher performance. (During the protocols’ descriptions in Section 4.4, the type of the requests are explained.)

### 4.3 Memory Management

DiSTM has the ability to operate in two modes: the first is by following a centralized approach while the second is by following a decentralized approach. The memory management also falls into these two categories.

As already stated (Section 4.2.1), all transactional objects in DiSTM are defined by annotating their corresponding interfaces with the @distatomic annotation. All distributed atomic objects, irrespectively of the mode DiSTM operates in, inherit two function definitions from the super interface. The first field is the Object ID (OID) which uniquely identifies each transactional object in the cluster. The second field is the Home Node (HOMENODE) which states the node id (NODEID) that created the specific transactional object. The following two subsections describe the memory management when running DiSTM in centralized and decentralized mode.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Active Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCC</td>
<td>Validator (per node), Data Updater(per node)</td>
</tr>
<tr>
<td>Single Lease</td>
<td>Lease Acquire, Lease Release (both at master node), Data Updater(per node)</td>
</tr>
<tr>
<td>Multiple Leases</td>
<td>Lease Acquire, Lease Release (both at master node), Data Updater(per node)</td>
</tr>
<tr>
<td>Anaconda</td>
<td>Validator (per node), Data Updater(per node)</td>
</tr>
</tbody>
</table>

Table 4.1: Active Objects per Protocol
4.3.1 Centralized approach

Figure 4.3 depicts the memory modules of DiSTM when operating in centralized mode.

When operating in centralized mode, the master node maintains a consistent view of the transactional data while each worker node keeps a cached view of the data (which may become stale during execution). All transactional objects have the master node’s ID as their HOMENODE field. Furthermore, the master node assigns the OIDs to all statically created transactional data. Upon a transaction’s commit, the committing transaction updates the data on the master node. Consequently, the master node is responsible for updating the cached data of the worker nodes, aborting any conflicting transactions. During bootstrap, the master node creates the transactional data and consequently copies them to the worker nodes. For fixed sized data structures (such as distributed atomic arrays or plain distributed atomic objects), the master node assigns OIDs to the data which in turn are delegated to the worker nodes upon the data caching.

For dynamic-sized data structures, the situation becomes more complicated: the master node, in the same way as dealing with fixed sized data structures, initializes the data structures and copies them to the worker nodes. During
execution if a thread executing in a worker node creates a previously non-existent data element and manages to successfully commit, upon updating the master node’s consistent view of data, the dynamically assigned OID is copied to the atomic object. Furthermore, the object’s HOMENODE field (although it was created in a worker node) is assigned the NODEID value of the master node.

Section 4.3.3 provides a more detailed description of the internals of the distributed atomic collection classes implemented in DiSTM.

### 4.3.2 Decentralized approach

Figure 4.4 depicts the memory modules of DiSTM when operating in decentralized mode.

As shown in Figure 4.4, when operating in decentralized mode, there is no master node involved in the execution\(^1\). Each worker node has three distinct memory heaps. The local data heap stores all the non-transactional local objects of the worker node. The owned transactional data heap stores the distributed atomic transactional data that each worker node creates, and therefore it is the

---

\(^1\)For bootstrapping purposes a worker node is selected in order to perform the initialization of the rest of the nodes.
home node of them. Finally, the cached transactional data heap stores all the
distributed atomic objects that have not been created on the worker node but
accessed from it. This implies a remote request being sent to the cluster in order
to fetch and cache those objects.

When operating in decentralized mode, the transactional data are partitioned
amongst the worker nodes. Each node at any time maintains a consistent view of
the data which it owns (the node is the home node of the data). Upon accessing
an object which does not reside in the caller node’s heap (the caller node is not
the home node of the object), if a cache miss is detected a remote request is
broadcast. The receiver node, which is the home node of the requested object,
returns the object to the caller node. The caller node caches the returned object
and accesses it. If the requested object does not exist in the cluster, the caller
node creates it and becomes the home node of the object.

Section 4.3.3 provides a more detailed description of the internals of the dis-
tributed atomic collection classes implemented in DiSTM.

4.3.3 Distributed Atomic Collection Classes

This section describes in detail the distributed atomic collection classes imple-
mented while porting the benchmarks (see Section 5.2.2) to DiSTM. The descrip-
tion of each class is logically separated into two subsections: the first section
concerns the functionality of the class when running in centralized mode, while
the second that when running in decentralized mode.

4.3.3.1 Distributed Atomic Singleton Objects

Centralized Mode: The implementation of singleton objects is fairly straight-
forward when run in centralized mode. Upon initialization the master node cre-
ates the object, assigning its OID. The object is copied to the worker nodes where
locally running threads can access it.

Decentralized Mode: When running in decentralized mode, the node that
first accesses (initializes) the object assigns the OID to it and becomes its HOMEN-
ODE. Any subsequent request to that object from any remote node will result in
caching it from its home node.
4.3.3.2 Distributed Atomic Arrays

Centralized Mode: Similarly to singleton objects, the master node creates and initializes the whole array assigning OIDs per array element. The array is cached on the worker nodes and threads can access it normally.

Decentralized Mode: In this mode the array is partitioned amongst the nodes of the cluster. The partitioning can be achieved in several ways such as HORIZONTAL, VERTICAL or BLOCK; see Figure 4.5.

The home-owner of each array partition initializes it and assigns OIDs to the elements contained in it. If a thread attempts to access an array element which is not in the boundaries of the local partition owned by the node where it resides, a remote request is broadcast and the corresponding array element is fetched and cached on the local node.

4.3.3.3 Distributed Atomic LinkedList

The type of linked list required for the implemented benchmarks prohibits the existence of duplicates. Hence, a lookup phase precedes every addition in order to discover if the potentially inserted element already exists in the list. The following descriptions concern the list implemented which is not a generic linked
Centralized Mode: During the initialization phase, the head of the linked list is created at the master node and is cached on the worker nodes. In this way all worker nodes can access the head of the list and perform operations on it. During an insertion to the list, the previous value as well as the new value of the head pointer are sent to the master node in order to commit the data to the linked list residing on the master node (always the consistent view). The master node, being responsible for maintaining the consistent state of the cached versions of the lists of the worker nodes, multicasts the new values to them aborting any conflicting transactions. Removals and retrievals from the list are performed similarly to insertions.

Decentralized Mode: In this mode the situation differs significantly from that of the centralized mode. Now each node has its own head of the list. The list is fully distributed amongst the nodes and operations include remote requests to remote nodes. Figure 4.6 presents the state diagrams of the three major operations on a list (Insertion, Removal and Lookup).

Upon an insertion, the list is traversed in order to discover if the element already exists. If the element is present, it is returned and the operation completes. If the element is not found, a remote request is broadcast. If the element exists on a remote node, it is fetched and cached on the caller node and the operation completes with the return of the fetched element. If the remote request fails (the requested element does not exist in the cluster), the caller node inserts the element to the list and completes the operation returning the value. The removal operation as shown in Figure 4.6 differs slightly from the insert operation. The difference is that upon a failed lookup and a successful remote request an insert operation is simulated before the element is removed. Finally, the lookup operation has two phases. First, the part of the distributed list residing on the local current node is checked. If the element exists it is returned. On the other hand, an insertion operation is simulated in order to fetch the element from the remote nodes. If the insertion operation succeeds (i.e. the element exists in the cluster), it is fetched and cached on the local current node and returned (and finally removing it from the list). On the contrary, a null value is returned.
CHAPTER 4. DISTM ARCHITECTURE

4.3.3.4 Distributed Atomic HashMap

The Distributed HashMap has been modeled as an non-atomic array of distributed atomic linked lists. Therefore, its operation is similar to that of the distributed atomic linked list. Figure 4.7 illustrates the structure of the hashmap. The `bucket[]` array holds the heads of the distributed atomic linked lists. When a transaction accesses an element of the hashmap, the key is hashed and the corresponding bucket index is retrieved in order to perform the operation.

**Centralized Mode:** In this mode, the master node knowing in advance the size of the hashmap creates and initializes the `bucket[]` array as well as the heads of the atomic lists. Those values are copied to the worker nodes and hence the application threads can operate on the hashmap. Transactions’ commit procedures follow the same strategy of the corresponding centralized versions of the collection classes described above.

**Decentralized Mode:** In this mode, all worker nodes create and initialize
privately owned bucket[] arrays assigning distinct OIDs per list head. This scheme can be regarded as a collection of distributed atomic linked lists running in decentralized mode. After the indexing of the key (hashing the key object associated with the value\(^2\)) and the retrieval of the corresponding bucket[] index, the operations adhere to the state diagrams of Figure 4.6.

### 4.4 Transactional Coherence Protocols

This section describes in detail the transactional coherence protocols implemented in DiSTM. Currently, four transactional coherence protocols are implemented in DiSTM: three centralized and one decentralized.

As already mentioned in Section 2.2, in a typical multiprocessor configuration, the lifecycle of a memory transaction normally has two major stages. The first stage is the validation() stage where any potential conflicts, with other concurrently running transactions, are discovered and resolved. The second major stage, which always follows a successful validation() phase, is the the commit() stage. In this stage, the transaction makes its data visible to the system. In the case of a shared memory, single-node multiprocessor system, the validation() phase entails all threads executing in the system. In a clustered environment the validation() phase.

\(^2\)Please note that in order for the scheme to work correctly, all distributed instances of the hashmap must provide a platform independent, identical hashing function.
CHAPTER 4. DISTM ARCHITECTURE

phase must include all threads running on remote nodes. Hence, the validation() function must either provide a unified solution that includes both locally and remotely running threads or include two distinct phases. The first phase, local-Validation(), should discover and resolve potential conflicts with locally running transactions, while the second phase, remoteValidation(), should discover and resolve potential conflicts with remote running transactions. DiSTM employs both the aforementioned strategies: the decentralized approach follows the unified validation approach whereas the centralized protocols employ the one with the two distinct validation phases.

4.4.1 Centralized Protocols

This section illustrates DiSTM’s centralized protocols: TCC, Single Lease and Multiple Leases. The data management also follows the centralized approach when one of these protocols is employed. Transactional Coherence and Consistency protocol \(^3\) is a modified version of Stanford’s Transactional Coherence and Consistency protocol [52]. The remaining two centralized protocols utilize network leases [47].

4.4.1.1 TCC

Figures 4.8 and 4.9 illustrate the TCC protocol schematically and as a state diagram respectively. These two figures will be used in order to explain the protocol. As shown in Figure 4.9, TCC has four major stages after the transaction enters its commit phase: local validation, remote validation, local commit and update global data.

Local Validation: The local validation phase in TCC as well as the rest of the centralized protocols is performed similarly to the way DSTM2 [57] performs its validation phase. Each transaction when accessing a shared object creates a cloned version of the object which it keeps private. The cloned object is modified and if the transaction manages to commit successfully, the old object is replaced by the cloned version with a CompareAndSwap (CAS) instruction. The local validation phase is eager, meaning that when a transaction attempts to read

\(^3\)Although TCC is a decentralized protocol in the way it operates, it is intentionally placed in the centralized protocols section as it is operated when the data management is achieved in the centralized manner
Figures 4.8: Stages of the TCC protocol.

Figures 4.9: TCC protocol state diagram.
or write any object currently modified by another transaction, the conflict is discovered and resolved at that time. Eager validation prohibits multiple cloned objects being created per transactional object. Only one transaction at any time can modify a transactional object.

**Remote Validation:** The remote validation, on the contrary, is lazy. Any transaction that successfully passes its local validation phase broadcasts its write sets during an arbitration phase before committing. All remote transactions, executed concurrently, compare their read/write sets with those of the committing transaction and if a conflict is detected, one of the conflicting transactions aborts in order for the other to commit safely. In order to maintain an ordering of the transactions on the cluster, a “ticketing” mechanism has been employed. Each transaction, before broadcasting its read/write sets in order to be validated against transactions running on remote nodes, acquires a “ticket” (global serialization number) from the master node. The role of the “ticket” is to assist the contention manager upon conflict detection between transactions running on different nodes and to avoid livelocks. The contention management policy adopted is the oldest-commit-first policy (oldest in terms of remote validation time – which transaction attempts to remotely validate its read/write sets first).

Upon remote validation, see Figure 4.8 Step 1, a transaction’s read and write sets (incoming transaction) are compared against the read and write sets of the transactions executing on a remote node (local transactions) resulting in three possible scenarios:

1. **There is no conflict** — No transaction is aborted and the `remoteValidation()` method returns true, so the caller can commit safely.

2. **There is a conflict with a “younger” local transaction** — In that case, a conflict is detected against a transaction which has a greater “ticket” number than the incoming transaction. That means that the local transaction has acquired the ticket after the transaction it is validated against (incoming transaction). The local transaction is considered to be “younger” and therefore has to be aborted\(^4\). Instead of aborting the local transaction immediately, its id is stored in a temporary queue. Each transaction with a greater “ticket” (than the incoming transaction) will be stored in the

\(^4\)Transactions that do not have a ticket are also considered younger.
queue. The decision of whether to abort the transactions in the queue takes place when the incoming transaction has been validated against all local transactions. When the validation of the incoming transaction has finished and there has been no conflict with a local transaction with a smaller “ticket” number (older) the transactions stored in the queue are aborted. This is due to serial validation. Each transaction’s read/write sets are validated serially against the read/write sets of other transactions. Furthermore, all transactions which attempt to be validated against the transactions running on another node are queuing up and each one performs the remoteValidation() function serially. Therefore, there may be a case where the first transaction to be validated against is younger while the second one is older. If we were to abort the first transaction immediately then the incoming transaction could be aborted by the second (older) transaction and, hence, we would have unnecessarily aborted the first one. If the incoming transaction conflicts only with “younger” locally executed transactions, the remoteValidation() method returns true so the caller can proceed in committing its transaction.

3. There is a conflict with an “older” transaction — In that case, a conflict is detected against a transaction which has a smaller “ticket” number than the incoming transaction, i.e. the local transaction is older and the incoming younger transaction should abort. Consequently, the remoteValidation() function returns false and the queue holding the conflicting younger local transactions (if there are any) is released.

**Local Commit:** The local commit phase follows the validation phase. A transaction that has passed the remote validation phase successfully attempts to make its changes visible locally (at the node on which it is running). If the transaction has not been aborted, while waiting for the remote request of the remote validate phase to return, it commits locally (Figure 4.9). From that point on, any local running transaction accesses the new transactional objects that have just been committed.

**Update Global Data:** After a transaction has made its changes visible locally, it updates the global dataset kept at the master node. In turn, the master node eagerly updates all the cached datasets on the rest of the nodes of the cluster
(Figure 4.8 Step 4). The home node of the committing transaction is omitted from this step as its cached dataset has been already updated during the previous phase. Upon updating the cached datasets, a validation phase occurs which invalidates the transactions that have read “dirty” values from the cached dataset. Any transactions discovered during the invalidation phase are aborted and re-executed after the node gets a consistent view of the data.

A very important aspect of the TCC, and all the centralized, protocols is the fact that the update global data phase is a blocking request. The request that updates the cached data of the nodes is nested in the update global data request. Thus, any other committing transaction blocks and waits for the current committing transaction to commit. In this way, any conflicting remote transaction that “escaped” from the remote validation phase will be aborted while the committing transaction updates the cached datasets.

