UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Strongly Consistent Transactions for Enterprise Applications

Using Software Transactional Memory to Improve Consistency and Performance of Read-Dominated Workloads

Sérgio Miguel Martinho Fernandes

Supervisor: Doctor João Manuel Pinheiro Cachopo

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor Pascal Felber

Doctor Hervé Miguel Cordeiro Paulino

Doctor João Manuel Pinheiro Cachopo

Doctor João Pedro Faria Mendonça Barreto

2014

UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Strongly Consistent Transactions for Enterprise Applications

Using Software Transactional Memory to Improve Consistency and Performance of Read-Dominated Workloads

Sérgio Miguel Martinho Fernandes

Supervisor: Doctor João Manuel Pinheiro Cachopo

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor Pascal Felber, Professor Catedrático, Institut d'Informatique, Université de Neuchâtel, Suiça

Doctor Hervé Miguel Cordeiro Paulino, Professor Auxiliar da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Doctor João Manuel Pinheiro Cachopo, Professor Auxiliar do Instituto Superior Técnico, da Universidade de Lisboa

Doctor João Pedro Faria Mendonça Barreto, Professor Auxiliar do Instituto Superior Técnico, da Universidade de Lisboa

Funding Institutions

FCT Fundação para a Ciência e Tecnologia

2014

Resumo

No desenvolvimento de aplicações empresariais, a tendência tem sido relaxar os critérios de consistência oferecidos pelo middleware transacional, procurando melhorar a escalabilidade e o desempenho desses sistemas. Tal decisão coloca novos desafios aos programadores de software: Estes têm de ter presente esta consistência fraca ao programar a lógica aplicacional, o que tipicamente conduz a um esforço extra de desenvolvimento para lidar com possíveis inconsistências resultantes de execuções concorrentes, o que não é trivial e propicia erros nos programas.

Nesta dissertação, eu proponho o desenho de um novo sistema de middleware que fornece transações com consistência forte e, ao mesmo tempo, melhora o desempenho das aplicações empresariais cujas operações são predominantemente de leitura. Este desenho baseia-se na mudança do controlo transacional da base de dados para o código da aplicação. Para isso usa a tecnologia de Memória Transacional de Software para garantir a consistência forte e estende a MTS com as capacidades de operar com dados persistentes e funcionar em ambiente distribuído. Eu apresento duas implementações complementares do sistema de middleware: Primeiro uma alternativa mais simples baseada no uso de trincos—mais adequada para aplicações de um só servidor—e depois uma alternativa que usa sincronização não bloqueante baseada na primeira MTS multi-versão completamente livre de bloqueios, que também é proposta nesta dissertação.

Os resultados experimentais obtidos com as duas implementações mostram que este desenho melhora consistentemente o desempenho das aplicações empresariais típicas—em até cem vezes nalgumas configurações—garantindo ainda transações com consistência forte.

Abstract

The trend in the development of enterprise applications has been to relax the consistency guarantees provided by the transactional middleware, with the goal of improving the resulting systems' scalability and performance. Such decision, however, poses new challenges to software developers: They must now take into account this weaker semantics when programming application logic, which typically leads to additional, nontrivial and error-prone development effort, to cope with possible inconsistencies produced by concurrent executions.

In this dissertation, I propose a new middleware design that provides transactions with strong consistency and, at the same time, improves the performance of the typical read-dominated workloads observed in enterprise applications. This design is based on seamlessly shifting transaction control from the database into the application code. It does so using Software Transactional Memory to ensure strong consistency, and then extending the STM with additional capabilities to handle persistent data and to operate in a clustered environment. I present two complementary middleware implementations based on the same principle: First, a simpler lock-based alternative—best suited for single-node applications—and then a nonblocking alternative based on the first entirely lock-free multi-version STM, which is also proposed in this dissertation.

The experimental results obtained with the implementations of this middleware design confirm that it consistently improves the performance of the typical workloads—by as much as a hundredfold in some configurations—while executing strongly consistent transactions.

Palavras Chave

Transações

Consistência Forte

Memória Transacional de Software

Programação Concorrente

Sincronização não Bloqueante

Algoritmos Livres de Bloqueio

Arquitetura de Aplicações Empresariais

JVSTM

Middleware

Operações Predominantemente de Leitura

Keywords

Transactions

Strong Consistency

Software Transactional Memory

Concurrent Programming

Nonblocking Synchronization

Lock-free Algorithms

Enterprise Application Architecture

JVSTM

Middleware

Read-Dominated Workloads

Acknowledgements

Several people have helped, either directly or indirectly, in the making of the work herein, and without their assistance this work would not have been possible. I wish to express my heartfelt thanks for all their invaluable contributions.

First and foremost, I would like to thank my advisor and friend Professor João Cachopo. I am without words to faithfully express how important his guidance has been throughout these years. Time and again, he has helped me to improve not only as a researcher, but also as a person in general, and learning from him on a daily basis has been one of the most rewarding outcomes of this work. In fact, he has been a source of inspiration from the first day I met him—many years ago, during my BSc, when he was lecturing the course on Artificial Intelligence. His clarity of thought and ability to provide clear explanations, for even the most complex matters, makes it a pleasure to discuss any matter with him. It is an honor to have him as my advisor.

I would like to thank Professor António Rito da Silva, for his guidance in the earliest stages of my work, and for helping me to set off in a good direction. I acknowledge also Professor Luís Rodrigues for his yearly feedback as a member of my thesis commission.

I would like to thank all the members, past and present, of the Software Engineering Group at INESC-ID, with whom I have contacted, for all the great discussions on multiple research subjects, including my own. Everyone has contributed somehow with thought provoking ideas or suggestions to improve my research. I also thank Pedro Ruivo from the Distributed Systems Group, for is constant availability and help with the configuration of Infinispan and JGroups.

I am thankful to INESC-ID for providing me with a comfortable environment and the resources I needed to perform my work. I wish to acknowledge *Fundação para a Ciência e Tecnologia* for partially supporting my research with a grant. Also, I thank Dr. Cliff Click for

kindly providing access to hardware from Azul Systems, on which I could run some of the experimental evaluations.

I would like to thank everyone who has contributed to the development of the new version of the Fénix Framework, in particular Pedro Santos, David Martinho, João Neves, and João Carvalho, with whom I had several insightful conversations, which allowed me to make critical choices regarding the refactoring of the Fénix Framework. Not forgetting, of course, my advisor, who was responsible for the initial version and contributed with a never ending list of ideas for the new version.

I would like to thank all my friends, and especially those who kept asking "*Are you there yet?*" throughout the years, reminding me not to stray from the path: Not only, but most notoriously to Ricardo Português and Rui Nunes.

I would like to thank my godparents, António and Fátima, for being an example of encouragement to tackle any adversities.

I would like to thank my mother Helena, and my grandmother Zulmira, for always being there for me and the rest of my family, and for contributing, since ever, to shape who I am today. They have laid the foundations for every success I ever achieve.

To my parents-in-law João and Mavilde, I thank for all the help with family support, especially during the final stages of this work.

Last, but never least, I thank my wife Catarina, with whom I am privileged to build a family. Thank you for your unconditional belief in me, and thank you for our lovely children Inês and Miguel. You three make my life even more rewarding every day!

February 2014

Sérgio Miguel Martinho Fernandes

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations and Acronyms

API	*Application Programming Interface* 5
ATR	*Active Transactions Record* 38
CAS	*Compare-And-Set* 65
DBMS	*Database Management System* 2
DML	*Domain Modeling Language* 196
GC	*Garbage Collection* 4
IaaS	*Infrastructure as a Service* 186
IMDG	*In-Memory Data Grid* 11
IST	*Instituto Superior Técnico* 22
JVM	*Java Virtual Machine* 35
JVSTM	*Java Versioned Software Transactional Memory* 4
OOP	*Object-Oriented Programming* 10
ORM	*Object-Relational Mapping* 10
SQL	*Structured Query Language* 8
STM	*Software Transactional Memory* 2
TDR	*Transactional Data Repository* 32
TM	*Transactional Memory* 12
UUID	*Universally Unique Identifier* 146
VBox	*Versioned Box* 35
VM	*Virtual Machine* 186

1 Introduction

Transactions are undoubtedly a useful mechanism in the development of software. Many applications use transactions or an equivalent mechanism to execute blocks of code atomically, with an all-or-nothing semantics. In case of success, the effects of a transaction appear to occur at a single point in time, and if, for some reason, the transaction fails, the state of the application, prior to executing the transaction, is restored automatically. Application programmers also expect their transactions to execute in isolation from other concurrent changes, and for the visible effects of a transaction to be consistent with the other computations.

Traditionally, transactions have been associated with databases. However, the use of transactions goes beyond such context and has been applied in more systems. Initially, upholding transactional consistency was relatively simple, because databases where monolithic and centralized. The existing computational power was at the time located in a single big server and changes to shared data were contained in the same shared memory space, which made them easy to track.

The evolution in computer systems led to a pervasive existence of computer hardware with enough computational power to store and to operate over large amounts of data. The concept of a transaction has remained useful, but as systems grew and became more complex, widespread, and interconnected, ensuring the original strong transactional semantics became expensive and a hindrance to scalability and performance.

As a result of these two forces, for one the usefulness of transactions—hence the desire to continue to use them in programming software applications—and for another the increased difficulty in upholding the transactional semantics in complex systems, transactions have evolved to provide several different semantics, most notoriously, with lessened consistency guarantees.

This varied set of transactional semantics makes it more difficult for programmers to write code: Not only are the guarantees that transactions provide typically weaker, but also

it is often difficult to understand the resulting interplay of concurrent transactions in the presence of weak consistency models, or even that of systems with different transaction models.

In this dissertation, I leverage on (1) the evolution of computational power, (2) research on transactions—especially in *Software Transactional Memory* (STM)—and (3) the typical architecture and workloads of enterprise applications to propose a transactional middleware that ensures strong consistency for transactions, while being able to improve the performance of access to transactional data in the common case. The solution that I propose is not a panacea, as not all workloads may benefit of the performance improvements. Nevertheless, even under the circumstances where the performance does not improve, this approach is still of use when some performance penalty may be an acceptable price to pay compared to the gains brought about by the strongly consistent transactions.

In the remainder of this chapter, first I present my thesis statement, then I summarize the contributions of my work and, lastly, I describe the structure of this dissertation.

1.1 Thesis Statement

From a software engineering perspective, I aim to reduce the effort required to develop applications. I believe that providing a transactional model that offers strong consistency contributes to this goal. Apparently, however, to provide strong consistency negatively affects performance so much so that the trend has been for transactional systems to relax consistency in favor of performance.

> This dissertation's thesis is that it is possible **to provide a model of strongly consistent transactions** to application developers and, simultaneously, **to improve the performance for the typical workloads of a class of applications**, by using a middleware that leverages on recent advancements both in hardware and in software.

More specifically, I propose to combine the use of recent computers with commodity hardware featuring many cores and large amounts of main memory with STM, to be able:

- To shift transaction control from its typical location—the *Database Management System* (DBMS)—to a dedicated component that is integrated into the application tier, thereby

being as close as possible to where the data-manipulation transactional operations take place. This component is implemented using STM, and it provides transactions with strong consistency. Unlike typical transactional systems, whose consistency semantics depends, to some degree, on the consistency semantics of the underlying DBMS, the solution that I propose always provides strong consistency, regardless of the guarantees offered by the concrete DBMS in use.

- To improve the performance of read-dominated workloads in domain-intensive applications, by greatly reducing the cost of the reads of transactional data. This is achieved by using an STM-backed cache for the transactional data accessed by the application: This cache reduces or, in some cases, even eliminates the need for round trips to fetch data from the DBMS, and it provides much faster reads than the typical cost of accessing the DBMS. This is true even when compared to using in-memory DBMSs.

In this dissertation, I provide the **design** of a concrete middleware, dedicated to the support of transactions in enterprise applications. This middleware, which provides strongly consistent transactions regardless of the underlying DBMS, can seamlessly replace the existing transactional system of an enterprise application and it relies on the DBMS for data storage only. Moreover, it poses very few requirements on the DBMS, making it practical to use this middleware with virtually any DBMS.

To demonstrate the feasibility of this solution and validate my thesis, I perform a complete **implementation** of the proposed middleware and perform an evaluation, in which I measure application performance with and without using this middleware, for a set of representative workloads and application deployment variations, on top of some of the available DBMSs currently in use.