4.4.1.2 Serialization Lease

Figures 4.10 and 4.11 illustrate the Single Lease protocol schematically and as a state diagram respectively. As shown in Figure 4.11, the single lease protocol has four major stages after the transaction enters its commit state: local validation, lease acquisition, local commit and update global data.

The local validation, local commit and update global data stages are similar to the ones described in the TCC protocol. The role of the lease is to serialize the transactions’ commits over the network and therefore to avoid the expensive broadcasting of transactions’ read/write sets for validation purposes. Each transaction that passes the local validation phase attempts to acquire the lease from the master node, Figure 4.10 Step 1. If no other transaction has acquired the lease, the committing transaction acquires the lease. Any other transaction that tries to acquire the lease, after the transaction from worker node A has acquired it, will block and wait its turn to commit after adding itself to a queue kept at the master node (as shown in Figure 4.11, the transaction that failed to acquire the lease will spin, checking either if the lease has been assigned to it or if it has been aborted by the committing transaction). The role of the queue is to maintain the order of the transactions waiting to acquire the lease. When the transaction (lease owner) of worker node A commits, it updates the global data at the master
Figure 4.10: Stages of the Single Lease protocol.

Figure 4.11: Single Lease protocol state diagram.
node and then releases the lease, Figure 4.10 Step 2. Before the lease is released, the cached datasets of the worker nodes are updated with the new values of the committed transaction. A validation phase while fetching the new data aborts any conflicting local transactions. After the lease is released, the master node retrieves the next transaction from the queue and attempts to assign the lease to it, Figure 4.10 Step 4. If the transaction has not been aborted (in the process of updating its cached dataset), it acquires the lease and proceeds in committing. On the contrary, if the transaction has been aborted, the master node attempts to assign the lease to the next transaction retrieved from the queue.

The advantage of the serialization lease is the minimization of the broadcast messages exchanged in TCC. The disadvantages are that transactions block waiting to be assigned the lease, that there may be attempts to assign the lease to aborted transactions and the bottlenecks created upon acquiring and releasing the lease from the master node.

### 4.4.1.3 Multiple Leases

Figures 4.12 and 4.13 illustrate the multiple leases protocol schematically and as a state diagram respectively. As shown in Figure 4.13, the multiple leases protocol has four major stages after the transaction enters its commit state: local validation, lease acquisition, local commit and update global data.
CHAPTER 4. DISTM ARCHITECTURE

The local validation, local commit and update global data stages are similar to the ones of the TCC/Single Lease protocols. In this scheme, multiple leases are assigned to transactions that attempt to commit. After a transaction passes the local validation phase, it attempts to acquire a lease from the master node, Figure 4.12 Step 1. Unlike the previous lease scheme where only one lease was assigned at a time, in this scheme multiple leases are available for committing transactions. When a transaction attempts to acquire a lease, a validation phase is performed at the master node. Each transaction that attempts to acquire a lease is validated against each transaction (stored in the pool) that currently owns a lease. If there is no conflict, the transaction acquires a lease and proceeds in committing after adding itself to the pool of transactions that hold a lease. If a conflict is discovered, the transaction aborts and restarts. Upon successful commit, Figure 4.12 Step 3, the transaction updates the global data at the master node and in turn the master node updates the cached data at the worker nodes. The transactions that unsuccessfully attempted to acquire a lease (and have been aborted) will be re-executed and consequently try to acquire a lease during their commit stage. There is no limit in the number of leases that can be assigned on the cluster. As long as no conflicts are detected at the master node leases can be assigned. However, controlling the number of leases could be a way to control the network traffic on the cluster. The advantage of this scheme is that multiple transactions can commit concurrently. The disadvantages are the fact that an extra validation step has been added to the master node as well as the bottlenecks created upon acquiring and releasing the leases.

Figure 4.13: Multiple Leases protocol state diagram.
4.4.2 Decentralized Protocol

This section describes the decentralized protocol employed in DiSTM with the code name **Anaconda**. The Anaconda protocol permits transactions to commit concurrently, overcoming the serialization of the update global data stage of the centralized protocols. Furthermore, the Anaconda protocol, in contrast to the centralized protocols, employs a unified validation stage for committing transactions which includes locally and remotely running transactions. The protocol employs a lazy validation approach compared to the eager-local/lazy-remote validation approach of the centralized protocols. In order to achieve that a set of helping data structures is maintained per node.

4.4.2.1 Anaconda Data Structures

The data structures required by the Anaconda protocol are: the Transactional Object Cache (TOC) and the Transactional Object Buffer (TOB). Each node maintains a single TOC that is shared by all threads executing on that node. The TOC provides caching for remotely fetched transactional objects. Furthermore, it maintains book-keeping information of executing transactions. Figure 4.14 illustrates a TOC. The first field is the Object ID (OID) that maps to a particular entry. The second entry is the Node ID (NID) of the home node of the object assigned the OID. Unless the NID of an entry in the TOC is equal to that node, it is a cached copy of an object residing on another node. Maintaining the NID of the objects can assist in identifying the home owners of every object cached. The Cache field maintains a list of all the nodes that have requested and retrieved a copy of an object (and consequently stored it in their caches). This information is used to assist conflict resolution. The Lock TID field is a lock associated with each entry and it is acquired during transactions’ commit stage. Finally, the Local TID field is a list of all the local transactions currently accessing this object.

The second data structure, the TOB, is maintained per transaction (Figure 4.15). After accessing an object for a write operation, a cloned copy of the object residing in the TOC is created and stored in the TOB. Thereafter read operations will be redirected to the cloned version of the object. The TOB actually serves the role of maintaining transactions’ book-keeping information.
4.4.2.2 Anaconda Commit Procedure

Figures 4.16 and 4.17 illustrate the Anaconda protocol schematically and as a state diagram respectively.

The Anaconda protocol has a three phase commit stage (Figure 4.16). As a transaction progresses through the phases it becomes more certain that there are no conflicts. During the first phase, the transaction interacts with the local TOC to ensure exclusive access to each object in its writeset. Exclusive access is ensured by acquiring locks of the objects existing in the transaction’s writeset. Depending on where the node-owner of each object is, locks can be acquired either on the node on which the transaction is running or on remote nodes. In the second phase, the objects in the writeset are multicast to those nodes that have cached copies of them. The idea is not to update yet, but to validate against the remote TOCs. In the third final phase, the transaction cannot be aborted by any other transaction and can commit the new values stored in its local TOB.
CHAPTER 4. DISTM ARCHITECTURE

Figure 4.16: Stages of the Anaconda protocol.

Figure 4.17: Anaconda state diagram.
4.4.2.3 Commit Process

**Phase 1: Lock acquisition** During this phase (Figure 4.16, Step 1) a transaction is required to acquire the locks for each object in its writeset. The writeset of each transaction is processed and the objects contained in it are grouped according to their home nodes. Batch requests are sent to each node starting from the local node. This is done in order to save remote requests upon failed local lock acquisition. The response from the remote TOCs to each request contains a list of the nodes having a cached copy of any object in the writeset. Note that this phase is liable to run into deadlock if implemented naively, as the gathering of all object locks is no longer an atomic operation. The rules used to overcome this issue are discussed in Section 4.4.2.5.

**Phase 2: Validation phase** Having successfully completed Phase 1, a transaction has the list of nodes which contain remotely cached versions of the objects that are part of its writeset. Thus, the modified objects (i.e., the OIDs as well as the new values) are multicast (Figure 4.16, Step 2) to the nodes in the list. Upon arrival in a remote node, a validation phase then takes place aborting any conflicting transactions local to that node. The validation phase concerns only the transactions pointed to by the Local TID field part of a remotely affected TOC. When a conflict is detected, the TIDs are compared in order to decide which transaction should be aborted using the “older-commit-first” policy (i.e., the transaction with the larger TID is aborted). In order to enhance performance of the validation phase, bloom filters are utilized to encode the readsets of the transactions. In this way, we try to minimize the validation phase time as it is a blocking request against the transaction that performs this phase as well as the transactions queued waiting their turn to validate against this node. If the validating transaction is aborted it revokes any locks (if it had acquired any) and removes its TID from any entry in the TOC.

**Phase 3: Update Objects** Having successfully completed Phase 2, the transaction cannot be aborted by any other transactions and therefore can safely proceed in swapping or updating the old objects with the new ones (Figure 4.16, Step 3). This can be done safely as the committing transaction still holds the locks on the objects being updated, not allowing any other transactions to fetch and cache, i.e. neither read from nor write to them. Any such request will result in a negative acknowledgment by the TOC. The requesting transaction will continue to retry until it gets aborted or until the committing transaction releases the lock.
CHAPTER 4. DISTM ARCHITECTURE

Upon an object update the TOC is responsible for updating all the cached copies of this object. This can be done in different ways (invalidate vs. update protocol). In the invalidate protocol, the transactions have to discover by themselves any potential stale object and consequently abort themselves, while in the update protocol the system eagerly patches all the cached values and eagerly aborts any conflicting transactions. Consequently, the new patches are sent to the nodes that hold cached copies of the objects. In those nodes receiving the patches, the local executing transactions that are accessing those objects are validated against the incoming writeset. Any local conflicting transactions are aborted. Finally, the updating transaction revokes all locks and, if this phase takes place at the node where the transaction belongs, the TID of the transaction is removed from any entries in the TOC

4.4.2.4 Simple Commit Example

Figure 4.18 demonstrates a simple commit example of two transactions, T1 and T2 executing on Nodes 1 and 2 respectively. For clarity, in the example we assume that Nodes 1 and 2 are the home nodes of objects A and B respectively. In this example T1 commits successfully while T2 is aborted and subsequently restarted.

In Step 1, T1 attempts to read object A with OID(A). As OID(A) is not present in T1’s TOB, T1 checks if OID(A) exists in the current node by inquiring at TOC 1. After the successful inquiry at TOC 1, T1 stores a reference of OID(A) in its readset (marking the Bloom filter). Furthermore, T1 adds itself to OID(A) Local TIDs entry in the TOC. The same procedure takes place upon T2’s effort to load object B. If the objects were not residing in the nodes, a remote request for fetching the object would be sent across the cluster.

In Step 2, T1 speculatively writes to object A by creating a private cloned copy pA’ of object A and storing its reference in T1’s TOB. At the same time, T2 reads object A by fetching it from Node 1. This request will add Node 2 to object’s A Cached field in TOC 1. In this way the home node of any object is aware of any remote nodes that have fetched a copy of a particular object. This information will be used later to maintain TM coherence between the various cached copies of objects residing on different nodes.

In Step 3, T1 starts its three stage commit phase while T2 continues execution.

---

5The removal of a transaction’s TID from any entries in the TOC is also performed if the transaction is aborted.
Figure 4.18: Anaconda protocol simple commit example.
During phase 1, T1 requests the lock for object A. Although a translation of OID(A) exists in both TOCs, the lock acquisition will take place only at the home node of OID(A). In this way, T1 gets the lock by adding itself to the Lock TID field for TOC 1. Now any lock request to this particular lock would cause the contention manager to be invoked and one of the two transactions to be aborted. TOC 1 will respond to T1’s request for OID(A) lock by sending back the list of the nodes that have a cached copy of OID(A). Hence, the return node list for T1 will contain Node 2 as it has a cached copy of object A. Upon successful acquisition of all locks, T1 proceeds to Phase 2. Phase 2 entails the multicast of T1’s writeset to the node list received from Phase 1. A validation step will abort either any conflicting local transactions or T1. In this example T2 is aborted by T1 and acknowledges the verification. Upon receiving positive replies from phase 2, T1 proceeds to phase 3 by CompareAndSwapping (CASing) its status from ACTIVE to UPDATING. If the update of the flag is successful, no other transaction can abort T1 and it moves to the final phase. In Phase 3, T1 sends an update-objects request to the same nodes to which it multicast in Phase 2 (note that the objects themselves were already sent in Phase 2). When Node 2 receives the requests it replaces the old objects with the new ones and performs a validation check only to the transactions contained in the Local TID field of its OID(x) mapping contained in the updating dataset. In the meantime, the aborted transaction T2 releases any locks acquired, if any. Finally, both transactions revoke their TIDs for the corresponding Local TID fields of their TOCs.

The life-cycle of a request for an object is depicted in Figure 4.18 (a), (b), (c) and (d). First, in step (a), the transaction’s TOB is checked in order to discover whether or not the object has been already read/written. In case of a miss (Step (b)), a request is sent to the local TOC. If the request is successful the object is added in the transaction’s TOB. On the contrary, a request is sent to remote nodes (Steps (c),(d)) which in turn can either be successful (Step (d)) or not (Step (e)).

4.4.2.5 Contention Cases

There are phases within the protocol where contention can occur. This section describes them and explains how the contention is resolved.

**Lock Acquisition Contention phase** The first phase where contention can occur is during commit phase 1. During this phase transactions are required
to gather all locks sufficient to cover all modified objects and the locks are gathered in the order in which they appear in the TOB. It is possible that multiple transactions requiring multiple locks can enter a scenario similar to the deadlock achieved in the dining philosophers problem. A typical scenario might be: T1 holds the lock for object A, and needs the lock for object B. T2 holds the lock for object B, and needs the lock for object A. In such a scenario, when T1 requests the lock for object B, the TOC containing that lock forwards a message to the owner (T2) informing it that the lock must be revoked, because T1 has a higher priority (T1’s TID timestamp is smaller than T2’s). T2 will release the lock and abort.

**Multicast Contention** During commit phase 3 it is possible that a transaction will multicast its intent to modify an object contained in another transaction’s readset. If this causes a violation, i.e. it is not a false conflict, a contention manager will be invoked in order to resolve the conflict. Anaconda allows the plug-in of different contention managers. After the contention manager invocation, only one transaction will continue in committing while the other will be aborted.

**TOC trimming** Upon successful lock acquisition of a transaction in the commit stage, the number of nodes to which the writeset of the committing transaction is multicast is determined by the number of nodes that have a cached version of the objects. The TOC has the responsibility to multicast the committing transaction’s writeset to the corresponding nodes and the extra validation step will reveal any conflicts. However, the TOCs can grow large slowing down any operations on them. This problem can be easily tackled by periodically trimming the TOC, i.e. removing records that have not been accessed lately.
4.5 Summary

The DiSTM framework is a clustering JVM solution with software transactional memory support. DiSTM’s flexible implementation allows a variety of transactional coherence protocols to be implemented. Currently, four protocols have been implemented: three centralized (TCC, Single Lease, Multiple Leases) and one decentralized (Anaconda). The data management of DiSTM corresponds to the nature of the protocols. Therefore the transactional data can follow either a centralized or a decentralized approach. Table 4.2 presents the characteristics of each protocol.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Local Validation</th>
<th>Remote Validation</th>
<th>Data Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCC</td>
<td>eager</td>
<td>lazy</td>
<td>centralized</td>
</tr>
<tr>
<td>Single Lease</td>
<td>eager</td>
<td>single lease (serialized)</td>
<td>centralized</td>
</tr>
<tr>
<td>Multiple Leases</td>
<td>eager</td>
<td>multiple leases</td>
<td>centralized</td>
</tr>
<tr>
<td>Anaconda</td>
<td>lazy</td>
<td>lazy</td>
<td>decentralized</td>
</tr>
</tbody>
</table>

Table 4.2: DiSTM’s transactional coherence protocols characteristics.
Chapter 5

Evaluation

5.1 Introduction

The following subsections discuss the performance evaluation of DiSTM. All the protocols proposed in this thesis are comparatively evaluated. Furthermore, DiSTM is evaluated against the Terracotta JVM clustering system which support lock-based distributed execution. In detail, Section 5.2.1 describes the hardware platform used for the evaluation. Section 5.2.2 describes the benchmarks used for the evaluation. Finally, Section 5.4 presents the results and compares all the proposed protocols against each other as well as the lock-based implementations of the benchmarks.

5.2 Experimental Platform

5.2.1 Hardware

The platform used for the evaluation is a cluster with five nodes (one master and four worker nodes). The worker nodes are 4x dual core (8 core) AMD Opteron 2.4GHz systems with 16GB RAM, openSUSE 10.1, and Java HotSpot 1.6 64-bit using the parameters \(-Xms8024m -Xmx18000m\). The master node is utilized when the centralized protocols are employed maintaining the consistent view of the transactional data (Section 4.4.1). When the decentralized protocol is employed the master node is not involved during the execution. Its only purpose is to bootstrap the rest of the system. As each node has a maximum of 8 cores, all benchmarks are executed using 1 to 8 threads per node in order to minimize the
costs of thread switching. In total the maximum number of threads used in the experiments is 32.

5.2.2 Benchmarks

A variety of benchmarks, some of them well-known benchmark suites, have been used to evaluate novel proposals in existing TM publications. The majority of them fall broadly into three categories: micro-benchmarks, parallel benchmarks with trivial transactional characteristics, and parallel benchmarks with non-trivial transactional characteristics.

Micro-benchmarks are based on common abstract data structures such as linked lists and hash tables. The benchmarks falling into this category can be useful in analyzing specific components of TM implementations. However, they cannot represent full-fledged commercial applications, which would likely have transactions that access several different data structures or perform complex calculations. The TM research community still uses micro-benchmarks as necessary, but has begun to divert to parallel benchmarks with non-trivial transactional characteristics.

Parallel benchmarks with trivial transactional characteristics are typically derived from existing established parallel benchmark suites such as SPLASH-2 [92], Java Grande [39], and SPECjbb2000 [91], which have been converted from using locks to using transactions. However, these benchmarks have been heavily tuned by experts to maximize scalability, and consequently have few conflicts when translated into transactional benchmarks. Consequently, these benchmarks provide little insight into a TM implementation’s behavior in the presence of non-trivial conflict patterns. Furthermore, it is unlikely that the average programmer, at whom TM is aimed, will have the skills and experience to develop such well-tuned applications.

Finally, parallel benchmarks with non-trivial transactional characteristics have been developed from the TM community in order to specifically evaluate the emerging TM implementations. These include STMBench7 [50], STAMP benchmark suite [77], and Lee-TM [9, 96].

The evaluation of a distributed TM implementation differs significantly from the evaluation of a TM implementation targeting a Chip MultiProcessor (CMP). In a distributed environment, the validation procedure entails all transactions running on remote nodes. Therefore, expensive messages across the network are
exchanged amongst the nodes of the cluster. In order to benefit from distributed parallel processing, the applications used should exhibit computational intensity in order to hide the delay introduced by the remote communication. Neither micro-benchmarks nor trivial transactional benchmarks suit this scenario. Hence, the benchmarks used to evaluate the DiSTM are derived from the third category.

In this chapter, several TM oriented benchmarks are used to evaluate the distributed transactional coherence protocols proposed in this thesis. The benchmarks which are commonly used in experimental evaluations of TM publications are: Lee-TM, the STAMP (version 0.9.5) benchmark suite (KMeans, Genome, and Vacation) and a transactional implementation of Conway’s Game of Life [19].

Most benchmarks have been modified to create and distribute transactional jobs to work queues during benchmark initialization, which is excluded from the execution time. Additionally, KMeans and Genome also create transactions dynamically during execution: details are provided in their respective descriptions later. Below, each benchmark is described briefly.
5.2.3 Lee-TM

Lee-TM is a transactional circuit routing application based on Lee’s path connection algorithm [86]. Circuit routing makes connections automatically between pairs of points that have been loaded from a file, and sorted by ascending straight-line length to reduce ‘spaghetti’ routing. Routing is performed on a 3D grid that is implemented as a multidimensional array, and used to represent a layered printed circuit board (PCB). Each array element is called a grid cell.

Routing consists of two phases: expansion and backtracking. Figure 5.1 shows the result of a successful expansion phase. During expansion, each thread attempts to find a route from the source point to the target point of a connection by performing a breadth-first search, reading grid cells, and recording the depth count of each cell in a thread-local scratch space representing the 3D grid. However, if the grid cell is occupied, then it is marked as such in the thread-local space, and does not have its depth count recorded. Figure 5.2 shows the result of a successful backtrack phase. If a route is found, backtracking lays the route by performing write operations to grid cells starting from the target point and tracing back to the source point. The backtrack path is constructed by following a trail of successively smaller depth counts in the scratch space. Concurrent
Figure 5.3: Sample routing of the mainboard.txt circuit, a realistic microcode processor.

routing requires reads and writes to the grid cells to be performed transactionally.

A second version of Lee-TM has been implemented that uses early release (see Section 2.2.2.5). This version removes grid cells from the read set during the breadth-first search. Two transactions may be routable in parallel, i.e. the sets of grid cells occupied by their routes do not overlap, but because of their spatial locality, the breadth-first search of one transaction reads grid cells to which the second transaction writes its route, thus causing a conflict. Removing grid cells from the read set during the breadth-first search eliminates such false conflicts.

Lee-TM is fully parallel, with no phases of serial execution. Conflicts occur between transactions that expand and backtrack into the same grid cell. As routes are sorted in ascending length order, it is reasonable for the likelihood of transactional conflict to increase as execution proceeds, since later routes will have larger average read and write sets, and longer average execution durations.

Lee-TM is executed with an input file as a parameter, and a flag to enable or disable the early release version. Lee-TM, and Lee-TM-ER are used to distinguish between non-early release, and early release versions, respectively. The input file used in the evaluation is mainboard.txt [96], which is a real circuit used in
routing research. The circuit represents a microcode processor and consists of 1506 routes, and a sample routing is shown in Figure 5.3.

### 5.2.4 KMeans

KMeans assigns objects into a number of clusters. The application loads objects from an input file, and then works in two alternating phases. One phase assigns objects to their nearest cluster. The other phase re-calculates cluster centers based on the mean of the objects in each cluster (initially cluster centers are assigned randomly). Execution repeatedly alternates between the two phases until two consecutive iterations generate, within a specified threshold, similar cluster centers. Objects and cluster centers can have an arbitrary number of dimensions, as clustering does not require visual representation in 2D or 3D form.

The phase that assigns objects to clusters is executed in parallel, and the phase that re-calculates cluster centers is done serially. Conflicts occur when two transactions attempt to assign an object to the same cluster; if objects are close to each other, then they are more likely to be assigned to the same cluster, increasing the likelihood of conflict. Furthermore, the fewer clusters, the higher the likelihood of conflict. Transactional jobs consist of objects to be assigned to clusters, and are distributed during initialization. Threads synchronize at a distributed barrier once all objects have been assigned, and the serial phase is then executed. The serial phase performs the threshold check, and if further execution is required it distributes new jobs in order to repeat the cluster assignment phase, after which all threads start executing again.

KMeans is executed with a minimum of two parameters: the input file, and the number of clusters to which objects should be assigned. The input file supplied by the suite is used, called `random10000_12`, which has 10000 randomly spaced objects, each with 12 dimensions. Different numbers of clusters are assigned in KMeansLow and KMeansHigh to demonstrate different levels of contention.

### 5.2.5 Genome

Genome is a gene sequencer that rebuilds a unique gene sequence from a large number of equal-length overlapping gene segments. The gene sequence and gene segments are objects consisting of a character string. Additionally, the gene segments contain a pointer to the start segment (the segment identified as the
The aim of gene sequencing is to construct a unique gene sequence from a pool of overlapping gene segments. The application begins by constructing the unique gene sequence using random ASCII characters, and then constructing a number of fixed length gene segments that are substrings of the gene sequence, constructed by selecting a random starting character.

Figure 5.4 gives a high-level illustration of how gene sequencing works. The application executes in three phases. The first phase removes duplicate segments by transactionally inserting them into a hash set. The second phase attempts to link segments by matching substrings: checking if the tail substring of one segment (identified as the tail-segment) is equal to the head substring of another segment (identified as the head-segment). If two segments are found to overlap then linking is a three-step process, all wrapped in one transaction. First, the next segment pointer of the tail-segment is made to point to the head-segment. Second, the overlap length in the tail-segment is set equal to the number of characters in the overlapping substring. Third, both segments are removed from the hash set, as multiple gene segments may match and result in conflict. The matching is done in a for-loop that starts by searching for the largest substring overlap (gene-segment-length-minus-one characters, since duplicates were removed in the first phase), down to the smallest overlap (one character). In the third phase, a single thread passes over the linked chain of segments to output the rebuilt gene sequence.
The execution of Genome is completely parallel except for the third phase. The likelihood of conflict rises as execution progresses in the middle phase since smaller overlaps will lead to more matches. The initial duplicate removing phase is an optimization to quickly reduce the global likelihood of conflict.

Genome is executed with the following input parameters: gene sequence length, length of overlapping gene segments, and number of overlapping gene segments. The default parameters are 1834, 24, and 19430. Genome’s threads synchronize at a distributed barrier once all matches for a given overlap length have been attempted, at which point new jobs are distributed to attempt matching at the next lower length.

### 5.2.6 Vacation

Vacation simulates a travel booking database for cars, hotels, and flights. Each database table (cars, hotels, and flights) is represented as a transactional hash table. There are several types of transactions in this benchmark. Transactions can be customers making reservations, which changes the availability of the booked item in the database. Customers are linked lists of reservations. Transactions can also be suppliers who change the availability of items in the database. Finally, transactions can be customers canceling all their bookings. Customer booking transactions attempt to update multiple items in a single transaction, which simulates a customer, for example, booking a package holiday that requires a car, hotel, and flight for all the necessary dates. However, Vacation randomly selects the multiple items a customer will book, and thus it is slightly less realistic. The transaction type is selected randomly for each job.

Vacation is a fully parallel benchmark. Conflicts arise when customers try to change the same item (e.g. availability of a car on a certain date). The more items a single customer transaction tries to book, the more shared data it touches, which increases the likelihood of conflict. The smaller the database, the higher the likelihood of conflicts as well.

Vacation is executed with the following parameters: the total number of transactions to be committed, the number of items a customer books in a single transaction, and the size of each database table. The default parameters are 4096, 60 and 16384.
5.2.7 Game of Life

The transactional implementation of Game of Life (GLife) is a cellular automaton which applies the rules of Conway’s Game of Life. The benchmark consists of a two-dimensional grid of cells. Each cell can have one of two possible states, live or dead. During an iteration (referred as generation) each cell interacts with each neighboring cells in order to determine its state. The decision of whether the cell will be live or dead after a generation is based on the following rules:

1. Any live cell with fewer than two live neighbors dies.
2. Any live cell with more than three live neighbors dies.
3. Any live cell with two or three live neighbors lives on to the next generation.
4. Any dead cell with exactly three live neighbors becomes a live cell.

The initial pattern constitutes the seed of the system. The grid is divided amongst the threads of the system and each thread applies the rules to the cells contained in its partition. The decision making of each cell constitutes a transaction. Conflicts occur when two transactions try to apply the rules on neighboring cells (boundaries of grid partition). The initial seed of the grid has been generated in order to produce a 50% write transactions workload. GLife is executed with the following parameters: grid size and number of generation. The default parameters used are: 100 columns, 100 rows and 10 generations.

5.3 Transactional Profiles

Table 5.1 summarizes the benchmarks used in the evaluation along with their parameters. In order to give a first insight to the behavior of each benchmark, their correspondent average readset and writeset sizes have been computed (Tables 5.2, 5.3). Furthermore, Figure 5.5 illustrates the average ratio of ReadSet to WriteSet (i.e. the average number of reads divided by the average number of writes), which will be used as a reference point in the remaining of the chapter.
Table 5.1: Benchmarks’ parameters

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeeTM</td>
<td>Lee without early release</td>
<td>early_release: false, input_file: mainboard, 600x600x2 circuit with 1506 transactions</td>
</tr>
<tr>
<td>Lee-TM-ER</td>
<td>Lee with early release</td>
<td>early_release: true, input_file: mainboard, 600x600x2 circuit with 1506 transactions</td>
</tr>
<tr>
<td>KMeansHigh</td>
<td>KMeans with high contention</td>
<td>min_clusters: 20, max_clusters: 20, threshold: 0.05, input_file: random10000, 12</td>
</tr>
<tr>
<td>KMeansLow</td>
<td>KMeans with low contention</td>
<td>min_clusters: 40, max_clusters: 40, threshold: 0.05, threshold: 0.05, input_file: random10000, 12</td>
</tr>
<tr>
<td>GLife</td>
<td>Game of Life</td>
<td>gene_length: 1834, number of segments: 19430, segment_length: 24</td>
</tr>
<tr>
<td>Vacation</td>
<td>Vacation</td>
<td>number of transactions: 4096, table size: 16384, items size: 60</td>
</tr>
<tr>
<td>Genome</td>
<td>Genome</td>
<td>grid_size: 100x100, generations: 10</td>
</tr>
</tbody>
</table>

Table 5.2: Average readset size of committed transactions (number of transactional objects).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>4 threads</th>
<th>8 threads</th>
<th>12 threads</th>
<th>16 threads</th>
<th>20 thread</th>
<th>24 thread</th>
<th>28 thread</th>
<th>32 thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lee-TM</td>
<td>59119</td>
<td>58845</td>
<td>58877</td>
<td>58961</td>
<td>58908</td>
<td>58996</td>
<td>59279</td>
<td>58606</td>
</tr>
<tr>
<td>Lee-TM-ER</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>KMeansHigh</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>KMeansLow</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>GLife</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Vacation</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Genome</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

As shown in the tables 5.2 and 5.3, LeeTM as well as LeeTM-ER have the largest read/write sets from all benchmarks. By contrast, the rest of the benchmarks have small read/writesets. Furthermore, LeeTM has the largest RStoWS ratio with a factor of almost 600. It is expected that benchmarks with large read/write sets will be more computational intensive and hence, they should benefit more from distributed execution.
CHAPTER 5. EVALUATION

5.4 Speedups and Execution Times

This section presents the speedups and the execution times of the four distributed transactional coherence protocols. The speedups are calculated normalized to executing one thread per node execution times. Section 5.4.1 presents the speedups achieved per transactional protocol in order to show their scalability. In Section 5.4.2 the executions times of the benchmarks are presented. The results are grouped per benchmark in order to understand which protocols perform better. Finally, Appendix B contains the number of commits and aborts per benchmark per protocol which will be used throughout the discussion.

5.4.1 Speedups

5.4.1.1 TCC

Figure 5.6 depicts the speedups of the benchmarks when the TCC protocol is employed. Accordingly, Table 5.4 presents their execution times.