1.2 Contributions

In this dissertation, I provide the following contributions:

- I propose a **new architecture for enterprise applications, which shifts transaction control to a dedicated component, built with STM**. This component provides to its application a well defined and strong transactional semantics that, unlike traditional designs, is independent of the transactional semantics of the underlying

DBMS of choice. Additionally, this design can greatly improve the performance of read-dominated workloads by reducing the cost of the accesses to shared data.

- To implement the design that I propose, I develop **a straightforward extension of the *Java Versioned Software Transactional Memory* (JVSTM) with support for both data durability and deployment of multiple application instances in a cluster environment**. This extension is designed with evolution in mind, by making it practical to replace the transactional system of existing applications, and by using the existing DBMS to store application data. As such, the extended version of the JVSTM immediately enables developers to use STM in real-world enterprise applications.

- I develop **a new efficient and nonblocking (lock-free) commit algorithm for the in-memory JVSTM only**. This new algorithm eliminates the potential bottleneck of the single commit lock, used in the original JVSTM, and it increases the scalability and performance of the JVSTM when executing write transactions in systems with many cores, where factors such as (1) the percentage of write transactions, (2) the number of concurrent transactions and (3) the duration of the commit phase could impair the performance of the JVSTM and, consequently, of the application. Also, I develop **a new *Garbage Collection* (GC) algorithm for the lock-free JVSTM**, which asynchronously cleans up unreachable versions and reduces the overhead of starting/finishing a transaction, imposed by the existing GC. The lock-free commit algorithm stems from the need to improve the performance of the extended JVSTM, which increases the duration of the commit phase, by holding the commit lock while communicating with the DBMS. The new GC algorithm stems from the performance improvements obtained with the lock-free commit, which unveiled a dormant bottleneck in the original GC. Both improvements target the in-memory JVSTM. To the best of my knowledge, this is the first entirely lock-free implementation of a multi-version STM.

- I integrate the previous two contributions into the design of a best-effort **nonblocking STM-based middleware for transactional support in enterprise applications**. By best-effort, I mean that my entire implementation of this middleware is nonblocking. However, because it depends on two external components—the underlying DBMS and the distributed communication infrastructure—whether the full middleware stack behaves in a nonblocking fashion depends on the concrete implementations of these external components.

1.3 Structure of the dissertation

In this introductory chapter, I focus on the presentation of my thesis statement, the summary of the contributions from my work, and the outline of the remaining chapters in this dissertation, which are structured as follows:

Chapter 2 Context, Problem Statement, and Solution Overview. This chapter provides the background for my work. I begin with an historical overview of enterprise application development, where I frame its evolution in a path that has lead to the current state, in which programmers are burdened by the lack of a strong transactional semantics. I discuss the problems associated with this deficiency, and state the problem that I address with my thesis statement. Afterwards, I present the proposed approach to the solution and some statistical data on the use of STM in real-world enterprise applications, which further motivates my work and substantiates the viability of my design choices.

Chapter 3 TMM: A Transactional Memory Middleware for Enterprise Applications. This chapter presents the basic solution that provides transactions with strong consistency and improved performance for the typical workloads of enterprise applications. It describes how to integrate STM into existing enterprise applications, and how to extend the in-memory STM, thereby creating a middleware with support for data storage and the ability to deploy multiple application nodes running STM-backed transactions with strong cluster-wide consistency.

Chapter 4 Lock-free JVSTM. Even tough adequate for a single node, the solution from the previous chapter has poor scalability when applied to a distributed system, due to the use of a global lock. In this chapter, I concentrate on the scalability of the in-memory STM: I present the modifications to the original lock-based STM—namely the new commit and GC algorithms—which improve its performance and enable it to scale for a large number of cores in a single machine. This new STM implementation is entirely nonblocking, more specifically lock-free.

Chapter 5 Nonblocking TMM. This chapter combines the ideas proposed in the two previous chapters, resulting in an STM-based middleware that can be integrated into enterprise applications to provide transactions with global ordering and strong consistency, without requiring a global lock. The resulting STM implementation is also nonblocking, apart from any blocking synchronization used by the underlying *Application Programming Interface* (API) on which the STM depends—namely the distributed communication mechanisms and data

storage operations. If these APIs can be guaranteed to never block indefinitely, then the STM implementation is in effect nonblocking as well.

Chapter 6 Conclusions. This chapter concludes my dissertation with a summary of the work presented and possible directions for future work.

1.4 Notation

In this dissertation, I need to present and discuss some algorithms. I will show them in listings format, using code written in the Java language. Given that the main goal of showing the code is to clearly convey the algorithms to the reader, for the sake of clarity in the presentation, I have removed accessorial things, such as visibility qualifiers. Exceptions have been made where relevant. Also, often, I elide some details from the actual implementation when they are not relevant to the part of the algorithm being addressed: As such, I use "..." (ellipsis) to represent code that as been elided. I discretionarily use **bold** font to highlight some parts of the code shown, typically method names and class attributes. When reusing or extending parts of an algorithm that has been shown previously, I use a faded gray color. When referring to elements of code in running text I use a `typewriter` font.

2 Context, Problem Statement, and Solution Overview

This chapter sets the background on top of which my work is based. I begin by presenting a perspective on the evolution of enterprise application development, in which I focus on how it has affected the transactions' consistency and overall application performance, leading to the *status quo* in which programmers are burdened with additional nontrivial and error-prone coding effort. After that, I present STM and associated concepts that are relevant for the work in this dissertation. Afterwards, I state the problem that I intend to address, and I outline the solution that I propose to implement. In doing so, I also provide statistical data that backs up some of the assumptions I make regarding some characteristics of typical enterprise applications.

2.1 A Historical Perspective of Transactions in Enterprise Applications

Over the last forty years, the development of enterprise applications has evolved, influenced by diverse factors such as the changes in hardware, or the changes in the users' expectations about how applications should behave. Often, these changes had a reflection in the architecture of applications. Still, one thing has remained the same for most applications: The underlying DBMS provides not only persistence but also the semantics of the transactions on which application developers rely for programming business transactions.

Unfortunately, this means that, when using transactions, developers tend to be limited by the isolation levels provided by the DBMS, which often does not provide serializability—a correctness property for concurrent transactions that is crucial to ensure the integrity of the applications' data.

Historically, transactional systems have weakened the isolation level of transactions as a trade-off between correctness and performance, which resulted in the generally accepted idea that serializability is incompatible with performance. This line of reasoning is clearly illustrated by the following passage from Martin Fowler's well-known book on patterns of

enterprise application architecture [Fowler, 2002]:

> To be sure of correctness you should always use the serializable isolation level. The problem is that choosing serializable really messes up the liveness of a system, so much so that you often have to reduce serializability in order to increase throughput. You have to decide what risks you want take and make your own trade-off of errors versus performance.
>
> You don't have to use the same isolation level for all transactions, so you should look at each transaction and decide how to balance liveness versus correctness for it.

Even though this book is from 2002, I believe that it still represents faithfully the actual state of the practice in the development of most enterprise applications, in which consistency is often traded-off for performance.

One of the problems with the existence of weaker transactional semantics is that it burdens developers with additional, nontrivial and error-prone development effort, especially when developing applications that require stronger semantics than what is provided by the transactional system being used. The ability to program using serializable transactions simplifies the reasoning, reduces the possibility of bugs, and reduces the overall development effort. However, this option—to use strongly consistent transactions—is seldom, if ever, available to developers.I shall return to this problem in more detail later in this chapter. Before that, however, I provide a brief historical perspective of the development of client/server applications and relate it to the *status quo* of enterprise application development.

In this dissertation, I use the term *database* as a general term to refer to a destination designated for storage of application data. A DBMS is a component that provides users or applications with an interface to manipulate stored data, possibly using transactions. Historically, DBMSs are associated with the storage of data using the relational model and with providing a *Structured Query Language* (SQL) interface for the data manipulation operations. Throughout this dissertation, when referring to a database or a DBMS, I do not intend to imply any concrete system, such as the underlying data model or the topology of data organization, unless specifically mentioned: Both should be considered in their most generic definition.

2.1.1 The Shift From a 2-tier to a 3-tier Architecture

A core component of most modern enterprise applications is a DBMS, and its use in such applications goes a long way back. The first enterprise applications were 2-tiered—that is, they had a simple client/server architecture. The clients made their requests directly to the database server, which in turn executed the requested operations and gave back the results. This architecture was adequate in a scenario where all the clients were on a trusted network and most of the computational power resided on a big server. In this architecture, transactional control was placed in the (only) obvious location: The database server. Each client request would perform within a database transaction, and the server ensured the ACID properties [Gray, 1981].

Database servers were expensive and they had to cope with a growing usage demand. Eventually, different semantics for the ACID properties appeared, which relaxed some of the properties (e.g., isolation [International Standards Organization, 1992]), mostly for performance reasons. As client machines became more and more powerful, more computation could be performed on the client side, which would also take part in ensuring data consistency. Business logic consistency started to get more complex than simple low-level consistency checking (e.g., referential integrity), including complex high-level consistency (e.g., a list can only contain odd numbers). Often, in this architecture the business logic code was intertwined with the user interface code. Nevertheless, transaction control remained centralized on the server.

With the growth of the internet and the World Wide Web, a new architecture emerged. As organizations felt the need to interconnect their systems, the 2-tiered architecture no longer served their purposes. There were several reasons for this, including the systems' security and the use of network bandwidth. The clients (now including systems in diverse geographical locations and outside the control of the intranet) were no longer trusted. The number of clients grew. The available bandwidth was far less than it was previously available on the intranet, such that sending large volumes of data, as the result of a database query, was no longer viable. This led to a 3-tiered architecture on which the server side was decomposed in two tiers: One for the application server and another for the database server. The database server was still responsible for the data persistence and for ensuring transactional access to its data. The application server was responsible for executing the complex business logic and interfacing with the clients, which would provide the user interface. This new architecture presented, at least, two main advantages over the previous: (1) the information was

kept safe within the intranet and it was only accessed directly by a trusted system; and (2) large queries could be obtained and processed on the server-side before sending the results over the internet to the clients. The downside, with regard to transactions, was that the transaction logic was coded in the application server, whereas the responsibility for ensuring transactional properties was in the database server. Application server logic was typically stateless and relied on the data stored in the database to process every client request, which meant that adding more application servers was cheaper and simpler than adding database servers. So, database performance was still a critical aspect in this architecture and, notably, relaxed transactional semantics remained in use.

2.1.2 Application Logic Becomes Increasingly Complex

As the internet and the number of interconnected servers kept growing, another revolution was taking place in the software development area: *Object-Oriented Programming* (OOP) languages became mainstream and started to be used commonly in the development of large server-side applications. Characteristics such as component modularization, ease of reuse, data encapsulation, and abstraction, helped developers control the growing complexity of their applications.

The adoption of OOP, however, created a mismatch between the most common persistent representation of data and their in-memory representation, known as the object-relational impedance mismatch [Ireland et al., 2009]. The development of *Object-Relational Mapping* (ORM) tools was the industry's answer to handle this mismatch. The purpose of an ORM tool is to take care of the data mapping between the object-oriented model and the relational model, such that the programmers could write code in the OOP environment and the ORM layer would handle the conversion from/to the underlying relational structure. Whereas, in part, these tools simplified the programmers' coding efforts, they also created some difficulties of their own, such as the maintenance of O/R-mapping metadata, and the varied semantics implied by object caching, lazy loading, and support for *ad hoc* queries. Additionally, different ORMs provided different semantics.

Yet, most notably, the features provided by ORM tools kept depending on transactional support that was still under the responsibility of the underlying DBMS, which, in turn, still offered different flavors for ACID semantics, but none included support for strict serializability [Papadimitriou, 1979]. In fact, the isolation guarantees provided by databases have been, for long, a matter of confusion and discussion [Berenson et al., 1995].

2.1.3 The Growth in Computational Power

Up until recently, powerful database servers were very expensive and held much computational power when compared to the simpler application servers. In recent years, however, the computational power has grown dramatically, which has brought about commodity hardware that features many CPU cores and large amounts of main memory at a very low cost.

The availability of many low-cost and powerful servers together with the improvements in network technology have contributed to a reinvigorated interest in distributed computing, quite notoriously pushed through *cloud computing*: This form of computing relies on economies of scale to offer clients an abstract view of computational resources that can be provisioned on demand over a computer network (commonly referred to as the *cloud*).