Concerning LeeTM (see Figure 5.6), no speedup is observed. In fact, as more threads are added, the execution slows down to 0.7x and the execution time is increased by 30% (Table 5.4). The cause of the slowdown is the high abort rate
of the benchmark as more threads are added. As shown in Table B.1 and Figure B.1, as more threads are added the number of aborts increases and hence the performance degrades. Concerning the computational intensity of LeeTM, Figure 5.11 illustrates the breakdown of the percentages of time spent in each stage of a transaction. The result of adding more threads to the system is the decrease of the percentage of time spent for computation. As more threads are added, more transactions are executed and therefore, more remote requests are being dispatched amongst the nodes of the cluster. As already explained in Section 4.2.2 the active objects serve those remote requests serially. Consequently, as the number of threads increases the number of the pending remote requests increases too resulting in the transactions spending a larger percentage of their time in remote requests.

Concerning LeeTM-ER (see Figure 5.6), the maximum speedup achieved is 2.4x at 28 threads. Furthermore, as shown in Table 5.4, the execution time decreases as more threads are added in contrast to LeeTM. Both the observed speedup and the decrease in the execution times result from the low abort rate (Table B.5, Figure B.2) and the less time spent in remote requests. By applying the “early-release” technique, LeeTM’s abort rate decreases dramatically. The benefits from that are two-dimensional. First the likelihood of conflicts is minimized and thus the number of aborts decreases and second the transactions spend less time in the remote validation phase as their writesets are not compared against the readsets of the rest of the transactions. Consequently, the time spent in remote requests is determined only by the writesets. Therefore, in LeeTM-ER, as shown in Figure 5.11, the percentage of time spent in remote requests is smaller than in LeeTM.

Table 5.3: Average writeset size of committed transactions (number of transactional objects).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>WriteSet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 threads</td>
</tr>
<tr>
<td>Lee-TM</td>
<td>104</td>
</tr>
<tr>
<td>Lee-TM-ER</td>
<td>104</td>
</tr>
<tr>
<td>KMeansHigh</td>
<td>21</td>
</tr>
<tr>
<td>KMeansLow</td>
<td>22</td>
</tr>
<tr>
<td>GLife</td>
<td>1</td>
</tr>
<tr>
<td>Vacation</td>
<td>3</td>
</tr>
<tr>
<td>Genome</td>
<td>5</td>
</tr>
</tbody>
</table>
CHAPTER 5. EVALUATION

Concerning KMeans, both KMeansHigh and KMeansLow configurations exhibit similar behavior with slight differences. Up to 8 threads a slight speedup (1.1x for KMeansHigh and 1.2x for KMeansLow) is observed (Figure 5.6). After that, the performance deteriorates down to 0.6x for KMeansHigh and 0.7x for KMeansLow. In addition, as shown in Table 5.4, the execution times increase after 8 threads. The deterioration in performance stems from the high abort rate both KMeansHigh and KMeansLow exhibit (Tables B.25, B.3 and Figures B.21, B.4). Furthermore, both benchmarks, as shown in Figure 5.11, spend the majority of their time in remote requests. Both the high abort rate and the fact that KMeans’ transactions are not computationally intensive increase the execution times even more. As transactions are waiting for their remote requests to complete, the aborted transactions have the time to restart and conflict again with the same transaction repeatedly (due to spatial locality). This pathology of Transactional Memory has been identified by Ansari et al. [11], [7] and solutions have been proposed. In distributed environments, where sometimes the percentage of time spent in remote requests is much higher than the percentage of time spent for computations, this pathology can have severe impact to performance as in both KMeansHigh and KMeansLow.

Concerning GLife, a slight speedup which peaks at 28 threads with 1.24x is observed (Figure 5.6). In contrast to KMeans, GLife has a low abort rate as shown in Table B.9 and Figure B.7. However, similarly to KMeans, GLife is also a non-computationally intensive benchmark. Consequently, although performance does not deteriorate as in KMeans (due to low abort rate) the scalability is highly influenced by the remote requests.

Vacation is similar to GLife in terms of abort rate and computational intensity (Table B.13, Figure B.5 and Table 5.4). Hence, it behaves similarly by showing a slight speedup of 1.16x at 16 threads (Figure 5.6).

Finally, Genome also falls into the category of GLife and Vacation. It exhibits low abort rate (Table B.17, Figure B.6) and it is also non-computationally intensive (Table 5.4). Therefore, as shown in Figure 5.6, it has a maximum speedup of 1.15x at 20 and 24 threads. Furthermore, as depicted in Table 5.4, Genome is even less computationally intensive than GLife and Vacation resulting in lower speedups.
### Table 5.4: Benchmarks’ execution times (seconds) - TCC protocol.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeeTM</td>
<td>551.8</td>
<td>574.6</td>
<td>575.2</td>
<td>621.2</td>
<td>645</td>
<td>686.2</td>
<td>750.8</td>
<td>786.8</td>
</tr>
<tr>
<td>LeeTM-ER</td>
<td>454</td>
<td>309</td>
<td>276</td>
<td>222</td>
<td>205</td>
<td>195</td>
<td>190</td>
<td>201.2</td>
</tr>
<tr>
<td>KMeansHigh</td>
<td>434.8</td>
<td>409.8</td>
<td>432.6</td>
<td>476.2</td>
<td>532.8</td>
<td>567.8</td>
<td>632</td>
<td>681.8</td>
</tr>
<tr>
<td>KMeansLow</td>
<td>337.8</td>
<td>290.8</td>
<td>327.8</td>
<td>383</td>
<td>385.2</td>
<td>397</td>
<td>463</td>
<td>463.8</td>
</tr>
<tr>
<td>GLife</td>
<td>334.4</td>
<td>321.4</td>
<td>314.2</td>
<td>314.8</td>
<td>314.6</td>
<td>301.6</td>
<td>268</td>
<td>301.2</td>
</tr>
<tr>
<td>Vacation</td>
<td>32.2</td>
<td>30</td>
<td>27.8</td>
<td>28.2</td>
<td>28.6</td>
<td>29.2</td>
<td>29</td>
<td>28.2</td>
</tr>
<tr>
<td>Genome</td>
<td>368.3</td>
<td>331.6</td>
<td>331</td>
<td>330.6</td>
<td>329.3</td>
<td>329</td>
<td>336.3</td>
<td>340</td>
</tr>
</tbody>
</table>

![LeeTM-TCC Speedup](image.png)

![LeeTM-ER-TCC Speedup](image.png)
CHAPTER 5. EVALUATION

![Graph](image)

**KMeansHigh-TCC**

![Graph](image)

**KMeansLow-TCC**

![Graph](image)

**GLife-TCC**
Figure 5.6: Benchmarks’ speedups - TCC protocol.
Figure 5.7: Breakdown by transaction stage of the execution times. - TCC protocol
5.4.1.2 Single Lease

Figure 5.8 depicts the speedups of the benchmarks when the Single Lease protocol is employed. Accordingly, Table 5.5 presents their execution times.

Concerning LeeTM, although a slight speedup is observed initially, performance deteriorates reaching a slowdown of 0.74x \(^1\) at 32 threads (Figure 5.8). Similarly to LeeTM-TCC, LeeTM-SL spends the majority of its time for computational purposes as shown in Figure 5.9. However, the number of aborts in LeeTM-SL is smaller (Table B.2, Figure B.1) due to the serialization technique employed by the SL protocol. Consequently, the serialization of the transactions’ commits results in steady performance up to 16 threads (with a slight speedup) in contrast to LeeTM-TCC. However, the congestion that occurs on active objects as more threads are added eventually slows down the system.

Concerning LeeTM-ER, a maximum speedup of 2.47x is achieved at 28 threads (Figure 5.8). Similarly to LeeTM-SL, LeeTM-ER-SL spends the majority of its time for computational purposes as shown in Figure 5.9. However, the number of aborts in LeeTM-SL is smaller (Table B.6, Figure B.2) due to the serialization technique employed by the SL protocol. Hence, the speedup up to 28 thread. After that, performance drops due to congestion on active objects.

Concerning KMeans (Figure 5.8), both KMeansHigh-SL and KMeansLow-SL exhibit slowdowns 0.5x and 0.6x respectively (although a slight speedup of 1.20x is observed). As in TCC, KmeansHigh-SL and KMeansLow-SL spend the majority of their time in remote requests (Figure 5.9). Concerning the number of aborts, although KMeansHigh-SL has less aborts than KMeansLow-SL (Tables B.26, B.22), the APC ratio of KMeansHigh-SL is higher as the number of commits is smaller than in KMeansLow-SL (Figures B.3, B.4).

GLife’s performance, although steadily deteriorating down to 0.34x at 32 threads (Figure 5.8), fluctuates amongst the number of threads used. This is due to scheduling issues of neighboring conflicting cells. As already mentioned in Section 5.2.7, in GLife remote conflicts (conflicts between transactions running on different nodes) occur only on the boundaries of the blocks of the distributed array. Hence, depending on the number of threads running on the system, the number of blocks created varies resulting in different scheduling of the conflicting cells. Consequently, as shown in Table B.10, the number of remote aborts (aborts

\(^1\)Slowdown is the same as speedup - the difference being whether the factor is <1 or >1.
between transactions running on different nodes) varies, depending on the number of threads. In general, the number of aborts increases (Table B.10, Figure B.7) but that is due to the increasing number of the local aborts which does not influence performance as much as remote aborts.

Vacation does not exhibit any speedup. On the contrary, as shown in Figure 5.8, it exhibits a constant slowdown down to 0.3x. Furthermore, as GLife, Vacation is a non-computational intensive benchmark as depicted in Figure 5.9. Hence, the performance is heavily influenced by the messages being exchanged during execution. The result of adding more threads to the system in the loss in performance, due to congestion on the active objects, despite the fact that Vacation has a low abort rate (Table B.14, Figure B.5).

Finally, Genome behaves similarly to Vacation with a maximum slowdown of 0.61x at 32 threads, Figure 5.8. This is due to two reasons. Both traffic congestion, Figure 5.9, and high abort rate (Table B.18, Figure B.6).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeeTM</td>
<td>554</td>
<td>533</td>
<td>515.8</td>
<td>528.4</td>
<td>558.6</td>
<td>610.8</td>
<td>641.6</td>
<td>745</td>
</tr>
<tr>
<td>LeeTM-ER</td>
<td>479.8</td>
<td>314.4</td>
<td>240.6</td>
<td>230.8</td>
<td>217</td>
<td>206.4</td>
<td>193.8</td>
<td>265.4</td>
</tr>
<tr>
<td>KMeansHigh</td>
<td>461.2</td>
<td>422</td>
<td>392</td>
<td>421.2</td>
<td>471.6</td>
<td>537</td>
<td>615.2</td>
<td>769.8</td>
</tr>
<tr>
<td>KMeansLow</td>
<td>370.2</td>
<td>434</td>
<td>450</td>
<td>477.4</td>
<td>562.6</td>
<td>611</td>
<td>680.8</td>
<td>741.8</td>
</tr>
<tr>
<td>GLife</td>
<td>463.6</td>
<td>509</td>
<td>561</td>
<td>551</td>
<td>556</td>
<td>610</td>
<td>585</td>
<td>630</td>
</tr>
<tr>
<td>Vacation</td>
<td>58</td>
<td>70</td>
<td>59</td>
<td>68</td>
<td>73</td>
<td>74</td>
<td>88</td>
<td>87</td>
</tr>
<tr>
<td>Genome</td>
<td>492.6</td>
<td>605.6</td>
<td>618.3</td>
<td>627.6</td>
<td>635.1</td>
<td>726.3</td>
<td>769.3</td>
<td>801.2</td>
</tr>
</tbody>
</table>

Table 5.5: Benchmarks’ execution times (seconds) - SL protocol
CHAPTER 5. EVALUATION

![Graph showing speedup vs. number of threads for LeeTM-SL and LeeTM-ER-SL]
Figure 5.8: Benchmarks’ speedups - Single Lease protocol
Figure 5.9: Breakdown by transaction stage of the execution times. - SL protocol
5.4.1.3 Multiple Leases

Figure 5.10 depicts the speedups of the benchmarks when the Multiple Leases protocol is employed. Accordingly, Table 5.6 presents their execution times.

In the Multiple Leases (ML) protocol the transactions that wish to commit, acquire multiple leases from the master node in contrast to the Single Lease (SL) protocol where only one transaction can commit at a time. This is achieved by the addition of one extra validation step at the master node (before the lease acquisition). The committing transactions, upon lease acquisition, send their readsets and their writesets to the master node where the validation phase is performed. Upon a successful lease acquisition, and hence a successful commit, the writesets of the committing transactions currently residing at the master node are reused as the updated objects are sent to the remaining of the nodes.

The transfer of the transactions’ readsets can severely influence performance. An example of such a scenario is the comparison of the speedup between LeeTM-ML and LeeTM-ER-ML. In LeeTM-ML, the large readsets of the transactions are being sent to the master node, where in LeeTM-ER-ML they are not. In LeeTM-ML, performance drops significantly resulting in 0.26x slowdown (see Figure 5.10). Furthermore, due to spatial locality of the aborted transactions, the abort rate in LeeTM-ML is very high as shown in Table B.3 and in Figure B.1. On the contrary, in LeeTM-ER-ML, a maximum speedup of 2.45x is observed at 28 threads (see Figure 5.10). Furthermore, the abort rate in contrast to LeeTM-ML is significantly lower (Figure B.2, Table B.7).

Concerning KMeans, both KMeasnHigh-ML and KMeansLow-ML exhibit significant slowdowns of 0.27x and 0.31x maximum respectively (see Figure 5.10). Both benchmarks’ performances are highly influenced from the high abort rate of the benchmarks again due to the spatial locality of the aborted transactions (Tables B.27, B.23 and Figures B.3, B.4). In contrast to LeeTM-ML, both the readsets and the writesets of the conflicting transactions in KMeans are small (Tables 5.2, 5.3) and therefore the performance is not influenced from sending them to the master node as much as in LeeTM-ML.

Vacation-ML and Genome-ML also suffer from the same pathology as KMeans with maximum slowdowns of 0.28x and 0.32x respectively. The abort rates for both benchmarks are high as shown in Tables B.15, B.19 and in Figures B.5, B.6.

\(^2\)In the ML protocol, the transactions which failed to acquire a lease are aborted and immediately restart to acquire a lease again.
### Table 5.6: Benchmarks’ execution times - ML protocol

Finally, GLife-ML’s performance is not affected by the pathology mentioned earlier due to its low abort rate (Figure B.7, Table B.11). Therefore, it exhibits a maximum speedup of 2x at 28 threads (see Figure 5.10).
CHAPTER 5. EVALUATION

![Graph of KMeansHigh-ML Speedup vs Number of Threads]

![Graph of KMeansLow-ML Speedup vs Number of Threads]

![Graph of GLife-ML Speedup vs Number of Threads]
Figure 5.10: Benchmarks’ speedups - Multiple Leases protocol
CHAPTER 5. EVALUATION

Figure 5.11: Breakdown by transaction stage of the execution times. - ML protocol
5.4.1.4 Anaconda

Figure 5.12 depicts the speedups of the benchmarks when the decentralized Anaconda protocol is employed. Accordingly, Table 5.7 presents their execution times.

Concerning LeeTM, Anaconda is consistent with the previous protocols resulting in a maximum slowdown of 0.58x at 32 threads (see Figure 5.12) due to high abort rates as shown in Table B.4 and in Figure B.1. On the contrary, LeeTM-ER-ANA exhibits a speedup of 2.65x at 24 threads. Beyond 24 threads, performance drops reaching finally a 2.51x speedup at 32 threads. As in previous protocols the abort rate is low as illustrated in Table B.8 and in Figure B.2. Once again, results indicate that LeeTM-ER can benefit from distributed transactional execution no matter which protocol is employed.

Concerning KMeans, both KMeansHigh-ANA and KMeansLow-ANA configurations fail to achieve any speedup (Figure 5.12) due to the problems of high abort rate (Tables B.28, B.24 and Figures B.3, B.4) and spatial locality of the aborted transactions. GLife-ANA exhibits a steady performance no matter the number of threads, a slight speedup of 1.05x (Figure 5.12). As depicted in Figure 5.13 the percentage of time spent in execution is low compared to that spent for remote requests. Hence, the execution time is heavily dominated by the time spent in remote requests. Even the fact that in the Anaconda protocol transactions can commit in parallel, avoiding the bottlenecks in the master node of the centralized protocols, can not benefit execution. Furthermore, in the Anaconda protocol, potential local conflicts do not cause immediate aborts as in the centralized protocols. As already mentioned, the centralized protocols follow an eager-local validate approach which means that local conflicts are detected and resolved upon objects’ accesses. The Anaconda protocol, on the contrary, follows a lazy-local validate approach where objects are detected and resolved at the end of the transactions’ stages (during the commit phase). Transactions that attempt to read objects that are currently being modified by other transactions will block waiting for the opponent to commit. In GLife-ANA, local threads read/write consecutive cells. When centralized protocols are employed the number of aborts is increased while in the Anaconda protocol threads are blocked and hence the speedup is influenced (and remains steady).

Finally, both Genome-ANA and Vacation-ANA exhibit speedups of 1.65x (at 28 threads) and 1.42x (at 24 threads) respectively (Figure 5.12). Both benchmarks’ abort rates in Anaconda are significantly lower than in the centralized...
Table 5.7: Benchmarks’ execution times - Anaconda protocol

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeeTM</td>
<td>477.6</td>
<td>466.4</td>
<td>489</td>
<td>543.6</td>
<td>587</td>
<td>646.8</td>
<td>736.2</td>
<td>821.8</td>
</tr>
<tr>
<td>LeeTM-ER</td>
<td>153</td>
<td>95.4</td>
<td>73.8</td>
<td>66.6</td>
<td>61.4</td>
<td>57.8</td>
<td>59.2</td>
<td>62</td>
</tr>
<tr>
<td>KMeansHigh</td>
<td>558</td>
<td>615</td>
<td>83.0</td>
<td>1110</td>
<td>1248</td>
<td>1400</td>
<td>1530</td>
<td>1650</td>
</tr>
<tr>
<td>KMeansLow</td>
<td>385</td>
<td>558</td>
<td>690</td>
<td>780</td>
<td>920</td>
<td>1010</td>
<td>1230</td>
<td>1300</td>
</tr>
<tr>
<td>GLife</td>
<td>99.8</td>
<td>93</td>
<td>94</td>
<td>94.4</td>
<td>94.4</td>
<td>94.2</td>
<td>94</td>
<td>94</td>
</tr>
<tr>
<td>Vacation</td>
<td>75.6</td>
<td>62.8</td>
<td>52.8</td>
<td>56.8</td>
<td>54</td>
<td>53.2</td>
<td>57.2</td>
<td>58.8</td>
</tr>
<tr>
<td>Genome</td>
<td>149</td>
<td>114</td>
<td>100.2</td>
<td>97</td>
<td>93.8</td>
<td>92.4</td>
<td>90.2</td>
<td>91.2</td>
</tr>
</tbody>
</table>

protocols as shown in Figures B.20, B.16 and in Tables B.6, B.5.
CHAPTER 5. EVALUATION
CHAPTER 5. EVALUATION

KMeansHigh-ANA

Number of Threads

KMeansLow-ANA

Number of Threads

GLife-ANA

Number of Threads
Figure 5.12: Benchmarks' speedups - Anaconda protocol
Figure 5.13: Breakdown by transaction stage of the execution times. - ANA protocol
5.4.2 Execution Times

This section presents the execution times of benchmarks when run in all protocols as well as in their implemented lock-based implementations. Initially, the lock-based implementations of the benchmarks are explained (Section 5.4.2.1). In turn, their execution times are presented and a categorization of the benchmarks’ performances in relation to their characteristics and the protocol employed is illustrated.

5.4.2.1 Lock-based implementations

Lee  Lee’s benchmark has two lock-based implementations: coarse-grain and medium-grain. In the coarse-grain implementation, the whole grid is locked every time a thread accesses it. Thus, the execution is serialized on the grid and every thread waits until the current lock-owner of the grid releases the lock. In the medium-grain implementation, the grid is partitioned in blocks and a lock is assigned to each block. Every thread retrieves a job (track) to execute (lay track) from the main queue. Upon a job’s retrieval a bound check is performed in order to discover in which block the track is laid. In turn, the thread acquires the block’s lock and executes the job. If the track, in the process of being laid, strays into another block, it is aborted and the job is placed in a secondary job queue. When the first round of tracks to be laid (or attempted to be laid) finishes, the number of grid blocks is halved (making them more coarse grain) and new locks are assigned to the new blocks. The two job queues, the main and the secondary, are switched and the threads attempt to lay the remaining tracks. This iterative process continues until the grid blocks become one (a single lock guarding the grid, similar to coarse-grain locking) and all routes are eventually laid.

GLife  GLife lock-based implementations are identical to those of Lee benchmark. Again two modes exist, coarse-grain and medium-grain that function similarly to those of Lee benchmark.

KMeans, Vacation, Genome  KMeans, Vacation and Genome utilize other data structures than arrays (in contrast to Lee and GLife). The data structures utilized are: arrays, hashmaps and hashtables. Concerning the arrays, the coarse-grain approach is used (a single lock guarding the array). The hashmaps are inherently thread-safe and Terracotta’s underlying infrastructure automatically

provides distributed locking functionality. Finally, hashmaps are replaced with ConcurrentHashMaps from the java.util.concurrent package. Terracotta provides automated distributed locking functionality for ConcurrentHashMaps also.

5.4.2.2 Execution Times

As explained in the previous subsection, LeeTM, does not exhibit any speedup in any protocol used. In some cases, in addition, it exhibits slowdowns with the one derived from the utilization of the Multiple Leases (ML) protocol to be the most significant one (0.3x). Concerning the rest of the protocols (Anaconda, Single Lease (SL) and TCC), they outperform the lock based configurations (both coarse grain and medium grain) run in Terracotta. In particular, Anaconda is 3.54 times faster than Terracotta Coarse Grain Locking (TCGL) and 1.9 times faster than Terracotta Medium Grain Locking (TMGL). In turn, TCC is 2.82 times faster than TCGL and 1.5 times faster than TMGL. Finally, Single Lease (SL) is 2.89 times faster than TCGL and 1.57 times faster than TMGL. Concerning the transactional protocols, Anaconda is the winner outperforming TCC, SL and ML by 1.25x, 1.2x and 11.6x respectively.

In LeeTM-ER, Anaconda is the winner outperforming all other protocols. Concerning the Terracotta lock-based implementations, Anaconda is 17.6 and 9.48 times faster than TCGL and TMGL respectively. TCC, SL and ML protocols exhibit similar performance where both configurations of Terracotta are slower than all of them. In general, LeeTMs transactions are large (Tables 5.3, 5.2)
and Terracotta seems not to be able to provide memory coherency in such large workloads efficiently. On the contrary, due to the early-release optimization the transactional protocols can benefit from distributed execution with Anaconda exhibiting the best performance due to its non-blocking commit procedure.

KMeansHigh and KMeansLow exhibit similar behavior. In both benchmarks, Terracotta outperforms all transactional protocols. Due to high contention, the high number of aborts significantly slow down the execution of all transactional protocols. On the contrary, the lock based implementations, although they do not scale while increasing the number of threads, perform better resulting in lower execution times. More specifically, in KMeansHigh, Terracotta is 3.6, 3.96, 6.49
and 7.12 times faster than TCC, SL, ML and Anaconda respectively. Comparing the transactional protocols, Anaconda is the slowest one with TCC and SL being the best performing ones but again over 3 times slower than the lock-based implementation. Concerning KMeansLow, again Terracotta is 2.39, 3.29, 4.44 and 5.04 times faster than TCC, SL, ML and Anaconda respectively. Since the contention is lower than in KMeansHigh, the transactional protocols perform better but again they are slower than the Terracotta lock-based implementations.

In GLife, Anaconda, in average, outperforms all protocols. In particular, Anaconda is 1.68, 1.59, 3.27, 5.62 and 1.41 times faster than TCGL, TCML, TCC, SL and ML respectively. Both coarse-grain and medium-grain implementations
of Terracotta perform worse than the transactional implementation of Anaconda. However, the lock-based implementations outperform TCC and SL protocols.

In Vacation, the lock-based implementations outperform the transactional implementations (TCGL is 1.46, 3.53, 4.8 and 3.24 times faster than TCC, SL, ML and Anaconda respectively). Once more we can observe that the distributed fine-grain lock based implementations of Terracotta perform better than the transactional implementations. Concerning the transactional protocols, TCC performs better resulting in 2.41x, 3.28x and 2.21x speedups compares to SL, ML and Anaconda respectively.

Figure 5.19: Vacation execution times.

Figure 5.20: Genome execution times.
Finally, in Genome, the lock-based implementation outperforms the transactional ones similarly to Vacation. In particular, TCGL is 11, 20, 27 and 3.75 times faster than TCC, SL, ML and Anaconda respectively. Concerning the transactional protocols, Anaconda is the fastest by factors of 2.95, 5.38 and 7.2 in respect to TCC, SL and ML.
5.5 GC Impact

A significant factor affecting the performance of Java systems is the Garbage Collector. Commercial JVMs, most commonly used in the enterprise domain, face difficulties when utilizing tens of GBs of heap. In order to overcome this limitation, several JVM instances are deployed and incoming requests are distributed amongst them. Solutions to overcome this problem exist but they require changes in the whole stack of software used as well as in the underlying architecture [15].

Unfortunately, when incorporating STM into JVMs the garbage collection problem deteriorates due to the amount of meta-data being created. Furthermore, when moving to distributed JVM configurations the garbage collection problem becomes even worse due to explosion of meta-data and book-keeping information that have to be maintained. In addition, the RMI interface generates a significant amount of data due to the marshaling and serialization of objects.

A performance study of the GC impact on DiSTM is out of the scope of this thesis. However, an indicator can be drawn by profiling dstm2 (on which three of DiSTM’s configuration are based). The percentage of the time spent in GC cycles of dstm2 is depicted in Figure 5.21.

As shown in Figure 5.21, the time spent for GC in some benchmarks is over 40% (Vacation). In other cases, the percentage is smaller such as in LeeTM or in KMeans. The memory footprint of LeeTM and KMeans during execution stabilizes as those benchmarks do not generate dynamic data. Consequently, after some GC cycles when the vital data are promoted to the mature generations, the subsequent GC cycles concern only the young generations which are fast. However, special care has to be taken in order to ensure that the total memory footprint of the application will be able to fit in the initial heap in order to avoid the expensive resizing of the heap. Concerning Vacation, the situation differs significantly as it creates a significant amount of dynamic objects throughout its execution. Since Vacation is simulating a database, every insertion or deletion from the database either creates or deletes new/existing objects. This stretches the garbage collection as shown in Figure 5.21.

DiSTM’s memory footprint is significantly higher compared to dstm2 (between 6GB and 15GB). Hence, the GC’s impact can be detrimental to the performance of DiSTM hindering the potential performance benefits of the transactional protocols.
Figure 5.21: Percentage of time spent in GC.
5.6 Summary

As demonstrated in this chapter, the performance of protocols vary depending on the workloads and the level of contention. Tables 5.22, 5.23 categorize the best performing protocols according to transaction’s length and levels of contention. Table 5.22 includes only the transactional protocols while Table 5.23 includes also the lock-based implementations of Terracotta.

As illustrated in Table 5.22, in benchmarks with large transactions (LeeTM, LeeTM-ER) no matter the level of contention, Anaconda outperforms all transactional coherence protocols. Concerning LeeTM, as shown in Figure 5.14, Anaconda is marginally better than the other transactional protocols. As the number of threads increases, however, the performance of Anaconda deteriorates and becomes worse than SL and TCC. On average though, up to 32 threads, Anaconda is the best performing protocol. The reason for the deterioration of Anaconda’s performance is the increasing number of aborts as more threads are added. This leads to a bigger abort ratio than the other protocols (Figure B.1) and Anaconda’s strength, the parallel commit, can not compensate for that. Concerning LeeTM-ER, Anaconda outperforms by far all other protocols. The low-abort rate (Figure B.2) in combination with the parallel commit phase of Anaconda seems to suit this category of benchmarks. In benchmarks with small transactions, the situation differs significantly. In KMeans (both KMeansHigh and KMeansLow), TCC is the best performing protocol. Both KMeansHigh and KMeansLow have a high abort rate (Figures B.3, B.4). As already explained in Section 4.4, all protocols except Anaconda have a distinct two-stage validation phase. The first stage is the local validation phase while the second is the remote validation phase. In scenarios with high abort rates, it is desirable not to let doomed transactions communicate with remote transactions as it would be a waste of resources and also it would unnecessarily throttle the network. Having a distinct local validation phase helps in that direction, as it keeps doomed transactions (by aborting them)
within a node. Of course, due to spatial locality (aborted transactions immediately re-attempt to commit) the number of local conflicts will increase rapidly (Tables B.21-B.28). However, this side-effect does not deteriorate performance as much as in Anaconda where the validation-commit procedure is unified and hence doomed transactions almost always communicate through the network. In Genome, where transactions are small and the contention is medium, Anaconda outperforms all transactional protocols. In this case the parallel commit benefits execution in contrast to the commit-blocking protocols. Concerning the last category of benchmarks, small transactions with low contention, two cases are observed. In GLife, Anaconda outperforms all protocols whereas in Vacation TCC is the best performing one. The reason behind Anaconda’s performance in GLife is locality. As explained in Section 5.2.7, in GLife, the array (grid) is partitioned amongst the nodes of the cluster. In turn, each thread is assigned a portion of this array and they perform the rules of life on the grid cells of their portion. During commit time, threads do not have to travel through the network as most of the cells are not cached from remote nodes. Consequently, when updating the objects with the new values, transactions again most of the time will not have to broadcast as there will be a few cases where objects are cached from remote nodes. The exploitation of the locality benefits Anaconda in applications with small transactions because the overhead introduced due to its three-stage commit protocol is leveraged. On the contrary, in applications where locality can not be exploited, such as Vacation, Anaconda is not the winner. In Vacation, customers randomly query/insert/delete records from a database. The dynamic nature of the benchmark results in objects being cached from the distributed hashmap and hence, transactions have to go through the network upon commit time. The overhead imposed by Anaconda slows down execution significantly and parallel commit can not compensate for that.