Concomitantly, there has been the development of new programming models (e.g. MapReduce [Dean and Ghemawat, 2008]), large scale data storage systems (e.g. Amazon's S3 [Brantner et al., 2008]), along with various (weakened) consistency models (e.g. Eventual Consistency [Vogels, 2009; Pritchett, 2008]).

Meanwhile, the scalability and overall appropriateness of relational DBMSs was being questioned [Stonebraker et al., 2007] and distributed NoSQL key-value data stores were emerging as the *de facto* standard DBMS in the cloud, with the creation of a multitude of key-value data store implementations, such as BerkeleyDB [Olson et al., 1999], Dynamo [DeCandia et al., 2007], Cassandra [Lakshman and Malik, 2010], Oracle Coherence [Seovic et al., 2010], HBase [Jiang, 2012], Infinispan [Marchioni and Surtani, 2012], Hazelcast [Johns, 2013], and several others.[1] A key-value store uses a simple data model that maps an arbitrary name (key) to arbitrary data (value). These are typically, but not exclusively, implemented as an *In-Memory Data Grid* (IMDG) as a means to reduce the costs associated with disk-bounded operations, typical of traditional DBMSs. An IMDG is a type of DBMS that primarily relies on main memory for data storage. A major benefit of using an IMDG is the performance improvement achieved by using the main memory for storing data, which is much faster than using external storage. When data durability is a requirement, it may be provided by writing data to traditional DBMSs, by using new nonvolatile memory technology, or simply by ensuring that a minimum number of data replicas always exist. An IMDG also leverages on the fact that most workloads in enterprise applications are read-dominated, so that, even when using a disk-based solution for data durability, most operations will still be

[1] A growing list of key-value stores is maintained at `http://en.wikipedia.org/wiki/NoSQL#Key.E2.80.93value_stores`.

much faster, as they access main memory only.

In spite of the growing popularity of different systems and models for transactions and data storage, relational DBMSs are still widely used, with anecdotal evidence[2] suggesting that they are still the predominant type of DBMS.

Just as in early evolution of enterprise application architecture, a common theme in all these recent developments, is that the focus has been on improving performance: Unfortunately, it has come at the expense of reducing the consistency guarantees. To this effect have contributed not only the perseverance of old ideas about the development of databases and transactions, but also demonstrations of how consistency requirements can affect other desirable properties in a distributed system, namely availability and partition tolerance [Brewer, 2000; Gilbert and Lynch, 2002].

The natural evolution in software and hardware has led to the current state, in which many developers depend on a multi-tiered architecture, develop enterprise applications with complex transactional logic, and rely (either directly or indirectly) on a DBMS for persistence and transactional support.

2.2 Software Transactional Memory

Since the seminal work on *Transactional Memory* (TM) [Herlihy and Moss, 1993], and the following proposal of its software-based version (STM) by [Shavit and Touitou, 1995], a few years passed without much interest in this topic. Soon after the time when the first multicore processors became commercially available, interest in STM was rekindled and, since then, this has been an area of growing research interest, leading to several proposals for STM implementations, each with their own set of characteristics [Herlihy et al., 2003b; Harris and Fraser, 2003; Marathe et al., 2004; Harris et al., 2005; Dice et al., 2006; Spear et al., 2008; Felber et al., 2008b; Dalessandro et al., 2010].

STM is a concurrency control mechanism for concurrent programming. It supports the generic notion of an atomic block of code that can perform shared memory accesses with transactional guarantees. The concept of STM transactions shares some resemblance with that of database transactions, but has specific characteristics that make its implementations differ from those of database transactions [Felber et al., 2008a]. Both types of transactions

[2] `http://db-engines.com/en/ranking_categories`

focus on controlling access to shared data and providing an all-or-nothing semantics for the actions within the transaction. However, because STM transactions operate over transient data, they do not need to be concerned with aspects related to external data storage, nor provide guarantees of data survival or state recovery in case of a system crash. Instead, STM puts a strong emphasis on aspects related to program safety and liveness. To a large extent, theoretical developments on STM have focused on the correctness criteria and progress conditions that are ensured to the application code when using this concurrency mechanism. In the following, I summarize those concepts which are relevant to the work that will be described ahead.

2.2.1 Correctness Criteria

From the perspective of the user of STM, transactions must appear to execute atomically at a single indivisible point in time. For this reason, a correct execution will be one in which, although several transactions may progress concurrently, their outcome must be equivalent to one where they executed sequentially, one at a time. This is represented in the concept of *serializability* [Papadimitriou, 1979], which originates from database transactions. It states that any set (or history) of transactions is serializable if the execution of all committed transactions in the set can be explained by some sequential ordering of those transactions.

A stronger requirement over serializability is *strict serializability* [Papadimitriou, 1979], which additionally imposes that the sequential order that explains the outcome of a history of transactions must preserve the order of temporally nonoverlapping transactions, that is, if a transaction T_1 is seen to commit before another T_2 starts, then the effects of T_1 must necessarily be seen by T_2, because T_2 cannot be reordered before T_1 to justify a strictly serializable execution.

The two criteria above leave open the possibility for transactions that do not commit, such as those that abort or enter infinite loops, to see inconsistencies. To tackle this situation the criterion of *opacity* [Guerraoui and Kapalka, 2008] additionally imposes that all transactions access consistent data, regardless of their outcome.

These three criteria are useful to discuss the correctness of a set of operations, in this case, a set of transactions. When dealing with the individual execution of a concurrent operation, such as the commit of a transaction, the concept of *linearizability* [Herlihy and Wing, 1990] comes into play. Linearizability is akin to strict serializability, except that it is

a criterion that characterizes the execution safety of operations in a concurrent object. A linearizable operation is one that appears to take effect instantaneously at some moment between its invocation and response. Linearizability is an important and desirable property, because it is compositional and nonblocking [Herlihy and Wing, 1990]. The point where the effects of a linearizable operation become visible to others is called the *linearization point*. For lock-based operations, the linearization point typically occurs within the critical section. For operations that do not use locking, the linearization point must be such that it is seen by others as an indivisible operation.

2.2.2 Progress Conditions

A system that only handles sequential executions is free to progress at will, apart from any bugs or purposeful halting. Concurrent executions, however, may get in each other's way and, consequently, limit each other's progress. Progress conditions state how each thread or the system as a whole behaves under concurrent executions.

Progress guarantees can be either *blocking*, in which the delay of any one thread can prevent others from continuing, or *nonblocking*, in which a thread's execution may not be prevented by others [Herlihy and Shavit, 2008].

A nonblocking algorithm can be characterized as *wait-free* [Herlihy, 1991] if it ensures that every thread will be able to execute its tasks in a finite number of steps, regardless of what other threads may be doing. It is *bounded wait-free* if it is possible to establish a bound on the number of steps required to complete the algorithm. Wait-free (and consequently bounded wait-free) algorithms can never be prevented from completing, hence they are *starvation-free*. If individual thread progress cannot be guaranteed, but still the algorithm is nonblocking, and there is at least one thread that is always able to make progress, then the algorithm is said to be *lock-free* [Dongol, 2006]. This type of algorithm allows for individual threads to suffer from starvation, but not all threads at the same time. Lock-freedom is a weaker property than wait-freedom, but in practice lock-free algorithms tend to be much more efficient and, in general, all threads will eventually make progress, under the typical scheduling algorithms of common operating systems [Fraser, 2004].

Another nonblocking progress condition is *obstruction-freedom* [Herlihy et al., 2003a], which guarantees that any one thread will make progress if it executes in isolation for sufficient time. This progress condition is even weaker than lock-freedom, because it does not

guarantee that any progress is ever made when multiple threads execute concurrently. In contrast to wait-freedom and lock-freedom, obstruction-freedom is a *dependent* progress condition [Herlihy and Shavit, 2011], because progress of the system relies on guarantees provided by the underlying platform. Even though the upside of obstruction-freedom is that it allows for the most efficient implementations of all three nonblocking progress conditions, its usefulness has been questioned [Ennals, 2006; Dice and Shavit, 2007] mainly based on the argument that, within the set of dependent progress conditions, this is the least efficient type of implementation.

The other dependent progress conditions are both blocking: *Starvation-freedom* guarantees that every thread that attempts to access a shared resource will eventually be able to do so, and *deadlock-freedom* guarantees that at least some thread will be able to make progress, even if all others starve. These two progress conditions are akin to wait-freedom and lock-freedom, respectively, except in that a thread accessing the shared resource may prevent all other threads from executing, causing them to block.

2.3 Problem Statement

In Section 2.1, I have described how the evolution in the development of enterprise applications has led to a state where providing strong transactional semantics is possible, but expensive, so much so that in order to have acceptable performance, transactional systems sacrifice consistency. Additionally, I argue that part of the performance penalty stems from the legacy use of DBMS both as data repositories and, ultimately, as transaction providers for the applications' business logic, leading to many costly interactions between the two components.

> **The problem**, from a software development perspective, **is that programmers are burdened with additional coding effort, which is required to solve the possible data inconsistencies that may arise from lacking transactional semantics**.

In this dissertation, I concentrate my attention on the development of complex enterprise applications that require transactional support and data persistence. By complex, I mean applications that have a rich domain structure with many relationships among the various domain entities, as well as business logic with nontrivial rules. Moreover, I assume that

these applications have many more read-only than write business transactions, typically in the rate of 10 to 1 or higher. In summary, the common characteristics that describe the class of applications on which I concentrate are the following:

- Operations can execute concurrently.
- Business logic is implemented with transactions.
- Application state is stored in some data repository—a DBMS.
- Business logic is complex, i.e. transactions are not simple CRUD[3] operations.
- The workloads contain many more reads than writes, typically in a ratio of 10 to 1 or higher.

For applications that fit the above characteristics, which I generically name *enterprise applications*, I identify two difficulties that derive from current implementations of the typical multi-tiered architecture: One is the difficulty in ensuring consistency; the other is the reduced performance of the application server in the processing of complex operations due to its dependence on data from a separate component—the DBMS—even when using common techniques such as data caching. The following two subsections provide further discussion on each of these difficulties.

2.3.1 Consistency Problems

If given the possibility, and all else being equal, it stands to reason that every programmer would like to have no less than the strong consistency guarantee of strictly serializable transactions, when programming concurrent transactional operations that manipulate the shared state of the entities in their code. Having such guarantee shields programmers from concurrency hazards, and allows them to write cleaner and simpler code. For example, the authors of Google F1 [Shute et al., 2013] clearly state the importance of consistency as a requirement for their system:

> The system must provide ACID transactions, and must always present applications with consistent and correct data. Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.

[3] Create, Read, Update, Delete.

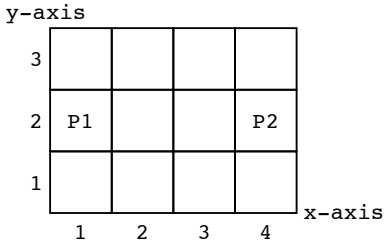

Figure 2.1: Two players occupying nonadjacent positions on a map.

To illustrate this point consider the following example: Imagine a multi-player game where players can concurrently move their pieces in a map from one point to another. The only restriction is that, after each movement is performed, no player can be in the immediate vicinity of another player. Now consider the starting scenario depicted in Figure 2.1, which shows part of the map containing two players, P_1 and P_2.

Suppose that, concurrently, P_1 attempts to move one position to the right and P_2 attempts to move one position to the left. Only one of the moves can succeed, because otherwise the two players would end in a position next to each other, which is illegal. In Listing 2.1 on page 18, I present a possible implementation of the `moveTo(int, int)` operation: The operation checks that both the target position and its surroundings are available and, if that is the case, it updates the position of the player on the map.

This code looks quite trivial and, when several moves are executed concurrently, each within its own transaction, the programmer should expect the application to work just fine—that is, after each transaction finishes, the moved player should be in a location that does not contain any adjoining players, thus maintaining the domain consistent. Sadly, this may not be the case if transactions are executed with the isolation level provided by most of today's mainstream transactional systems, which ensure snapshot isolation, at best. Under snapshot isolation a transaction may commit even if the values that were read changed in the meanwhile, as long as concurrent writes do not intersect. In this example, if transaction T_1 executes P_1`.move(2,2)` and transaction T_2 executes P_2`.move(3,2)`, then Table 2.1 on page 18 presents the points in the read set and write set of each transaction with regard to map coordinates.