3The only exception are the cells that reside at the boundaries of the partitions of the nodes.
Concerning the lock-based implementations, as shown in Figure 5.23, the end result differentiates in three benchmarks: Vacation, Genome and KMeans. The common characteristics of these benchmarks is the fact that they utilize small transactions. As explained before, the distributed transactional coherence protocols slow down execution as they lead to pathology due to spatial locality. The transactions’ execution times are increased due to remote validation and commit phases and conflicting transactions repeatedly abort as they can resume and reach their validation phase many times while the committing ones are still in the commit phase. Consequently, the abort rate increases along with execution time (KMeans). Furthermore, Terracotta provides support for fine grain locking structures of the java.util.concurrent package such as ConcurrentHashMaps. The utilization of such high performing structures of Terracotta overcomes both the transactional overheads and its pathologies and therefore result in better execution times. On the contrary, when custom locking has to be employed (e.g. arrays), the distributed transactional coherence protocols perform better. If it is difficult to employ a dead-lock free fine-grain locking scheme, and hence coarse-grain or medium-grain locking schemes have to be implemented, transactional execution can help as it both abstracts away the need for explicit synchronization and results in better performance (LeeTM).
Chapter 6

Conclusions & Future Work

Stagnation in the research and development of uni-processor architectures has led both industry and academia to the exploitation of the domain of Chip Multiprocessors (CMPs). Nowadays, the majority of the vendors ship multi-core chips for desktop computers as well as for servers. Concerning the domain of distributed computing, the single node clusters are being replaced by clusters of multi-cores. Consequently, the potential parallelism to be exploited is two-dimensional. Firstly, the parallelism that can be achieved within a single node of the cluster and secondly the parallelism of distributing tasks across a network of computers.

The introduction of multi-cores brings new incentives in software development. In order to achieve higher performance, software has to exploit the presence of multiple cores with multithreading. Until today, the development of parallel software has been the primary focus of the HPC domain. The shift towards multi-cores, though, would necessitate even from mainstream programmers to design, develop and deploy their applications in parallel environments.

The development of parallel software is a challenging and error-prone task. Data races which potentially can lead to fatal results, such as deadlocks, can be solved by explicit synchronization of concurrent accesses over shared data. Traditionally, this has been achieved by the use of mutual exclusion locks. The scalability of the produced software, hence, is determined by the amount of serialization introduced in the software. Coarse-grain locking which is easier to use results in bottlenecks while fine-grain locking which favors performance is difficult to employ and of high risk.

During the last decade there has been a major effort to establish new parallel
programming models which would minimize the risks of concurrent programming without compromising on performance. Transactional Memory (TM) is one of the results of this effort which draw significant attention. Transactional Memory, derived from database theory, promises to abstract away from the programmer the need of explicit synchronization. Instead of explicit locking, with TM, programmers enclose the critical segments of code into transactions. The memory transactions are executed by the underlying TM system complying to the ACI principles.

This thesis investigated the role Transactional Memory can play in the domain of distributed multi-core architectures. More specifically, this thesis focused on applying TM in the domain of clustered Java Virtual Machines (JVMs). Several transactional coherence protocols have been implemented and compared against each other as well as against lock-based implementations of applications deployed on commercial state-of-the-art clustering solutions. The following subsection lists in detail the contributions of this thesis.

### 6.1 Contributions

The contributions of this thesis are the following:

- In Chapter 4 the Distributed Software Transactional Memory (DiSTM) systems has been introduced. DiSTM is a JVM clustering solution with special support for transactional execution. Transactions are defined by annotating objects (which will be accessed transactionally) with the @dis-tatonic annotation. DiSTM, upon bootstrap, transparently clusters the applications by creating remote threads over the cluster and filling the task pools of each node for execution.

- In Section 4.3.3 the distributed atomic data structures employed by DiSTM have been introduced. A variety of atomic collection classes has been implemented in order to ease programming and to abstract away from the protocol developer the need for explicit development of transactional libraries. Currently, the collection classes supported by DiSTM are arrays, singleton objects, HashMaps and HashTables.

- Section 4.4 discussed the distributed transactional coherence protocols implemented. Four novel distributed transactional coherence protocols are
introduced. Three out of the four protocols (TCC, SL and ML) are centralized and their operation is dependent on the presence of a master node which coordinates execution and guarantees memory consistency. The centralized protocols employ a blocking commit policy in which transactions’ commits are serialized. The fourth protocol, Anaconda, is a pure decentralized distributed transactional coherence protocol in which transactions can commit fully in parallel.

The proposed distributed transactional protocols have been evaluated with established TM oriented benchmarks ported both to DiSTM and to their correspondent lock-based versions run under the state-of-the-art Terracotta JVM clustering solution. Depending on the nature of the benchmarks different protocols performed better. Generally, as explained in the evaluation section (Chapter 5), Anaconda performs better as it allows parallel commits of transactions. The exception to the rule are low-contention benchmarks with small transactions. In this case, the overhead of Anaconda is high and the parallel commit can not compensate. Concerning the lock-based implementations, the transactional protocols outperform coarse-grain or medium-grain locking schemes. However, fine-grain highly-tuned concurrent data structures (such as the ones employed by Terracotta) outperform their corresponding transactional versions.

6.2 Future Work

This thesis has investigated a small portion of the design space in the distributed transactional memory execution. There is a significant number of areas for future investigation. Those areas concern:

- **Infrastructure Optimizations**: DiSTM is a clustering JVM solution written in Java. The remote communication system is based on the ProActive framework which is a high level API wrapper of Java RMI. Concerning ProActive, there is a variety of optimizations that can be exploited such as Group Communications and Immediate Services. With group communications a stream is serialized once and sent to a group of nodes limiting serialization costs. DiSTM currently uses group communications but there is still room for further exploitation. With immediate services, active objects can serve a request immediately bypassing the wait queue. Although
this might introduce consistency risks, if carefully used, it can speedup execution. DiSTM currently does not use immediate services as its primary focus was consistency. Concerning RMI which is inherently slow due to object marshaling and serialization, a variety of solutions exists which can boost performance significantly. For example, Java Fast Sockets (JFS) [95] bypass object serialization by directly mapping memory locations to the network interface. Furthermore, a more aggressive optimization would be the total replacement of Java’s network communication with MPI. In this way the modularity and flexibility of the programming interface of DiSTM would be maintained while increasing performance. ProActive currently supports wrapping of MPI code. In addition, moving to MPI could lead to network interconnects with higher bandwidth such as Myrinet or InfiniBand as it can support them.

- **Tuning:** Tuning DiSTM can be achieved in many ways. The most important tuning aspect is the Garbage Collector (GC). GC can lead to significant bottlenecks to Java execution. Especially in multi-threaded environments, the stop-the-world stages of the GC, can negatively affect performance. Currently DiSTM does not support any form of distributed GC. The GC process is performed per node with the standard HotSpot’s GCs. Extra care has been taken in order to avoid collection of vital transactional metadata. This resulted in DiSTM’s significant memory footprint (between 6GB and 15GB) which consequently led to increased stall times due to GCs. Another important tuning parameter is the scheduling of the transactions. As shown, high abort rates in a distributed environment are fatal for performance. Techniques tackling pathologies of transactional memory (for example repeat conflicts, Chapter 5) exist [10, 17] and their application is orthogonal to DiSTM.

- **Transactional Protocols:** Concerning optimistic distributed transactional protocols, which was the main focus of this thesis, there is still a vast design space for exploration. objects’ multiversioning, time-stamp ordering, contention managers and resolution policies are some of them.

- **Applications:** The applicability of systems and protocols such as the one proposed in this thesis can take place in the enterprise domain. Terracotta’s main target is the clustering of enterprise applications with minimal
alterations of the existing code. In addition, existing enterprise Java-based servers such as JBoss, WebSphere and GlassFish provide clustering of their instances in order to speed-up execution. Consequently, applying transactional protocols to existing products or porting existing enterprise applications to DiSTM could provide a better insight of the key problems hurting performance as it would allow the evaluation against real-life workloads.

Overall, the application of TM to distributed systems appears to be a promising solution especially for those applications that are difficult to parallelize with lock based solutions. Although the distributed TM protocols, presented in this thesis, do not cope well with certain workloads, the applicability of the optimizations described may enhance performance significantly. Furthermore, enterprise applications usually have loosely coupled datasets and therefore it is uncommon for highly contented workloads to exist. However, it is vital for any distributed TM system to be competitive with lock based systems in any workload. This might be impossible for certain cases as research in TM for ChipMultiprocessors has demonstrated until now. The ease in programmability of TM along with the continuous performance improvement may close that gap and finally TM will prevail against locks in multithreaded programming, distributed or not.
Bibliography


[27] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. HPJava: Data parallel extensions to Java. 1998.


[48] Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon. Robust Contention Management in Software Transactional Memory. In SCOOL ’05: Workshop on Synchronization and Concurrency in Object-Oriented Languages, 2005.


<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[60] IBM. Websphere application server.</td>
</tr>
</tbody>
</table>


Appendix A

Porting Benchmarks to DiSTM

Listing A.1 depicts the complete source code of the multi-threaded shared counter example. Each of the constructed thread randomly decides to increment or decrement the shared counter. Furthermore, the access to the counter is achieved through lock acquisition in order to achieve memory consistency. In order to port the above program in DiSTM a number of modifications have to be performed as described in Chapter 3. First, the interface of the atomic integer counter has to be defined, Listing A.2.
public class SharedCounter implements Runnable {

    private static int counter;
    private static final Object lock = new Object();
    private static final Random random = new Random();

    public void run() {
        int stopCounter = 0;
        while (stopCounter < 500) {
            stopCounter++;
            synchronized (lock) {
                if (random.nextBoolean()) {
                    counter++;
                } else {
                    counter--;
                }
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Thread[] threads = new Thread[16];
        for (int i = 0; i < 16; i++) {
            threads[i] = new Thread(new SharedCounter());
        }

        for (Thread thread : threads) {
            thread.start();
        }

        for (Thread thread : threads) {
            thread.join();
        }
    }
}

Listing A.1: Multi-threaded Shared Counter example
@distatomic

```java
public interface SharedCounter {
    public int getValue();
    public void setValue(int value);
}
```

Listing A.2: Shared Counter Atomic Integer

Furthermore, the “transactional job” that describes the increment or the decrement of the shared counter has to be implemented as in Listing A.3. The `SharedCounterJob` class extends DiSTM’s `TransactionalJob` class and defines the type of the job (in this example an enumeration has been used). In addition, the main class of the shared counter program has to provide an implementation for the `createTxJobs` method which constructs the initial number of jobs to be distributed.

```java
public class SharedCounterJob extends TransactionalJob {

    public static enum Type {
        INCREMENT, DECREMENT
    }
    private Type type;
    private static Random random = new Random();

    public SharedCounterJob() {
        if (random.nextBoolean()) {
            type = Type.INCREMENT;
        } else {
            type = Type.DECREMENT;
        }
    }

    public Type getType() {
        return type;
    }
}
```

Listing A.3: Shared Counter Transactional Job

Finally, the user has to provide an implementation for the `SharedCounterThread` class, Listing A.4. The singleton `DistAtomicObject`, from DiSTM’s collection
classes, is constructed and the type of the `@distatomic` object that the distributed TM semantics have to be applied are passed (in this case the `SharedCounter` type).

```java
public class SharedCounterThread extends DiSTMThread {

    public static final DistAtomicObject<SharedCounter> counter
        = new DistAtomicObject<SharedCounter>();

    public SharedCounterThread() {
    }

    @Override
    public void run() {
        TransactionalJob txJob;
        while ((txJob = getTxJob()) != null) {
            DiSTMThread.doIt(new Callable<Void>() {
                @Override
                public Void call() {
                    SharedCounterJob job = (SharedCounter) txJob;
                    if (job.getType() == Type.INCREMENT) {
                        counter.setValue(counter.getValue() + 1);
                    } else {
                        counter.setValue(counter.getValue() - 1);
                    }
                }
            });
        }
    }
}
```

Listing A.4: Shared Counter DiSTMThread

The `SharedCounterThread` threads attempt to acquire jobs from their local work queues (via the `getTxJob()` method) and execute them. In this example, the type of the job is examined and the correspondent action (increment or decrement the counter) is performed. The code enclosed within the `DiSTMThread.doIt` method is executed transactionally and the iterations are completed until there are no remaining jobs in the queues.
Appendix B

Results

The following subsections contain various statistics gathered from DiSTM’s execution. Subsection B.1 entails the absolute number of aborts per protocol per benchmark. The number of total aborts is split down to the number of aborts per protocol stage. Subsection B.2 presents the APC figures of the benchmarks.

APC (Aborts per Commit) is a statistic that represents the ratio of aborts per commit for each configuration. This metric shows diagrammatically the level of contention of the configurations.

B.1 Commits & Aborts

B.1.1 LeeTM

Tables B.1 to B.28 show the number of aborts per benchmark for each protocol. The number of total aborts is split down to the number of aborts per protocol stages.

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Remote Validation Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>1479</td>
<td>1477</td>
<td>0</td>
<td>1423</td>
</tr>
<tr>
<td>8 threads</td>
<td>1478</td>
<td>3733</td>
<td>1897</td>
<td>1775</td>
</tr>
<tr>
<td>12 threads</td>
<td>1478</td>
<td>5450</td>
<td>3238</td>
<td>2117</td>
</tr>
<tr>
<td>16 threads</td>
<td>1475</td>
<td>7820</td>
<td>4845</td>
<td>2868</td>
</tr>
<tr>
<td>20 threads</td>
<td>1476</td>
<td>9924</td>
<td>6310</td>
<td>3474</td>
</tr>
<tr>
<td>24 threads</td>
<td>1473</td>
<td>12490</td>
<td>8048</td>
<td>4285</td>
</tr>
<tr>
<td>28 threads</td>
<td>1475</td>
<td>15122</td>
<td>9906</td>
<td>5036</td>
</tr>
<tr>
<td>32 threads</td>
<td>1475</td>
<td>17615</td>
<td>11398</td>
<td>5841</td>
</tr>
</tbody>
</table>

Table B.1: LeeTM Commits/Aborts - TCC protocol
### APPENDIX B. RESULTS

<table>
<thead>
<tr>
<th>Threads</th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1478</td>
<td>785</td>
<td>0</td>
<td>47</td>
<td>738</td>
</tr>
<tr>
<td>8</td>
<td>1477</td>
<td>1722</td>
<td>131</td>
<td>121</td>
<td>1470</td>
</tr>
<tr>
<td>12</td>
<td>1477</td>
<td>2586</td>
<td>341</td>
<td>177</td>
<td>2668</td>
</tr>
<tr>
<td>16</td>
<td>1477</td>
<td>3280</td>
<td>584</td>
<td>205</td>
<td>2491</td>
</tr>
<tr>
<td>20</td>
<td>1471</td>
<td>4281</td>
<td>806</td>
<td>244</td>
<td>3131</td>
</tr>
<tr>
<td>24</td>
<td>1473</td>
<td>5026</td>
<td>1069</td>
<td>286</td>
<td>3671</td>
</tr>
<tr>
<td>28</td>
<td>1470</td>
<td>5803</td>
<td>1281</td>
<td>347</td>
<td>4175</td>
</tr>
<tr>
<td>32</td>
<td>1467</td>
<td>7043</td>
<td>1655</td>
<td>585</td>
<td>4803</td>
</tr>
</tbody>
</table>