Note that write sets do not intersect, and as such, under snapshot isolation both transactions will be allowed to commit, leading to an inconsistent domain state. This problem is well known by the name of **write skew** [Fekete et al., 2004]. Yet, with the given transactional semantics, the current solution is to put on the programmer's shoulders the responsibility

```
1  class Player {
2    ...
3    void moveTo(int x, int y) {
4      if (cell(x,y).available(this) &&
5          cell(x+1,y).available(this) &&
6          cell(x-1,y).available(this) &&
7          cell(x,y+1).available(this) &&
8          cell(x,y-1).available(this)) {
9        this.currentCell().clear();
10       cell(x,y).set(this);
11     } else {
12       throw new Exception("Move not allowed");
13     }
14   }
15 }
16
17 class Cell {
18   ...
19   boolean available(Player p) {
20     return this.isEmpty() || this.holdsPlayer(p);
21   }
22 }
```

Listing 2.1: Possible implementation of the `moveTo(int, int)` operation. The programmer checks the surroundings to ensure that the move is allowed and then writes to the destination.

Tx	Read set	Write set
T_1	(2,2), (3,2), (1,2), (2,3), (2,1)	(1,2), (2,2)
T_2	(3,2), (4,2), (2,2), (3,3), (3,1)	(4,2), (3,2)

Table 2.1: Read and write sets for concurrent transactions T_1 and T_2 with regard to map coordinates. T_1 and T_2 write to adjacent places and their respective write sets do not intersect.

of forcefully creating a conflict between the two transactions. In this case, one way of programmatically forcing the conflict is to call `clear()` for each of the positions surrounding the destination, thus causing an intersection in the write sets [Cahill et al., 2009]. This is definitely something undesirable from the programmer's perspective, and very much error-prone in complex applications where the potential conflicts might not be easily identified, as they may occur due to the interaction of many functionalities.

2.3.2 Performance Problems

So, why do not current transactional implementations change to support stronger consistency, such as strict serializability? Possibly, because of the generalized idea that providing strong consistency necessarily imposes unacceptable performance penalties. In fact, such may be true for today's standard architectures, which ultimately rely on the DBMS for transactional support.

When applications had simple domain models, it was common for programmers to implement a complex database query to return exactly the results sought. Generally, this meant that very few database round trips—often just one—were enough to process each unit of work requested by the client. Most of the business logic computation was embedded on the database query and handled by the DBMS. For an application with a typical 3-tier architecture, where the communication latency between the client and the server is measured in hundreds of milliseconds and the communication latency between the application server and the database takes fewer than 10ms, the time spent with the database query is almost negligible from the client's perspective.

Today, however, programmers can execute complex computations that require access to much data within a single transaction. The use of higher-level programming languages, such as OOP, facilitates and promotes data navigation instead of the creation of custom queries. This type of programming greatly increases the number of database round trips required to answer to a client's request, with negative influence on performance. If instead of one database round trip, the business transaction has to perform tens to hundreds of round trips in sequence, then the accumulated latency of all of those round trips will largely exceed the typical latency between the client and the server.

The trend towards using key-value data stores represents another factor driving the tendency towards the execution of multiple round trips per transaction. These types of DBMS

have no explicit data schema, which limits their ability to handle complex data queries. As a result, complex queries, once handled by the DBMS, are now performed by the application server, using multiple data requests in the same transaction.

Trying to reduce the number of round trips, dedicated middleware, such as ORMs, may cache data on the application server, but still they depend on the underlying database to provide the transactional semantics. Unfortunately, developers of such tools have followed suit with databases in terms of the transactional properties provided to the application programmer. In fact, the use of application tier caches may further weaken the consistency semantics of the database by inadvertently providing inconsistent reads (served from the cache) within a transaction, as discussed in Perez-Sorrosal et al. [2007].

Another way of reducing the cost of data access per request is to use an IMDG embedded in the application server, which has the potential of converting all remote calls—especially those related to reads—into local method calls only. In spite of reducing the cost of data access per request, the complexity of these data stores still leads to deep method call stacks, which affects the read performance. The performance of these calls can be greatly improved, as I will show with the experimental results in Chapter 3 and Chapter 5.

2.4 Proposed Solution

In spite of the common trend that sacrifices consistency to attain good performance, there are others who defend the importance of strong consistency [Corbett et al., 2012], and some even show that it can be achieved while attaining good performance [Shute et al., 2013]. Although, for the general case, consistency and performance may be two conflicting forces, I believe that, for a class of applications, it is possible to improve both—consistency and performance—by taking into account the following:

- The typical workloads in enterprise applications are mostly read dominated and display a very low rate of conflicts within write transactions.
- Advancements in STM technology provide in-memory transactions with exceedingly good performance, especially for read operations.
- Advancements in hardware technology provide affordable computers with large amounts of main memory, often capable of holding entire databases in main memory, and many CPU cores to deal with highly concurrent workloads.

- Communication with a DBMS tends to be several orders of magnitude slower than communication between the processor and main memory within a single computer.[4]
- There are many small- to medium-scale applications, whose typical workloads can be handled by a single powerful computer.

The solution that I propose leverages on the observations above. I propose to decouple transaction support from the DBMS and to place it in a dedicated component embedded into the application. This places the transaction control as close as possible to the location where business logic operates over shared data: The entire transactional context shares the memory with the application state. This shift reduces the cost of informing the transactional system of the operations performed, which should make the transaction's bookkeeping cheaper. Most importantly, it facilitates the preservation of a transaction's read set, which is required to uphold strong consistency.

The most important requirement of my solution is to have a transactional system that provides strong consistency. STMs are a natural fit for this purpose: They have been developed specifically to support in-memory transaction management, and they provide stronger transactional guarantees than the typical transactional system used in an enterprise application. Also, one of the core responsibilities of an STM is to intercept accesses to shared memory, which means that STM implementations tend to be concerned with providing fast memory accesses.

So, I propose to use an STM to provide transactional support directly at the application server. However, because STM works in-memory only, I will have to extend it to deal with the requirements of enterprise applications, namely to support interaction with a data storage component, and to support application clustering for cluster-wide consistent transactions.[5]

At first glance, the shift of transactional control from the DBMS to the application server can appear like a radical approach that would break with traditional enterprise application development. However, although the internal implementation of transactions in STM may differ considerably from that of transactions in a DBMS [Felber et al., 2008a], the transactional API for STM resembles that of any other transactional system (in its most fundamental

[4]This is less of a problem for embedded IMDG, but still, the greater complexity of the call stack makes it far more expensive than a simple memory access.

[5]Notice that, even though in this dissertation I will describe mechanisms to support the distributed coordination of an STM instance in each node, I dedicate my attention to the algorithmic aspects of the STM's operations. I assume a reliable underlying communication network and I do not delve into the complexities of distributed systems with regards to properties such as reliability and availability. Those are different concerns typically addressed within the scope of research on Distributed STM.

form it provides the operations to *begin*, *commit*, and *rollback* a transaction), so that the replacement should be seamless for the applications relying on the typical transactional API.

A transparent integration of this solution with the current practice in application development is important, so that adoption into existing applications is easy, and that it does not disrupt the typical process of developing a new application: This solution should appear as an evolution rather than a revolution to the application developers.

Moreover, the feasibility of the solution that I propose to implement emerges from previous positive results obtained from the application of STM to the development of enterprise applications. Next, I present some results underlying the motivation for the work developed in this dissertation. These results sustain the idea that STM can be used in practice for real-world enterprise applications providing for strong consistency and good performance, and that such applications exhibit read-dominated workloads with a low conflict rate.

2.4.1 STM in Real-World Applications

This section presents statistical data about the workload patterns of two real-world enterprise applications: FénixEDU and •IST.[6] These statistical data provide information about the typical workload of these applications, which have very high read/write ratios and a remarkably low rate of conflicts. These results were collected over an extended period of several years, with varying usage patterns, and they motivate the work in this dissertation for at least two reasons. First, they provide support to the common belief (but rarely confirmed with real data) that enterprise applications have a very high read/write ratio: In both cases, the number of write transactions are, on average, only 2% of the total number of transactions, peaking below 10% for short periods of write-intensive activity. These numbers are, actually, well below the numbers that are typically simulated by benchmarks such as TPC-W.[7] Second, these results show that strict serializability is harmless for (at least some) enterprise applications: The rate of conflicts among write transactions is almost negligible for these applications, averaging less than 0.2% of the total number of write transactions—meaning that they represent less than 0.004% of the total number of transactions.

FénixEDU is a large web application deployed as part of an academic information system for Higher Education developed at *Instituto Superior Técnico* (IST), the largest school of engineering in Lisbon, Portugal. IST is home to more than 6,000 undergraduate students

[6] Pronounced as *dot IST*.

[7] `http://www.tpc.org/tpcw/`

(BSc), 4,000 graduate students (MSc and PhD), and around 1,100 faculty members and researchers. FénixEDU supports the majority of IST's web-based functionalities for the entire school ranging from courses and academic management to administrative support, scientific support, and admissions. The functionalities it provides can be as simple as logging a summary for a class, or as complex as generating and validating timetables for the entire school.

The development of FénixEDU begun in 2002, following the at-the-time best practices of software development and engineering: It was based on a traditional web application architecture. Data was stored in a relational database (MySQL), which provided also for the transaction support required by the application's business logic. Following a rapid evolution of its feature set, with an ever increasing number of users, in late 2003 the application started to have not only performance problems, but was also facing development problems due to the complexity of the programing model, which was compounded by the pressure put on developers to make the application perform better. These problems led to the first experiments of using an STM-based architecture to develop the application. Since September 2005, the application has been running with the new architecture.

By 2011, the FénixEDU web application contained approximately 1.2 million lines of code, over 8,000 classes, of which more than 1,200 represent domain entities whose instances are manipulated transactionally and stored persistently. It had over 3,600 different web pages for user interaction. Every 5 minutes, FénixEDU logs statistics about its operation. To the best of my knowledge, FénixEDU was the first and still is, currently, the largest application to use an STM-based architecture. It is, by far, the STM-supported application for which there is the most statistical data collected.

Figure 2.2 shows the evolution in the total number of transactions processed per quarter since the last quarter of 2006. Overall, the number of transactions has been increasing, which correlates to the continuous increase in the functionalities provided by the system to its users. The fluctuations occur mostly because the users' activity is not constant over the year. For example, during vacation periods, activity drops considerably.

FénixEDU processes a daily average of 1 million transactions during work days. The peak usage of the system occurs twice a year, when the enrollment period for the following semester opens, at which time nearly 10,000 students hit the system concurrently. Figure 2.3 shows one such peak that occurred in February 2011. During the entire day of February 10, 2011, the system processed 3.7 million transactions. However, enrollments only started at

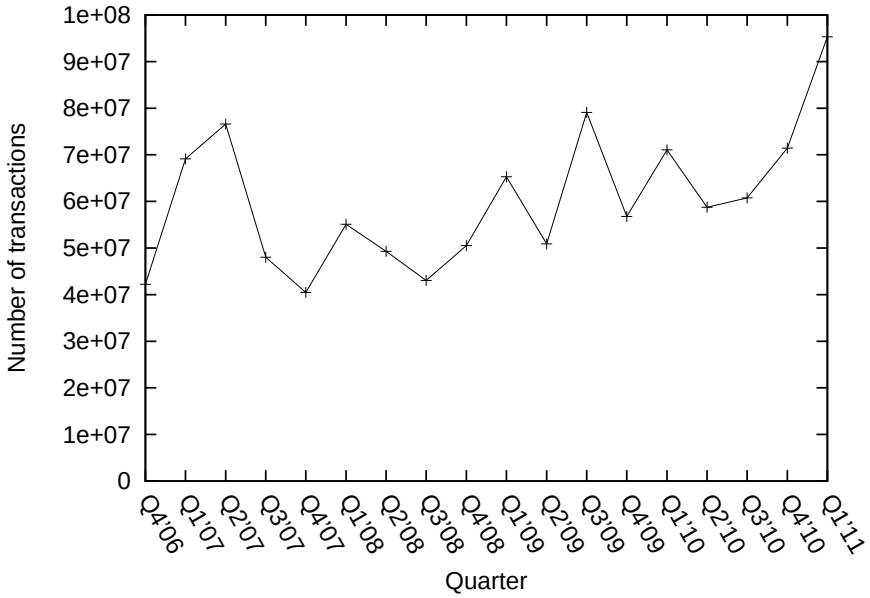

Figure 2.2: Total number of transactions per quarter in FénixEDU.