Table B.2: LeeTM Commits/Aborts - SL protocol

<table>
<thead>
<tr>
<th>Threads</th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1475</td>
<td>1456</td>
<td>0</td>
<td>646</td>
<td>810</td>
</tr>
<tr>
<td>8</td>
<td>1471</td>
<td>3693</td>
<td>1095</td>
<td>1297</td>
<td>1301</td>
</tr>
<tr>
<td>12</td>
<td>1480</td>
<td>5990</td>
<td>2323</td>
<td>2053</td>
<td>1619</td>
</tr>
<tr>
<td>16</td>
<td>1464</td>
<td>8995</td>
<td>4072</td>
<td>2904</td>
<td>2019</td>
</tr>
<tr>
<td>20</td>
<td>1472</td>
<td>11968</td>
<td>5745</td>
<td>3674</td>
<td>2279</td>
</tr>
<tr>
<td>24</td>
<td>1463</td>
<td>14391</td>
<td>7212</td>
<td>4457</td>
<td>2722</td>
</tr>
<tr>
<td>28</td>
<td>1470</td>
<td>18109</td>
<td>9942</td>
<td>5241</td>
<td>2926</td>
</tr>
<tr>
<td>32</td>
<td>1476</td>
<td>23614</td>
<td>12507</td>
<td>7798</td>
<td>3309</td>
</tr>
</tbody>
</table>

Table B.3: LeeTM Commits/Aborts - ML protocol

<table>
<thead>
<tr>
<th>Threads</th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Lock Acquisition Aborts</th>
<th>Revoked Lock Aborts</th>
<th>Broadcast Aborts</th>
<th>Update Objects Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1479</td>
<td>1533</td>
<td>4</td>
<td>2</td>
<td>1513</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>1473</td>
<td>4357</td>
<td>14</td>
<td>4</td>
<td>4273</td>
<td>64</td>
</tr>
<tr>
<td>12</td>
<td>1476</td>
<td>7834</td>
<td>23</td>
<td>4</td>
<td>7706</td>
<td>101</td>
</tr>
<tr>
<td>16</td>
<td>1475</td>
<td>11439</td>
<td>33</td>
<td>7</td>
<td>11270</td>
<td>129</td>
</tr>
<tr>
<td>20</td>
<td>1475</td>
<td>16153</td>
<td>49</td>
<td>9</td>
<td>15932</td>
<td>163</td>
</tr>
<tr>
<td>24</td>
<td>1476</td>
<td>21296</td>
<td>88</td>
<td>18</td>
<td>20994</td>
<td>196</td>
</tr>
<tr>
<td>28</td>
<td>1474</td>
<td>29113</td>
<td>147</td>
<td>25</td>
<td>28705</td>
<td>236</td>
</tr>
<tr>
<td>32</td>
<td>1472</td>
<td>33068</td>
<td>265</td>
<td>43</td>
<td>32365</td>
<td>395</td>
</tr>
</tbody>
</table>

Table B.4: LeeTM Commits/Aborts - Anaconda protocol
## B.1.2 LeeTM-ER

<table>
<thead>
<tr>
<th></th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Remote Validation Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4 threads</strong></td>
<td>1469</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td><strong>8 threads</strong></td>
<td>1454</td>
<td>141</td>
<td>141</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>12 threads</strong></td>
<td>1440</td>
<td>270</td>
<td>269</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>16 threads</strong></td>
<td>1444</td>
<td>419</td>
<td>417</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>20 threads</strong></td>
<td>1441</td>
<td>531</td>
<td>529</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>24 threads</strong></td>
<td>1427</td>
<td>668</td>
<td>668</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td><strong>28 threads</strong></td>
<td>1418</td>
<td>720</td>
<td>720</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>32 threads</strong></td>
<td>1419</td>
<td>890</td>
<td>890</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

Table B.5: LeeTM-ER Commits/Aborts - TCC protocol

<table>
<thead>
<tr>
<th></th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4 threads</strong></td>
<td>1477</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td><strong>8 threads</strong></td>
<td>1477</td>
<td>90</td>
<td>88</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td><strong>12 threads</strong></td>
<td>1477</td>
<td>133</td>
<td>104</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td><strong>16 threads</strong></td>
<td>1477</td>
<td>170</td>
<td>122</td>
<td>30</td>
<td>18</td>
</tr>
<tr>
<td><strong>20 threads</strong></td>
<td>1471</td>
<td>217</td>
<td>146</td>
<td>45</td>
<td>20</td>
</tr>
<tr>
<td><strong>24 threads</strong></td>
<td>1473</td>
<td>262</td>
<td>180</td>
<td>60</td>
<td>24</td>
</tr>
<tr>
<td><strong>28 threads</strong></td>
<td>1470</td>
<td>352</td>
<td>230</td>
<td>72</td>
<td>30</td>
</tr>
<tr>
<td><strong>32 threads</strong></td>
<td>1467</td>
<td>445</td>
<td>320</td>
<td>90</td>
<td>35</td>
</tr>
</tbody>
</table>

Table B.6: LeeTM-ER Commits/Aborts - SL protocol

<table>
<thead>
<tr>
<th></th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4 threads</strong></td>
<td>1460</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td><strong>8 threads</strong></td>
<td>1465</td>
<td>92</td>
<td>45</td>
<td>41</td>
<td>6</td>
</tr>
<tr>
<td><strong>12 threads</strong></td>
<td>1460</td>
<td>178</td>
<td>88</td>
<td>80</td>
<td>10</td>
</tr>
<tr>
<td><strong>16 threads</strong></td>
<td>1477</td>
<td>278</td>
<td>131</td>
<td>132</td>
<td>15</td>
</tr>
<tr>
<td><strong>20 threads</strong></td>
<td>1460</td>
<td>363</td>
<td>170</td>
<td>172</td>
<td>21</td>
</tr>
<tr>
<td><strong>24 threads</strong></td>
<td>1475</td>
<td>482</td>
<td>239</td>
<td>292</td>
<td>41</td>
</tr>
<tr>
<td><strong>28 threads</strong></td>
<td>1477</td>
<td>566</td>
<td>251</td>
<td>250</td>
<td>65</td>
</tr>
<tr>
<td><strong>32 threads</strong></td>
<td>1465</td>
<td>692</td>
<td>319</td>
<td>301</td>
<td>72</td>
</tr>
</tbody>
</table>

Table B.7: LeeTM-ER Commits/Aborts - ML protocol
### APPENDIX B. RESULTS

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Lock Acquisition Aborts</th>
<th>Revoked Lock Aborts</th>
<th>Broadcast Aborts</th>
<th>Update Objects Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>1479</td>
<td>17</td>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8 threads</td>
<td>1470</td>
<td>45</td>
<td>31</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>12 threads</td>
<td>1472</td>
<td>74</td>
<td>43</td>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>16 threads</td>
<td>1477</td>
<td>97</td>
<td>42</td>
<td>9</td>
<td>35</td>
</tr>
<tr>
<td>20 threads</td>
<td>1469</td>
<td>140</td>
<td>54</td>
<td>20</td>
<td>51</td>
</tr>
<tr>
<td>24 threads</td>
<td>1472</td>
<td>158</td>
<td>50</td>
<td>25</td>
<td>65</td>
</tr>
<tr>
<td>28 threads</td>
<td>1471</td>
<td>202</td>
<td>64</td>
<td>32</td>
<td>81</td>
</tr>
<tr>
<td>32 threads</td>
<td>1479</td>
<td>250</td>
<td>85</td>
<td>40</td>
<td>95</td>
</tr>
</tbody>
</table>

Table B.8: LeeTM-ER Commits/Aborts - Anaconda protocol

#### B.1.3 GLife

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Remote Validation Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>100000</td>
<td>35</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>8 threads</td>
<td>100000</td>
<td>50</td>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>12 threads</td>
<td>100000</td>
<td>99</td>
<td>54</td>
<td>40</td>
</tr>
<tr>
<td>16 threads</td>
<td>100000</td>
<td>109</td>
<td>63</td>
<td>42</td>
</tr>
<tr>
<td>20 threads</td>
<td>100000</td>
<td>211</td>
<td>110</td>
<td>93</td>
</tr>
<tr>
<td>24 threads</td>
<td>100000</td>
<td>259</td>
<td>198</td>
<td>58</td>
</tr>
<tr>
<td>28 threads</td>
<td>100000</td>
<td>349</td>
<td>141</td>
<td>198</td>
</tr>
<tr>
<td>32 threads</td>
<td>100000</td>
<td>287</td>
<td>161</td>
<td>120</td>
</tr>
</tbody>
</table>

Table B.9: GLife Commits/Aborts - TCC protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>100000</td>
<td>30</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>8 threads</td>
<td>100000</td>
<td>67</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>12 threads</td>
<td>100000</td>
<td>397</td>
<td>296</td>
<td>50</td>
</tr>
<tr>
<td>16 threads</td>
<td>100000</td>
<td>477</td>
<td>392</td>
<td>43</td>
</tr>
<tr>
<td>20 threads</td>
<td>100000</td>
<td>989</td>
<td>819</td>
<td>90</td>
</tr>
<tr>
<td>24 threads</td>
<td>100000</td>
<td>2093</td>
<td>1387</td>
<td>94</td>
</tr>
<tr>
<td>28 threads</td>
<td>100000</td>
<td>2365</td>
<td>1796</td>
<td>104</td>
</tr>
<tr>
<td>32 threads</td>
<td>100000</td>
<td>2566</td>
<td>2566</td>
<td>80</td>
</tr>
</tbody>
</table>

Table B.10: GLife Commits/Aborts - SL protocol
### APPENDIX B. RESULTS

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads 100000</td>
<td>32256</td>
<td>0</td>
<td>32254</td>
<td>2</td>
</tr>
<tr>
<td>8 threads 100000</td>
<td>39478</td>
<td>10</td>
<td>39466</td>
<td>2</td>
</tr>
<tr>
<td>12 threads 100000</td>
<td>41857</td>
<td>77</td>
<td>41840</td>
<td>5</td>
</tr>
<tr>
<td>16 threads 100000</td>
<td>43836</td>
<td>114</td>
<td>43722</td>
<td>7</td>
</tr>
<tr>
<td>20 threads 100000</td>
<td>46002</td>
<td>198</td>
<td>45798</td>
<td>6</td>
</tr>
<tr>
<td>24 threads 100000</td>
<td>45009</td>
<td>280</td>
<td>44722</td>
<td>7</td>
</tr>
<tr>
<td>28 threads 100000</td>
<td>46291</td>
<td>343</td>
<td>45938</td>
<td>10</td>
</tr>
<tr>
<td>32 threads 100000</td>
<td>46395</td>
<td>438</td>
<td>46497</td>
<td>7</td>
</tr>
</tbody>
</table>

Table B.11: GLife Commits/Aborts - ML protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Lock Acquisition Aborts</th>
<th>Revoked Lock Aborts</th>
<th>Broadcast Aborts</th>
<th>Update Objects Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads 100000</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8 threads 100000</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>12 threads 100000</td>
<td>14</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>16 threads 100000</td>
<td>23</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>20 threads 100000</td>
<td>31</td>
<td>7</td>
<td>6</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>24 threads 100000</td>
<td>33</td>
<td>9</td>
<td>5</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>28 threads 100000</td>
<td>39</td>
<td>8</td>
<td>7</td>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>32 threads 100000</td>
<td>47</td>
<td>10</td>
<td>8</td>
<td>23</td>
<td>6</td>
</tr>
</tbody>
</table>

Table B.12: GLife Commits/Aborts - Anaconda protocol

### B.1.4 Vacation

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Remote Validation Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads 3930</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8 threads 3926</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>12 threads 3920</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>16 threads 3929</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>20 threads 3946</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>24 threads 3927</td>
<td>18</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>28 threads 3929</td>
<td>22</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>32 threads 3904</td>
<td>25</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

Table B.13: Vacation Commits/Aborts - TCC protocol

### B.1.5 Genome
### Table B.14: Vacation Commits/Aborts - SL protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>3931</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8 threads</td>
<td>3926</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12 threads</td>
<td>3916</td>
<td>18</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>16 threads</td>
<td>3929</td>
<td>72</td>
<td>61</td>
<td>11</td>
</tr>
<tr>
<td>20 threads</td>
<td>3946</td>
<td>180</td>
<td>150</td>
<td>15</td>
</tr>
<tr>
<td>24 threads</td>
<td>3927</td>
<td>229</td>
<td>200</td>
<td>14</td>
</tr>
<tr>
<td>28 threads</td>
<td>3929</td>
<td>411</td>
<td>370</td>
<td>21</td>
</tr>
<tr>
<td>32 threads</td>
<td>3925</td>
<td>735</td>
<td>680</td>
<td>25</td>
</tr>
</tbody>
</table>

### Table B.15: Vacation Commits/Aborts - ML protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>3929</td>
<td>11208</td>
<td>0</td>
<td>11207</td>
</tr>
<tr>
<td>8 threads</td>
<td>3925</td>
<td>26234</td>
<td>2</td>
<td>26229</td>
</tr>
<tr>
<td>12 threads</td>
<td>3920</td>
<td>40890</td>
<td>7</td>
<td>40889</td>
</tr>
<tr>
<td>16 threads</td>
<td>3928</td>
<td>53790</td>
<td>20</td>
<td>53774</td>
</tr>
<tr>
<td>20 threads</td>
<td>3948</td>
<td>70059</td>
<td>40</td>
<td>70013</td>
</tr>
<tr>
<td>24 threads</td>
<td>3927</td>
<td>84126</td>
<td>45</td>
<td>84110</td>
</tr>
<tr>
<td>28 threads</td>
<td>3928</td>
<td>98822</td>
<td>65</td>
<td>98757</td>
</tr>
<tr>
<td>32 threads</td>
<td>3903</td>
<td>112436</td>
<td>70</td>
<td>112366</td>
</tr>
</tbody>
</table>

### Table B.16: Vacation Commits/Aborts - Anaconda protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Remote Validation Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>47074</td>
<td>5141</td>
<td>0</td>
<td>4425</td>
</tr>
<tr>
<td>8 threads</td>
<td>47072</td>
<td>5533</td>
<td>75</td>
<td>4592</td>
</tr>
<tr>
<td>12 threads</td>
<td>47071</td>
<td>5993</td>
<td>463</td>
<td>5006</td>
</tr>
<tr>
<td>16 threads</td>
<td>47071</td>
<td>6176</td>
<td>590</td>
<td>5214</td>
</tr>
<tr>
<td>20 threads</td>
<td>47071</td>
<td>6304</td>
<td>712</td>
<td>5213</td>
</tr>
<tr>
<td>24 threads</td>
<td>47069</td>
<td>6527</td>
<td>853</td>
<td>5314</td>
</tr>
<tr>
<td>28 threads</td>
<td>47070</td>
<td>6710</td>
<td>900</td>
<td>5420</td>
</tr>
<tr>
<td>32 threads</td>
<td>47069</td>
<td>6809</td>
<td>910</td>
<td>5480</td>
</tr>
</tbody>
</table>