6:00 P.M. and in the following 60 minutes the system processed 1.1 million transactions, which amounts to a peak of more than 300 transactions per second.

Figure 2.4 presents the daily rate of write transactions and conflicts, also for February 2011. Notice that under normal load, over 98% of the total number of transactions processed by the system are read-only transactions, and of the remaining 2% (the write transactions) there are on average less than 0.2% restarts due to a conflict. At peak times, the rate of write transactions goes up, but still remains under 4% and the conflicts rise to about 9% (of the write transactions).

Note, however, that this throughput is not limited by the hardware, but merely reflects the demand made to the system by its users. In fact, all this is run on two machines (for fault-tolerance), each equipped with 2 quad-core CPUs and 32GB of RAM, and they are underused. Data loaded in memory usually take approximately 6GB, whereas the relational database size (measured by MySQL) is under 20GB. This shows that it is possible to deploy a real-world application running under strict serializability semantics without a negative effect on performance. I believe that these characteristics of the FénixEDU web application are not uncommon, and that, in fact, are representative of a large fraction of modern enterprise applications, for which an architecture that shifts transaction control to an STM embedded with the application server provides a very good fit.

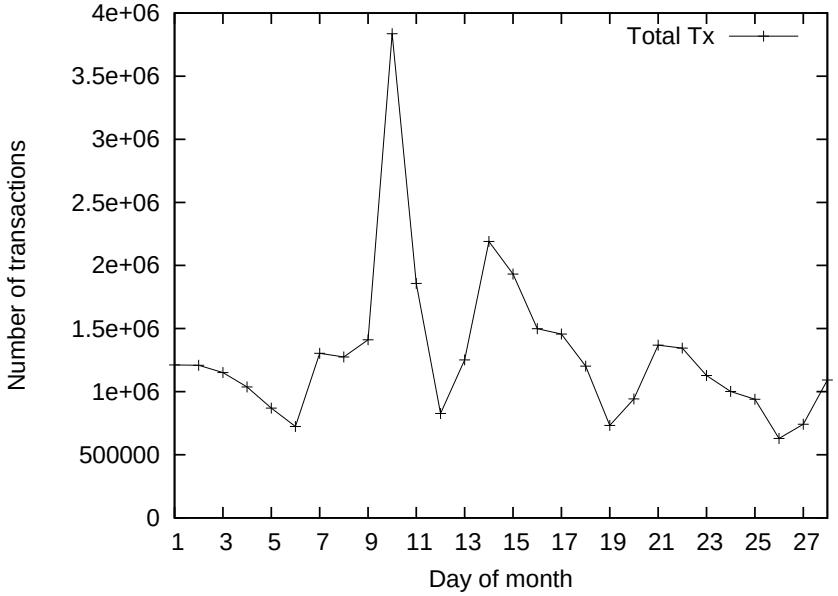

Figure 2.3: Daily number of transactions in FénixEDU during February 2011.

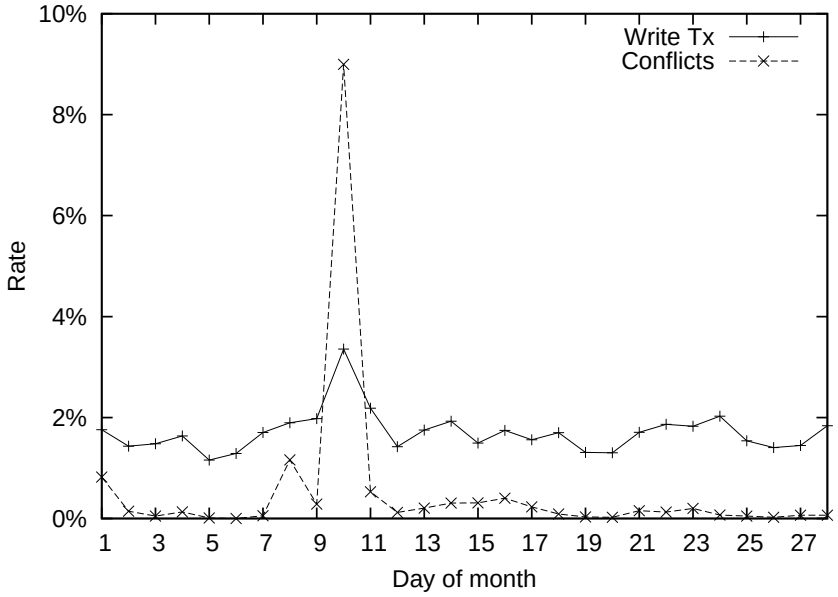

Figure 2.4: Daily rate of writes and conflicts in FénixEDU during February 2011.

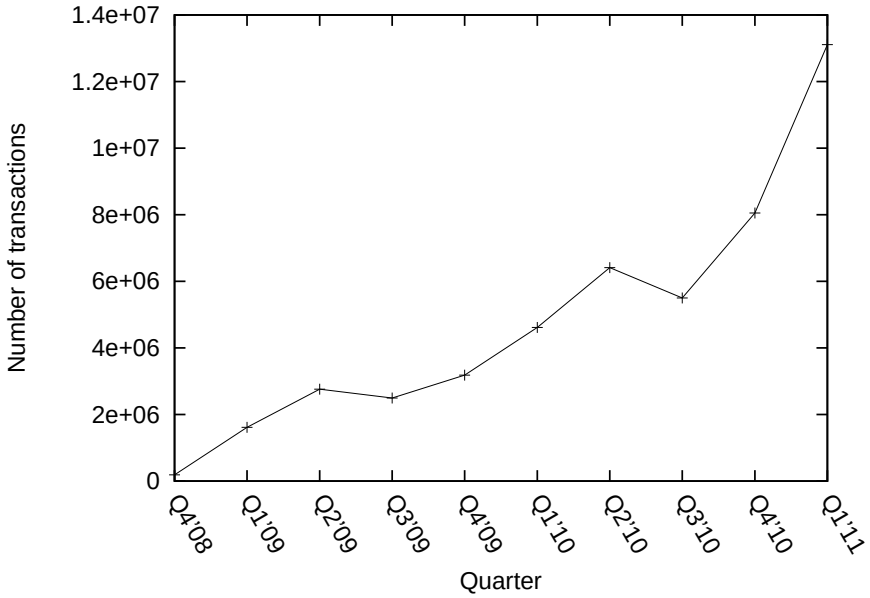

Figure 2.5: Total number of transactions per quarter in •IST.

More recently, in mid 2008, IST begun the development of another web application, named •IST. The goal of this new web application is to support many of IST's workflow processes. It includes the management of several administrative tasks, such as acquisition processes, travel authorizations, administrative staff evaluation, internal staff transfers, and document management, among others. It is dedicated to support the work of faculty members, researchers, and administrative staff. Because it does not include the students, its user base is much smaller than FénixEDU's, but it is used in a more regular fashion by its users.

Figure 2.5 shows the evolution in the total number of transactions processed per quarter, since the initial deployment of this application, which has used an STM for transaction support since the beginning. As statistical data has been collected since the initial deployment of this application, there has been a steady increase of the application's usage, since its early beta stages in the last quarter of 2008, when both few processes were supported and few users were using the application.

Since then, the application has been opened to its entire user base and has also included many more features, leading to a significantly higher number of transactions processed in recent quarters. It is expected that this growth may continue for a while as new features are added. However, it may eventually reach a plateau, because of the limited number of users.

Unlike the FénixEDU, the •IST does not have any publicly accessible web pages, other than the login page. The functionalities it provides tend to increase the number of write transactions, because many of the operations provided to the users involve the execution of workflow steps that cause changes to the application state. Figure 2.6 presents the rate of write transactions and conflicts in this application, and provides some insightful information. First, it confirms that, as expected, the percentage of write transactions is higher than in FénixEDU. However, it shows a tendency towards decreasing. In fact, the absolute number of write transactions per quarter (not shown in the plots) has been increasing, but so has the number of read-only transactions. And the latter have increased much more. I believe that this is a natural consequence of having more data available in the system, because more users access the system to check the status of the workflows in which they take part. So, in fact there is a very interesting conclusion to draw from this observation, and it reinforces the belief that, in this kind of applications, the reads largely outnumber the writes: Even when a web application is more geared towards operations that involve having its users making changes to the application's data, as in •IST, their users tend to execute many more of the read operations. In fact, some evidence suggests that the greater the number of users of an application, the greater the read/write ratio: In [Bronson et al., 2013] the authors report read/write ratios of approximately 1,000 to 1 for a system with more than 10^9 users.

In spite of •IST displaying a higher rate of writes than FénixEDU, the conflicts remain close to zero. The abnormal spike in the first quarter of 2010 is due to the execution of large migration scripts that executed scattered throughout the weeks and often conflicted with users's activities. Nevertheless, it represents a very low percentage of the write transactions.

Even though performance problems in the FénixEDU application were one of the primary reasons for developing a new architecture based on STM support, there are no quantitative measurements of the performance benefits of the new architecture for FénixEDU when comparing with the original architecture: There is only anecdotal evidence from its users that the performance of the application increased significantly once the new architecture was adopted, even though the load of the system increased steadily over time and the hardware remained the same. Unfortunately, as in most real-world scenarios, it is overly complex and expensive to reimplement any of these applications with a traditional architecture, so that the two alternatives could be compared directly.

In conclusion, the information provided by the statistical data collected in these two applications, corroborates the following ideas:

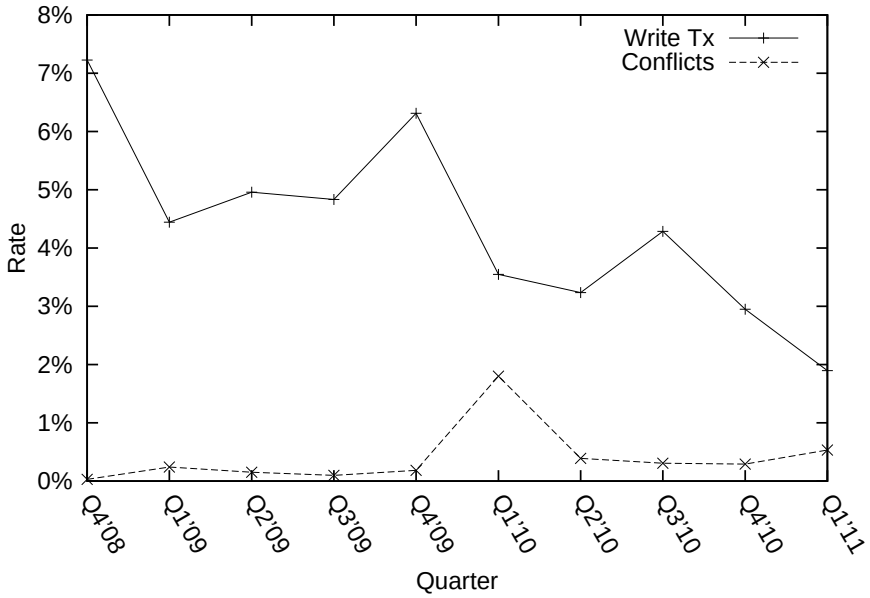

Figure 2.6: Quarterly rate of writes and conflicts in •IST.

- The majority of the workload consists of read-only transactions.[8]
- Write transactions seldom conflict. This occurs because (1) the load on the system is such that two write transactions have a low probability of intersecting in time, and (2) concurrent write operations from different users tend to access different data.
- Enforcing strict serializability for transactions does not affect the perceived performance of the applications. This is in part due to the low rate of write transactions and also because the STM in use enables read-only transaction to execute without interference.
- A single modern computer is enough to support the workload of a small- to medium-sized number of concurrent users, even under conditions where the application is naturally stressed by its users.

2.5 Summary

In this chapter, I provide the context for my work, I identify a concrete problem, and I summarize the solution that I propose to implement.

I begin with a historical perspective that focuses on how several factors drove the evo-

[8] Although not shown in the plots that I presented, there is also evidence that even write transactions are dominated by reads (cf. [Cachopo, 2007, p. 178]).

lution in the development of enterprise applications, leading to the present state, in which developers are burdened with the task of addressing consistency problems that may occur during the execution of transactional business logic. One of the causes of this problem is that traditionally consistency has been systematically weakened in favor of performance.

I argue that both performance and consistency can be improved for a class of applications, by leveraging on recent advancements in both software and hardware, and I propose the development of a middleware that provides strongly consistent transactions and that runs embedded in the application code, such that it can improve the performance of the read-heavy workloads. I back these ideas with the presentation of statistical data, collected from real-world enterprise applications.

3 TMM: A Transactional Memory Middleware for Enterprise Applications

In this dissertation, I posit that shifting transaction control from its typical location—a DBMS—to a dedicated STM-based component that is embedded into to the application is a better alternative (1) to provide strong transactional semantics, and (2) to improve performance by reducing the number of calls to the DBMS.

In this chapter I enact my proposal: I describe TMM, an STM-based middleware designed to provide efficient and strongly consistent transactional support for enterprise applications. I identify the properties of the main elements that constitute the solution I propose, and then I provide the detailed algorithms to implement this solution. Afterwards, I present an experimental evaluation of TMM, and conclude with a discussion on the related work.

3.1 The Repository Abstraction

One of the qualities of TMM is that it should be able to integrate transparently into any enterprise application, regardless of the underlying DBMS in use. This middleware needs to intercept and manage the application's operations that involve transaction control and data access, which, I assume, are operations supported ultimately by the DBMS.

To be able to design a solution that remains independent of any concrete DBMS, I introduce the *Repository* interface, which is an abstraction that represents the minimum set of operations that the DBMS must support: Such interface provides methods to store and retrieve data based on their identity, and it provides the basic operations for transaction management. The specification of this interface, however, should not be seen as a restriction to the type of enterprise applications to which TMM is applicable. The design principles of TMM are applicable regardless of the specific API used to communicate with the DBMS.[9]

[9]The applicability of TMM's design to other Repository interfaces is not mere speculation. In fact, there are already alternative implementations in the Fénix Framework (cf. `http://fenix-framework.github.io/`) that, e.g., operate with a relational database interface, and provide to the programmer higher-level data manipulation operations.

```
interface Repository {
  /* Get the object value for the given key.
     Return null if key is nonexistent. */
  Object get(Object key);
  /* Store the object value, associated with the given key.
     Overwrite previous value, if it exists. */
  void put(Object key, Object value);

  /* Begin a new transaction or throw an exception if already
     within a transaction. */
  void beginTransaction();
  /* Commit the current transaction. If commit fails, the caller
     is responsible for invoking rollbackTransaction(). */
  void commitTransaction();
  /* Abort the current transaction and restore program state. */
  void rollbackTransaction();
}
```

Listing 3.1: The `Repository` API. It represents the minimum set of operations that a TDR should provide.

For the purpose of presenting this dissertation, I shall assume that the set of operations shown in Listing 3.1 includes all the operations used by an enterprise application when interacting with the DBMS. In reality, the concrete interface provided by any specific DBMS will probably be slightly different and, inclusively, it may expose additional DBMS-specific functionality. Nevertheless, the API that I require is arguably a subset of the API that every DBMS should be able to provide: The `put(Object)` and `get(Object, Object)` operations represent the typical store and retrieve operations for a datum, and, regarding transactions, I only assume DBMS support for starting (`beginTransaction()`) and ending a transaction (`commitTransaction()` or `rollbackTransaction()`). In case a concrete application uses additional functionality, such could be decomposed into calls to the `Repository` API that I assume.

I use the term *Repository* when referring to the interface, and the term *Transactional Data Repository* (TDR) when referring to any DBMS that can provide functionality equivalent to the one described by the `Repository` interface. So, an application executes business transactions and manipulates shared data using the `Repository` interface, which some TDR then implements.

3.2 Overview of the Solution

The solution that I propose, relies on the following elements:

- **STM**. The STM runs embedded into the application, and, out-of-the-box, it provides node-local, in-memory transactions with strong consistency. It will be extended to support transactions over data that is stored externally, in the TDR.
- **STM-backed cache**. The purpose of this cache is to reduce the number of calls required to obtain data from the TDR.
- **STM synchronization protocol**. This protocol ensures that each of the node-local transactions becomes globally serialized when running multiple STM instances (one per application server).

The core idea is to shift the responsibility for transactional control: It is removed from the TDR and it is embedded into the application. To this effect, I use an STM to provide in-memory transactional support directly within the application server tier. Considering the ACID properties for the transactions that an application executes, the **Atomicity**, the **Consistency**, and the **Isolation** are supported by the STM, whereas the **Durability**, if provided, is still left as a responsibility of the TDR—I make no additional effort to ensure it, nor do I prevent it. Notice that, whether durability is provided, it will depend solely on the concrete TDR supporting it. The design that I propose applies to different types of TDRs. Some, such as IMDGs, may not always provide such guarantee.

The concrete STM that I use is the JVSTM: It has been used already in the development of enterprise applications with good performance results [Fernandes and Cachopo, 2011b], and it is one of the STMs that provides the strongest consistency: *opacity*.

I shall describe the complete middleware in three steps. In the first step, I describe how to integrate the JVSTM into the application to provide strong transactional consistency. I achieve this, simply by exchanging the application's existing Repository implementation with a JVSTM-based implementation that routes through the JVSTM the operations that the application invokes on the Repository. In turn, this new implementation will have full control of when it chooses to invoke any operation on the original Repository.

The JVSTM, however, is designed to manage in-memory–only transactions: Consequently, state is lost when the application terminates, unless there is some mechanism

in place to manage the storage of the memory contents to a TDR.

In the second step, to address the need for storing data externally, I introduce a mechanism by which data managed by the JVSTM are automatically stored to a TDR at commit time and loaded back into main memory whenever they are required. This mechanism transparently (to the application) maps between the in-memory data that is managed by the STM and the data stored in the TDR. The loading operation leverages on the fact that data is being managed by the STM (therefore, transactionally safe) to keep only a single copy of the same data in memory, thereby reducing the memory footprint of the application, when compared to middleware that ensures isolation by providing each transaction with a different copy of the data. In applications with complex business transactions that manipulate much data (such as an object-oriented application traversing a large object graph), the number of accesses to a TDR to fetch and store data can increase significantly. Without some form of caching, this high number of accesses to the data in the repository can greatly reduce overall application throughput, by largely increasing the latency of each transaction. To tackle this problem, I add an STM-aware cache of the TDR's data. This cache indexes the application data and maintains them accessible in memory for as long as possible, thereby avoiding costly calls to the TDR. Ultimately, given enough main memory, each datum is fetched only once, and the TDR will be accessed only to store new values written.

Finally, in a third step, I extend the STM with a synchronization protocol, which extends the previous approach so that it may be used in a distributed system. This protocol allows for the node-local transactions to become globally serialized. The underlying idea is that, when write transactions commit, they atomically write the data to the TDR and, also, they invalidate the written entries in other remote caches. This way, if a subsequent transaction, in another node, requests absent or invalidated data, it simply (re)loads it from the TDR.

The end result from the combination of these three steps is an STM-based middleware that can provide durable transactions and that guarantees cluster-wide strongly consistent transaction semantics. In addition, the experimental evaluation in Section 3.8 on page 77, shows that TMM is able to improve performance of typical read-dominated workloads.

Next, I introduce the JVSTM, and then describe each of the additions that I make on top of it, ending with a discussion of the consistency guarantees provided by TMM. The fully working implementation of all algorithms described in this chapter is performed within the Fénix Framework, and it is available at `https://github.com/smmf/fenix-framework` from commit with SHA-1 checksum `657ebe3ea2367368442322734ee56c7c046e9222`.

In this chapter, the solution that I propose values correctness and simplicity over performance. The design leverages on the original lock-based commit algorithm of the JVSTM and extends it as necessary. For this reason, the solution for clustering that I propose still uses an exclusive lock during the commit of write transactions. The immediate consequence is that, at any given moment, only one transaction can be executing a commit across the cluster. Although, as will be made clear with the experimental results, such solution greatly improves application performance on a single node, it does not scale well as the number of nodes in the cluster increases or the rate of write transactions increases. In the following chapters, I will revisit this issue with the aim to improve scalability, and provide an alternative solution, which does not depend on the use of locks, and enables commits to proceed in parallel.

3.3 A JVSTM Primer

The JVSTM[10] is a Java library that implements a word-based, multi-version STM that ensures strict serializability for all of its committed transactions. Actually, the JVSTM provides the even stronger correctness guarantee of opacity, which ensures additionally that even uncommitted transactions always see a consistent state of the data. The JVSTM operates only on a single *Java Virtual Machine* (JVM)—it does not support distributed transactions.

The JVSTM implements transactional memory locations [Harris and Fraser, 2003] using Versioned Boxes [Cachopo and Rito-Silva, 2006]. Each *Versioned Box* (VBox) holds a sequence of entries (VBoxBody—body for short), each containing a *value*, the corresponding *version*, and a reference to the *previous* body. Each entry in the sequence corresponds to a write made to the VBox by a successfully committed transaction. When such write transaction commits, it advances a shared version clock and uses the new clock value to tag each of the new bodies created by the commit. JVSTM's commit algorithm always maintains the sequence of bodies per box ordered, with the most recent committed value immediately accessible at the head.

Figure 3.1 on page 36 shows an example of a VBox for an integer that models a counter. In this example the counter was created (with the initial value 0) in the transaction that committed version 4. It was then incremented by the transactions that committed versions 8 and 13. From the programmer's perspective there is a single memory location of the type

[10] `http://inesc-id-esw.github.io/jvstm/`

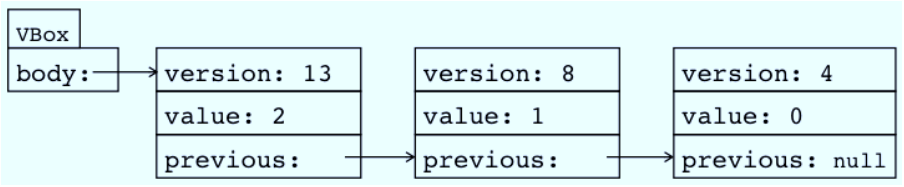

Figure 3.1: A transactional counter and its versions. From the programmer's perspective, the VBox is a transactional memory location that holds only one value at any given time.

`VBox<Integer>` that can be accessed via `get()` and `put(...)` methods.

A transaction reads values in a version that corresponds to the most recent version that existed when the transaction began. Thus, reads are always consistent and *read-only* transactions never conflict with any other transaction, being serialized at the instant they begin, as if they had executed atomically at that instant. Conversely, *write-only* transactions are serialized at commit time (when the changes are made visible atomically) and they never conflict as well. *Read-write* transactions (write transactions for short), on the other hand, may conflict with other transactions that write, and they require validation at commit time to ensure that the values read during the transaction are still consistent: A write transaction is valid to commit if the values it read have not been changed in the meanwhile by another concurrent transaction. One of the distinctive features of the JVSTM is that read-only transactions have very low overheads and they are lock-free. In fact, apart from the creation of a new transaction instance, read-only transactions run entirely wait-free.

The JVSTM follows a *redo log* model for STMs [Harris et al., 2010]: During a transaction, the JVSTM records each VBox access (either a read or a write) in the transaction's local log (in the read set or in the write set, respectively). At commit time, if the transaction's read set is valid, then its write set is *written back*: Each value in the write set is stored in a new body that is added to the corresponding VBox, thus producing a new version for each of the memory locations that changed, which will then be available to other transactions that begin afterwards; aborting a transaction, on the other hand, does not require any additional work.

The pseudocode in Listing 3.2 on page 37 summarizes the commit algorithm of write transactions (the commit of read-only transactions simply returns). The commit lock provides mutual exclusion among all committing write transactions, which ensures atomicity between validating the read set (line 6) and writing back the write set (line 8). Also, the version number is provided by a unique value obtained from a shared counter (line 7) that changes only inside

```
1  class Transaction {
2    ...
3    void commit() {
4      COMMIT_LOCK.lock();
5      try {
6        validate(); // throws a CommitException if not valid
7        int newTxnumber = getVersionClock() + 1;
8        writeBack(newTxNumber);
9        setVersionClock(newTxNumber);
10     } finally {
11       COMMIT_LOCK.unlock();
12     }
13   }
14 }
```

Listing 3.2: The commit algorithm that uses a single commit lock.

the critical region (line 9). The commit operation sets a linearization point when it updates the value of the shared counter. After that point, the changes made by the transaction are visible to other transactions that start. When a new transaction starts, it reads the version in the shared counter to know which version it will use to read values.