### Table B.17: Genome Commits/Aborts - TCC protocol
### APPENDIX B. RESULTS

#### Table B.18: Genome Commits/Aborts - SL protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>47069</td>
<td>4793</td>
<td>0</td>
<td>4241</td>
</tr>
<tr>
<td>8 threads</td>
<td>47070</td>
<td>26043</td>
<td>20239</td>
<td>5289</td>
</tr>
<tr>
<td>12 threads</td>
<td>47072</td>
<td>81155</td>
<td>75193</td>
<td>5436</td>
</tr>
<tr>
<td>16 threads</td>
<td>47072</td>
<td>154388</td>
<td>143888</td>
<td>5419</td>
</tr>
<tr>
<td>20 threads</td>
<td>47065</td>
<td>166086</td>
<td>160005</td>
<td>5526</td>
</tr>
<tr>
<td>24 threads</td>
<td>47073</td>
<td>188518</td>
<td>182231</td>
<td>5634</td>
</tr>
<tr>
<td>28 threads</td>
<td>47069</td>
<td>188876</td>
<td>187760</td>
<td>5741</td>
</tr>
<tr>
<td>32 threads</td>
<td>47070</td>
<td>197517</td>
<td>191210</td>
<td>5801</td>
</tr>
</tbody>
</table>

#### Table B.19: Genome Commits/Aborts - ML protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>47072</td>
<td>119767</td>
<td>0</td>
<td>119322</td>
</tr>
<tr>
<td>8 threads</td>
<td>47071</td>
<td>289487</td>
<td>176</td>
<td>289174</td>
</tr>
<tr>
<td>12 threads</td>
<td>47070</td>
<td>445389</td>
<td>1625</td>
<td>443700</td>
</tr>
<tr>
<td>16 threads</td>
<td>47072</td>
<td>596528</td>
<td>2261</td>
<td>594214</td>
</tr>
<tr>
<td>20 threads</td>
<td>47071</td>
<td>746425</td>
<td>2390</td>
<td>744075</td>
</tr>
<tr>
<td>24 threads</td>
<td>47070</td>
<td>884585</td>
<td>2312</td>
<td>882233</td>
</tr>
<tr>
<td>28 threads</td>
<td>47071</td>
<td>1030342</td>
<td>2330</td>
<td>1027972</td>
</tr>
<tr>
<td>32 threads</td>
<td>47072</td>
<td>1139246</td>
<td>2340</td>
<td>1136861</td>
</tr>
</tbody>
</table>

#### Table B.20: Genome Commits/Aborts - Anaconda protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Lock Acquisition Aborts</th>
<th>Revoked Lock Aborts</th>
<th>Broadcast Aborts</th>
<th>Update Objects Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>47050</td>
<td>77</td>
<td>34</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>8 threads</td>
<td>47071</td>
<td>204</td>
<td>82</td>
<td>8</td>
<td>45</td>
</tr>
<tr>
<td>12 threads</td>
<td>47075</td>
<td>220</td>
<td>91</td>
<td>4</td>
<td>65</td>
</tr>
<tr>
<td>16 threads</td>
<td>47069</td>
<td>294</td>
<td>119</td>
<td>8</td>
<td>84</td>
</tr>
<tr>
<td>20 threads</td>
<td>47092</td>
<td>313</td>
<td>127</td>
<td>6</td>
<td>86</td>
</tr>
<tr>
<td>24 threads</td>
<td>47081</td>
<td>337</td>
<td>134</td>
<td>8</td>
<td>95</td>
</tr>
<tr>
<td>28 threads</td>
<td>47085</td>
<td>370</td>
<td>161</td>
<td>11</td>
<td>89</td>
</tr>
<tr>
<td>32 threads</td>
<td>47086</td>
<td>388</td>
<td>170</td>
<td>13</td>
<td>95</td>
</tr>
</tbody>
</table>
### B.1.6 KMeansLow

<table>
<thead>
<tr>
<th>TCC protocol</th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Remote Validation Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads 40032</td>
<td>39311</td>
<td>0</td>
<td>36906</td>
<td>2405</td>
<td></td>
</tr>
<tr>
<td>8 threads 40032</td>
<td>78893</td>
<td>12049</td>
<td>57226</td>
<td>9798</td>
<td></td>
</tr>
<tr>
<td>12 threads 40032</td>
<td>114853</td>
<td>29571</td>
<td>72831</td>
<td>12451</td>
<td></td>
</tr>
<tr>
<td>16 threads 40032</td>
<td>157361</td>
<td>51365</td>
<td>91012</td>
<td>14654</td>
<td></td>
</tr>
<tr>
<td>20 threads 40032</td>
<td>175593</td>
<td>65448</td>
<td>96194</td>
<td>14741</td>
<td></td>
</tr>
<tr>
<td>24 threads 40032</td>
<td>194224</td>
<td>78106</td>
<td>100978</td>
<td>15140</td>
<td></td>
</tr>
<tr>
<td>28 threads 40032</td>
<td>224079</td>
<td>107472</td>
<td>119220</td>
<td>17387</td>
<td></td>
</tr>
<tr>
<td>32 threads 40032</td>
<td>252597</td>
<td>113623</td>
<td>121433</td>
<td>17541</td>
<td></td>
</tr>
</tbody>
</table>

**Table B.21:** KMeansLow Commits/Aborts - TCC protocol

<table>
<thead>
<tr>
<th>SL protocol</th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads 20016</td>
<td>20304</td>
<td>0</td>
<td>1003</td>
<td>10301</td>
<td></td>
</tr>
<tr>
<td>8 threads 20016</td>
<td>74190</td>
<td>20041</td>
<td>14400</td>
<td>30749</td>
<td></td>
</tr>
<tr>
<td>12 threads 20016</td>
<td>107618</td>
<td>52123</td>
<td>19910</td>
<td>35585</td>
<td></td>
</tr>
<tr>
<td>16 threads 20016</td>
<td>217878</td>
<td>140840</td>
<td>28740</td>
<td>48298</td>
<td></td>
</tr>
<tr>
<td>20 threads 20016</td>
<td>303130</td>
<td>212664</td>
<td>39610</td>
<td>50856</td>
<td></td>
</tr>
<tr>
<td>24 threads 20016</td>
<td>376720</td>
<td>274299</td>
<td>50010</td>
<td>52411</td>
<td></td>
</tr>
<tr>
<td>28 threads 20016</td>
<td>451196</td>
<td>326328</td>
<td>70130</td>
<td>54688</td>
<td></td>
</tr>
<tr>
<td>32 threads 20016</td>
<td>518106</td>
<td>382639</td>
<td>80190</td>
<td>55337</td>
<td></td>
</tr>
</tbody>
</table>

**Table B.22:** KMeansLow Commits/Aborts - SL protocol

<table>
<thead>
<tr>
<th>ML protocol</th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation Aborts</th>
<th>Lease Acquisition Aborts</th>
<th>Update Cache Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads 40032</td>
<td>104887</td>
<td>0</td>
<td>101084</td>
<td>3803</td>
<td></td>
</tr>
<tr>
<td>8 threads 40032</td>
<td>101290</td>
<td>18012</td>
<td>70395</td>
<td>12883</td>
<td></td>
</tr>
<tr>
<td>12 threads 40032</td>
<td>355108</td>
<td>40462</td>
<td>306052</td>
<td>11144</td>
<td></td>
</tr>
<tr>
<td>16 threads 40032</td>
<td>490568</td>
<td>69449</td>
<td>410012</td>
<td>11159</td>
<td></td>
</tr>
<tr>
<td>20 threads 40032</td>
<td>623331</td>
<td>98479</td>
<td>513496</td>
<td>11156</td>
<td></td>
</tr>
<tr>
<td>24 threads 40032</td>
<td>756575</td>
<td>128314</td>
<td>617054</td>
<td>11207</td>
<td></td>
</tr>
<tr>
<td>28 threads 40032</td>
<td>888703</td>
<td>159252</td>
<td>718719</td>
<td>10732</td>
<td></td>
</tr>
<tr>
<td>32 threads 40032</td>
<td>1021401</td>
<td>187631</td>
<td>823111</td>
<td>10669</td>
<td></td>
</tr>
</tbody>
</table>

**Table B.23:** KMeansLow Commits/Aborts - ML protocol
### APPENDIX B. RESULTS

#### Table B.24: KMeansLow Commits/Aborts - Anaconda protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Aborts</th>
<th>Aborts</th>
<th>Aborts</th>
<th>Aborts</th>
<th>Aborts</th>
<th>Aborts</th>
<th>Aborts</th>
<th>Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>35584</td>
<td>15383</td>
<td>5233</td>
<td>0</td>
<td>43</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td>35584</td>
<td>86310</td>
<td>43616</td>
<td>3</td>
<td>881</td>
<td>69</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 threads</td>
<td>35584</td>
<td>137058</td>
<td>78991</td>
<td>7</td>
<td>1381</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 threads</td>
<td>35584</td>
<td>161739</td>
<td>99333</td>
<td>11</td>
<td>2967</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 threads</td>
<td>35584</td>
<td>239562</td>
<td>156478</td>
<td>13</td>
<td>2758</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 threads</td>
<td>35584</td>
<td>323781</td>
<td>223414</td>
<td>20</td>
<td>3628</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28 threads</td>
<td>35584</td>
<td>385067</td>
<td>279500</td>
<td>32</td>
<td>4401</td>
<td>109</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 threads</td>
<td>35584</td>
<td>420242</td>
<td>301178</td>
<td>42</td>
<td>4906</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Table B.25: KMeansHigh Commits/Aborts - TCC protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation</th>
<th>Remote Validation</th>
<th>Update Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>53376</td>
<td>52002</td>
<td>0</td>
<td>48643</td>
</tr>
<tr>
<td>8 threads</td>
<td>53376</td>
<td>111326</td>
<td>19612</td>
<td>14100</td>
</tr>
<tr>
<td>12 threads</td>
<td>53376</td>
<td>151708</td>
<td>45835</td>
<td>1800</td>
</tr>
<tr>
<td>16 threads</td>
<td>53376</td>
<td>197015</td>
<td>74841</td>
<td>2212</td>
</tr>
<tr>
<td>20 threads</td>
<td>53376</td>
<td>246353</td>
<td>106084</td>
<td>2423</td>
</tr>
<tr>
<td>24 threads</td>
<td>53376</td>
<td>286610</td>
<td>136398</td>
<td>3152</td>
</tr>
<tr>
<td>28 threads</td>
<td>53376</td>
<td>340257</td>
<td>173631</td>
<td>3991</td>
</tr>
<tr>
<td>32 threads</td>
<td>53376</td>
<td>388100</td>
<td>212938</td>
<td>42</td>
</tr>
</tbody>
</table>

#### Table B.26: KMeansHigh Commits/Aborts - SL protocol

<table>
<thead>
<tr>
<th>Commits</th>
<th>Total Aborts</th>
<th>Local Validation</th>
<th>Lease Acquisition</th>
<th>Update Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>21345</td>
<td>26607</td>
<td>0</td>
<td>13607</td>
</tr>
<tr>
<td>8 threads</td>
<td>21345</td>
<td>85646</td>
<td>37809</td>
<td>14100</td>
</tr>
<tr>
<td>12 threads</td>
<td>21345</td>
<td>119041</td>
<td>81279</td>
<td>1800</td>
</tr>
<tr>
<td>16 threads</td>
<td>21345</td>
<td>177581</td>
<td>130887</td>
<td>2212</td>
</tr>
<tr>
<td>20 threads</td>
<td>21345</td>
<td>243956</td>
<td>199820</td>
<td>2423</td>
</tr>
<tr>
<td>24 threads</td>
<td>21345</td>
<td>313692</td>
<td>266671</td>
<td>3152</td>
</tr>
<tr>
<td>28 threads</td>
<td>21345</td>
<td>383476</td>
<td>332235</td>
<td>3991</td>
</tr>
<tr>
<td>32 threads</td>
<td>21345</td>
<td>457742</td>
<td>404686</td>
<td>4134</td>
</tr>
<tr>
<td>Threads</td>
<td>Commits</td>
<td>Total Aborts</td>
<td>Local Validation Aborts</td>
<td>Lease Acquisition Aborts</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>--------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>4</td>
<td>49372</td>
<td>129298</td>
<td>0</td>
<td>125492</td>
</tr>
<tr>
<td>8</td>
<td>49372</td>
<td>356832</td>
<td>11527</td>
<td>289560</td>
</tr>
<tr>
<td>12</td>
<td>49372</td>
<td>489314</td>
<td>4889</td>
<td>440144</td>
</tr>
<tr>
<td>16</td>
<td>49372</td>
<td>679714</td>
<td>74559</td>
<td>590226</td>
</tr>
<tr>
<td>20</td>
<td>49372</td>
<td>864002</td>
<td>108519</td>
<td>740363</td>
</tr>
<tr>
<td>24</td>
<td>49372</td>
<td>1047205</td>
<td>143230</td>
<td>890152</td>
</tr>
<tr>
<td>28</td>
<td>49372</td>
<td>1231818</td>
<td>176612</td>
<td>1040631</td>
</tr>
<tr>
<td>32</td>
<td>49372</td>
<td>1414302</td>
<td>210789</td>
<td>1189150</td>
</tr>
</tbody>
</table>

Table B.27: KMeansHigh Commits/Aborts - ML protocol

<table>
<thead>
<tr>
<th>Threads</th>
<th>Commits</th>
<th>Total Aborts</th>
<th>Lock Acquisition Aborts</th>
<th>Revoked Lock Aborts</th>
<th>Broadcast Aborts</th>
<th>Update Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>13344</td>
<td>26133</td>
<td>9071</td>
<td>0</td>
<td>82</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>13344</td>
<td>184297</td>
<td>90737</td>
<td>5</td>
<td>1731</td>
<td>69</td>
</tr>
<tr>
<td>12</td>
<td>13344</td>
<td>257782</td>
<td>150223</td>
<td>12</td>
<td>2658</td>
<td>59</td>
</tr>
<tr>
<td>16</td>
<td>13344</td>
<td>298887</td>
<td>196347</td>
<td>19</td>
<td>3655</td>
<td>92</td>
</tr>
<tr>
<td>20</td>
<td>13344</td>
<td>599094</td>
<td>392159</td>
<td>26</td>
<td>6048</td>
<td>94</td>
</tr>
<tr>
<td>24</td>
<td>13344</td>
<td>593466</td>
<td>438072</td>
<td>34</td>
<td>6335</td>
<td>100</td>
</tr>
<tr>
<td>28</td>
<td>13344</td>
<td>630767</td>
<td>465311</td>
<td>58</td>
<td>6637</td>
<td>109</td>
</tr>
<tr>
<td>32</td>
<td>13344</td>
<td>666368</td>
<td>490621</td>
<td>61</td>
<td>7039</td>
<td>110</td>
</tr>
</tbody>
</table>

Table B.28: KMeansHigh Commits/Aborts - Anaconda protocol

## B.2 APC

Figures B.1 to B.7 depict the APC metric per benchmark for each protocol. APC (Aborts per Commit) is a statistic that represents the ratio of aborts to commits for each configuration. This metric shows diagrammatically the level of contention of the configurations.
Figure B.1: Aborts per Commit of LeeTM

Figure B.2: Aborts per Commit of LeeTM-ER
**Figure B.3:** Aborts per Commit of KMeansHigh

**Figure B.4:** Aborts per Commit of KMeansLow
Figure B.5: Aborts per Commit of Vacation

Figure B.6: Aborts per Commit of Genome
Figure B.7: Aborts per Commit of GLife