In the JVSTM, the read operation from a transactional location is usually very fast, for the following reasons:

- No synchronization mechanism is required to read from a VBox. The only (lock-free) synchronization point, for the entire transaction, occurs at the start of the transaction, when it reads the most recent committed version number.

- The list of versions is always ordered by decreasing version number. The commit operation keeps the list ordered, because it always writes to the head of the list a version number that is higher than the previous head version.

- The required version tends to be at the head of the list, or very near to it. If it is not at the head, then it is because after this transaction started, another transaction has already committed and written a new value to that same location (recall that a transaction always starts in the most recent version available at the time).

3.3.1 Garbage Collecting Unused Versions

To keep the list of bodies in each VBox from growing indefinitely, bodies are kept only for as long as any transaction may require them. The JVSTM ensures *forward progress*—that is, new transactions always start from the most recent version committed. As older transactions finish (in either a commit or an abort), older versions may become unused, because they become logically inaccessible, and can be removed, when no longer needed. When a write transaction commits, all values that it writes logically replace the older values, from the point of view of transactions that start henceforth. Thus, these older values can be removed when there are no transactions running in a version that may still need such values.

To identify which versions are no longer accessible and to support for GC of old versions, the JVSTM uses the *Active Transactions Record* (ATR) list [Cachopo, 2007]. This structure is a singly-linked list of records, where each record holds a reference to the *next* more recent record. Each record keeps information about a write transaction that already committed, namely the commit version and its *write set*. The latter is kept as a list of references to the bodies created during the commit. The shared version clock is actually implemented using the ATR list: The method `getVersionClock()` (in Listing 3.2 on page 37) reads a shared reference `LAST_COMMITTED_RECORD` that always points to the most recent record committed, and returns its version number; similarly, the method `setVersionClock(int)` creates a new record with the appropriate commit version, adds it to the end of the list, and updates the `LAST_COMMITTED_RECORD`.

A record also has a counter of how many *running* transactions are using that record's version to read values. When a new transaction starts, it atomically increments the counter on the most recent record. When a transaction finishes, it decrements the same counter and checks if it has reached zero. A record is considered *active* if the counter is positive or if there is an older record that is active.

When a record becomes inactive, versions produced by newer records make older versions inaccessible. At this point, older versions are removed by setting the field `previous` in each body of the *newer* record to `null`. This process is dubbed *cleaning the record*. Notice that a version v_1 is never lost, unless there is a transaction that writes a newer version v_2 to the same VBox.

Figure 3.2 on page 39 provides an example scenario in which there are three active records representing the commits of versions 9, 10, and 11. There are also three transactional

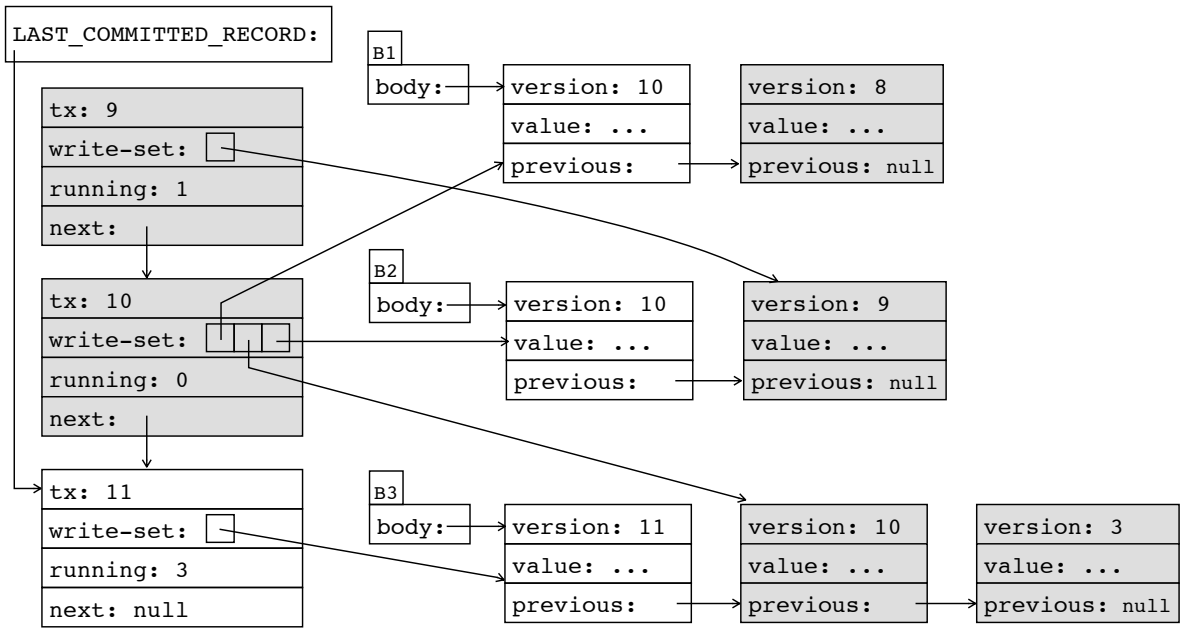

Figure 3.2: An example snapshot of the Active Transactions Record.

locations managed by VBoxes B_1, B_2, and B_3. By looking at each write set, it is possible to know that the transaction that committed version 9 wrote to B_2, transaction 10 wrote to all three locations, and transaction 11 (the most recent write transaction to commit) wrote to B_3. The shared reference for the `LAST_COMMITTED_RECORD` shows that version 11 is the most recent. Currently, there are four running transactions: One that started in version 9, and three that started in version 11. In the present state, when the only transaction running in version 9 finishes, it will decrement the `running` counter to zero. After this occurs, **record 9 is identified as inactive**, and **record 10 is cleaned**. This is accomplished by setting to `null` the `previous` field in all bodies written with version number 10. Now, record 10 also is identified as inactive and record 11 is cleaned. After both old records are cleaned, the objects shown in gray can be garbage collected by the JVM.

3.4 Integration of the JVSTM in an Enterprise Application

By definition, I assume that an enterprise application uses the interface shown in Listing 3.1 on page 32 to interact with the TDR. As I previously mentioned, the integration of the STM is transparent to the application: This is achieved simply by replacing the implementation of the `Repository` with an STM-based alternative that intercepts the interactions between the application and the TDR.

```
1  abstract class AbstractRepository implements Repository {
2
3    // The concrete underlying repository
4    Repository tdr;
5
6    AbstractRepository(Repository tdr) {
7      this.tdr = tdr;
8    }
9
10   Repository getTDR() {
11     return this.tdr;
12   }
13
14   abstract VBox vBoxFromKey(Object vboxId);
15
16   Object get(Object key) {
17     return vBoxFromKey(key).get();
18   }
19
20   void put(Object key, Object value) {
21     vBoxFromKey(key).put(value);
22   }
23
24   void beginTransaction() {
25     jvstm.Transaction.begin();
26   }
27
28   void commitTransaction() {
29     jvstm.Transaction.commit();
30   }
31
32   void rollbackTransaction() {
33     jvstm.Transaction.abort();
34   }
35 }
```

Listing 3.3: The JVSTM-based `AbstractRepository` implementation. It decorates a concrete TDR's `Repository` implementation by intercepting the invocations and using the JVSTM to implement them.

In Listing 3.3 on page 40, I present an implementation of an `AbstractRepository` that decorates a concrete `Repository` and keeps a reference to it (line 4). The implementation of the operations that control transactions (`beginTransaction()`, `commitTransaction()`, and `rollbackTransaction()`) is completely supported by the use of the JVSTM's transactions API. The implementation of the methods `get(Object)` and `put(Object, Object)` first obtains the corresponding VBox that wraps the application data (using `vBoxFromKey(Object)`), and then executes the requested operation directly on the VBox.

By using the JVSTM, it is already possible to provide a complete implementation for the `AbstractRepository`, albeit limited to applications that run only on a single JVM, and restricted to transient state. In Listing 3.4 on page 42, I present an implementation of the method `vBoxFromKey(Object)` that supports such use case. The implementation hides the (in)existence of a real TDR by keeping a table of all transactional memory locations accessed by the application. Notice that concurrent accesses to the same key obtain the values from the same VBox: The method `vBoxFromKey(Object)` is responsible for ensuring that it always returns the same instance of VBox for the same `key`.

This implementation demonstrates that it is possible to integrate the JVSTM in the enterprise application just by replacing the implementation of the operations that interact with the TDR and, by doing so, to upgrade the semantics of the application's transactions, so that they execute under strong consistency. However, because the `get/set` operations of the `TransientStateRepository` implementation operate over data that is kept in local memory only, it is possible to use this implementation to develop only a transient-state version of an enterprise application—that is, state is lost when the application terminates. To be able to interact with the TDR, I must first introduce some additional elements to the infrastructure.

3.5 Extension for Persistent Data

Considering the previous integration of the JVSTM in the application, it becomes necessary to persist changes to the underlying TDR. This operation is performed atomically during the commit of a valid transaction. Conversely, when the application starts, it may need to read existing state from the TDR, so there must also be a mechanism by which data are fetched into main memory. In the general case, during the execution of the application, not all data that are ever accessed are guaranteed to fit in memory, so the infrastructure must support

```
1  class TransientStateRepository extends AbstractRepository {
2
3    TransientStateRepository() {
4      super(null);
5    }
6
7    private static final ConcurrentHashMap<Object, VBox> repository =
8      new ConcurrentHashMap<Object, VBox>();
9
10   @Override
11   VBox vBoxFromKey(Object vboxId) {
12     VBox vbox = repository.get(vboxId);
13     if (vbox == null) {
14       vbox = new VBox();
15       VBox vboxExists = repository.putIfAbsent(vboxId, vbox);
16       if (vboxExists != null) {
17         vbox = vboxExists;
18       }
19     }
20     return vbox;
21   }
22 }
```

Listing 3.4: `TransientStateRepository` is a `Repository` implementation that does not use a concrete TDR. It keeps only an in-memory `Map` of every transactional memory location used by the application, without storing them to a TDR.

the loading and offloading of data between main memory and the secondary storage provided by the TDR. This mechanism must also ensure that it preserves the transactional semantics in case of a sudden application crash.

One of the goals of the architecture I propose is to avoid repetitive calls to read data from the TDR. I achieve this by keeping data accessible in memory for as long as possible in a transactionally safe cache, and resorting to interaction with the TDR only when either absent values are requested or changes are committed.

In this section, I first describe the mapping between the in-memory transactional data and their representation in the TDR. Then, I present the changes to the JVSTM's commit algorithm to include the writing of data to the TDR. Next, I show how VBoxes are maintained in main memory, in a way that ensures their referential uniqueness, thus sharing them among concurrent transactions. Finally, I describe what happens when the application writes data for new keys (requiring the allocation of new VBoxes) or attempts to get values for keys that are not in main memory yet (requiring the reload of VBoxBodies from the TDR).

3.5.1 Data Mapping

Whenever application data are stored externally, such as in a TDR, there must be a mechanism through which the application can reference its data for later retrieval. From the perspective of an application using a TDR, the identification of its data corresponds to the *key* argument used in the `get` and `put` operations defined by the `Repository` interface. The JVSTM uses VBoxes to wrap the data manipulated by the application, and from the perspective of the JVSTM, the object reference to the VBox itself is the actual identifier of the memory location being managed. To link the two identification mechanisms, I extend the VBox with an attribute (`vboxId`) to hold the key used by the application to identify the memory location that the VBox wraps.

Whereas the application is only concerned with a single value for each key, the JVSTM actually needs to be able to store multiple values for the same key: Each value written to the same key is kept in a unique VBoxBody that is created by the commit of the transaction that changed the memory location identified by the given key. So, this requires a mechanism to uniquely identify the values contained in the bodies and map between their representation in memory and their representation in the TDR. Recall that each body contains information about its version, the value written, and a reference to the previous body. The key used to

```
1  class RepositoryEntry {
2    public final int version;
3    public final int previousVersion;
4    public final Object value;
5
6    RepositoryEntry(int version, int previousVersion, Object value) {
7      this.version = version;
8      this.previousVersion = previousVersion;
9      this.value = value;
10   }
11 }
```

Listing 3.5: A `RepositoryEntry` stores the contents of a VBoxBody in the TDR.

store each body in the repository (*versionedKey*) is generated by the function *makeVersionedKey*:

versionedKey = *makeVersionedKey*(*vboxId*, *version*)

The entry corresponding to *versionedKey* holds the information of a body: the value, the committed version and the version of the previous body. Listing 3.5 shows this structure, which effectively represents in the TDR the linked list of bodies from the VBox, as shown in Figure 3.1 on page 36. The function *makeVersionedKey* must be injective, that is it must return a unique *versionedKey* value for each pair (*vboxId*, *version*). The argument *vboxId* is the identifier of the VBox, which was assigned by the application to the transactional memory location. It is the one explicitly used by the application when invoking the get/put operations in the `Repository` interface. When creating a new entry, the argument *version* corresponds to the transaction's commit version. When reading a body into a VBox, however, the version to retrieve depends on the current transaction: The JVSTM needs to provide to the application code the highest version that is less than or equal to the current transaction's read version. This poses a problem, because whereas the JVSTM knows the read version of its current transaction, it does not know in which version was the VBox last written. For this reason, the latest version of each VBox is kept in the *headKey* that is generated based only on the identity of the transactional memory location:

headKey = *makeHeadKey*(*vboxId*)

This enables the JVSTM to calculate the *headKey* and obtain the most recent entry without knowing its version beforehand. Then, if it needs a version older than the one obtained, it simply iterates to the previous entry, using the information contained in this

entry about the `previousVersion`. The intersection of the codomain of each of these functions should be an empty set. This ensures that *versionedKeys* and *headKeys* never intersect, which would cause inconsistencies in the data stored.

Using the mapping strategy I described, the entries stored in the TDR, in *versioned keys*, never change. The only entries that may change are the ones that correspond to the *head keys*, which correspond to the most recent commit of each VBox. Any operation that updates such entries in the TDR must preserve the following invariant: **Whenever the entry for a head key is replaced, the entry being overwritten must already exist in its corresponding versioned key**. This invariant is required to ensure correct execution when, concurrently with a lookup, some transaction commits a newer value. By definition, the newer entry does not contain the value of interest for the transaction doing a lookup, which must necessarily be running in an older version than the one being concurrently committed. Therefore, if the head key lookup already sees the new value it must be able to iterate to the next versioned key, which is guaranteed to exist.

As a result, conceptually, any application data that is publicized to the TDR is never overwritten, which ensures that any transaction will always be able to access a value for any given version. In spite of that, data could be removed from the TDR, if it is determined to be no longer necessary: This is akin to garbage collecting old versions from memory, as described in Section 3.3.1 on page 38.

3.5.2 Persisting Changes

The JVSTM already deals with all the in-memory transactional support, but it does not provide any form of data storage. After having established the definition of the mapping between the values in memory and those stored in the TDR, I can now present how to change the commit operation so that the writes are also propagated to the TDR.

Listing 3.6 on page 46 presents the new commit algorithm for write transactions, which is structurally identical to the original commit, containing the same three steps: (1) validate the read set, (2) write back the write set, and (3) make the commit visible to other transactions. I extend the JVSTM's original commit to include the `commitToRepository(int)` operation (in line 8), which sends the changes to the TDR.

The order of invocation of the `commitToRepository(int)` operation within the commit procedure is critical to ensure the required transactional behavior: It is performed after the

```
1  class Transaction {
2    ...
3    void commit() {
4      COMMIT_LOCK.lock();
5      try {
6        validate();
7        int newTxnumber = getVersionClock() + 1;
8        commitToRepository(newTxNumber);
9        writeBack(newTxNumber);
10       setVersionClock(newTxNumber);
11     } finally {
12       COMMIT_LOCK.unlock();
13     }
14   }
15
16   void commitToRepository(int txNumber) {
17     Repository tdr = getRepository().getTDR();
18
19     boolean success = false;
20     try {
21       tdr.beginTransaction();
22       putWriteSet(tdr, txNumber);
23       updateTxNumber(tdr, txNumber);
24       success = true;
25     } catch (Exception e) {
26       throw new CommitException(e);
27     } finally {
28       try {
29         if (success) {
30           tdr.commitTransaction();
31         } else {
32           tdr.rollbackTransaction();
33         }
34       } catch (Exception e) {
35         throw new CommitException(e);
36       }
37     }
38   }
39 }
```

Listing 3.6: The commit operation of the JVSTM is extended to write the changes to the TDR, after validating the transaction in memory. The code in gray color is the same from Listing 3.2 on page 37.

transaction has been validated, and before making the changes visible in memory. The former ensures that the state to persist is consistent, and the latter ensures that a possible failure in the `commitToRepository(int)` operation occurs before modifying any shared memory.

Listing 3.7 on page 48 shows the methods `putWriteSet(Repository, int)` and `updateTxNumber(Repository, int)`, which perform the necessary updates to the TDR. The method `putWriteSet(Repository, int)` iterates the write set entries and for each one:

- If the VBox being updated has an entry for the head key, then it stores the latest `RepositoryEntry` in its versioned key (line 13).
- Creates a new `RepositoryEntry` to represent the changes being committed (line 17): The value of the previous version depends on whether a previous version exists (the value -1 represents that there is no previous version).
- Puts the new entry in the head key (line 19), overwriting any previous value for that key.

The method `updateTxNumber(Repository, int)` simply overwrites a reserved key that holds the version of the last committed transaction. This information is strictly necessary only when bootstrapping the first node in the cluster. Part of the node's startup includes reading the version from this reserved key to initialize its local clock. Other nodes can obtain this information at startup from any existing node already in the cluster. After startup, existing nodes update their in-memory clock when processing a commit (either local or remote).

Because all writing to the TDR is performed within the commit lock, there will be at most only one write transaction running on the TDR at any given time and, thus, it will never conflict with another. At this point, the only requirement placed upon the TDR's transactional system is that it provides atomicity: The `commitToRepository(int)` executes a single TDR transaction that either fails, therefore leaving the TDR unchanged, or it succeeds, therefore having atomically performed the writes.

Atomicity is necessary to account for any catastrophic crash of the application server during the write to the TDR. If this happens, then when the application server restarts it will see a consistent state, either before or after the current commit operation. Also, after

```
class Transaction {
  ...
  void putWriteSet(Repository tdr, int txNumber) {
    for (WSEntry<VBox, Object> wsEntry : this.writeSet) {
      Object vboxId = wsEntry.getVBox().getId();
      Object newValue = wsEntry.getValue();

      RepositoryEntry current = tdr.get(makeHeadKey(vboxId));

      int previousVersion = -1;
      if (current != null) {
        previousVersion = current.version;
        tdr.put(makeVersionedKey(vboxId, current.version),
                current);
      }

      RepositoryEntry newEntry =
        new RepositoryEntry(txNumber, previousVersion, newValue);
      tdr.put(makeHeadKey(vboxId), newEntry);
    }
  }

  void updateTxNumber(Repository tdr, int txNumber) {
    tdr.put(KEY_LAST_COMMITTED_VERSION, txNumber);
  }

}
```

Listing 3.7: Implementation of the operations that write to the concrete TDR.

persisting changes, the remainder of the commit algorithm can only fail for catastrophic reasons (such as running out of memory), in which case the application server's restart will certainly see the current transaction as committed when reading the state from the TDR. There is no additional dependency on the transactional semantics of the TDR for any application transaction to succeed, because the application will never directly read state from the TDR, and the JVSTM ensures that new writes are ignored until their in-memory commit is published.

If the write to the TDR fails for some catastrophic reason (such as loosing a connection), then the transaction is trivially aborted in memory.[11] If, on the other hand, the catastrophic failure occurs when writing back to main memory (thus, after having committed changes to the TDR), when the application restarts it will consider that the transaction has been committed.

3.5.3 VBox Caching

There is one cache instance on each application server, shared by all threads. The cache indexes VBoxes by their identifier and it serves two main purposes:

- To keep the VBoxes available in memory for as long as possible, thereby to avoid hitting the TDR with repeated requests for the same data.
- To ensure that there is only one instance of the same VBox loaded in memory—the references to the VBoxes are shared by all threads, thereby enabling the JVSTM to detect conflicts.

So, a key feature of using the cache is that application transactions will only access the TDR if the requested data is not cached yet. The goal is that, once loaded from the TDR, a VBox will remain in cache, as long as possible—that is until the memory allocation requirements of the application cause the garbage collector to remove unused entries to make space for other needed data. Only in that case may the entry be removed from the cache and garbage collected (provided that it is not still in use by any running transaction). One of the consequences of using this approach is that for applications whose working data set fits in the cache, after warming up, **the application server will never access the TDR for**

[11] In fact, if the TDR becomes unavailable, the typical enterprise application becomes unusable. Using the JVSTM, the application may still continue to provide read-only transactions, as long as they access only data already loaded into main memory.

reading an entry. Write transactions, however, will still use the TDR at STM-commit time to store the entries written and to update control data used by the STM.

The `VBoxCache` implementation is shown in Listing 3.8 on page 51. It provides the typical *cache* and *lookup* operations taking, respectively, a VBox to store and a VBox identifier to lookup. TMM automatically handles the interaction with the cache, which is transparent to the application.

To ensure that there is at most one shared VBox instance per memory location, the `cache(VBox)` operation does not override an existing entry, when there is already another VBox cached with the same `vboxId`: This operation always returns a reference to the VBox in the cache with the `vboxId` of the VBox being cached—either the one being cached, if it is the first for the given identifier, or the existing instance, otherwise—so that concurrent attempts to cache a VBox with the same identifier, all get back the same reference. The `lookup(Object)` operation simply returns the entry for the requested VBox, or `null` if it does not exist in the cache.

The implementation of the cache follows the Identity Map architectural pattern [Fowler, 2002]: Every time the JVSTM needs to operate on a VBox, it first looks it up in the cache. Whenever creating a new VBox instance, it immediately caches a reference to it, **and it always operates on the returned reference** to ensure that it is operating on the unique instance reference.

Each cache entry is a Java `SoftReference`[12] to the actual VBox, so that the cache entry is maintained while the VBox is strongly referenced.[13] Whenever the VBox becomes softly referenced, the VBox may be garbage collected and eventually its entry will be removed from the cache.

3.5.4 Allocating VBoxes

The application's code accesses shared state through the `Repository` API, using keys to identify data. As I have shown, in Listing 3.3 on page 40, when intercepting an access (either a get or a put), the first thing to do is to obtain, via the method `vBoxFromKey(Object)`, the corresponding VBox that wraps the datum identified by the `key` and, then, to invoke through the JVSTM's VBox API the corresponding (get or put) operation.

[12] `http://docs.oracle.com/javase/7/docs/api/java/lang/ref/SoftReference.html`
[13] `http://docs.oracle.com/javase/7/docs/api/java/lang/ref/package-summary.html#reachability`

```
1  class VBoxCache {
2    ...
3    final ConcurrentHashMap<Object, SoftReference<VBox>> cache = new ...;
4
5    VBox cache(VBox vbox) {
6      return cacheNewEntry(new SoftReference<VBox>(vbox), vbox);
7    }
8
9    // Cache a VBox and return the canonical reference to it
10   VBox cacheNewEntry(SoftReference<VBox> newEntry, VBox vbox) {
11     SoftReference<VBox> entryInCache =
12           putIfAbsent(this.cache, vbox.getId(), newEntry);
13
14     if (entryInCache == newEntry) { return vbox; }
15
16     VBox vboxInCache = entryInCache.get();
17     if (vboxInCache != null) { return vboxInCache; }
18
19     // current entry was GCed already. Remove and retry
20     this.cache.remove(vBox.getId(), entryInCache);
21     return cacheNewEntry(newEntry, vbox);
22   }
23
24   VBox lookup(Object vboxId) {
25     SoftReference<VBox> entry = this.cache.get(vboxId);
26     if (entry != null) {
27       VBox result = entry.get();
28       if (result != null) {
29         return result;
30       } else {
31         this.cache.remove(vboxId, entry);
32         return null;
33       }
34     } else {
35       return null;
36     }
37   }
38
39   static <K, V> V putIfAbsent(ConcurrentHashMap<K, V> map,
40                               K key, V value) {
41     V oldValue = map.putIfAbsent(key, value);
42     return ((oldValue == null) ? value : oldValue);
43   }
44 }
```

Listing 3.8: The cache of VBoxes.

```
class RepositoryWrapper extends AbstractRepository {

  static final VBoxCache cache = new VBoxCache();

  RepositoryWrapper(Repository tdr) {
    super(tdr);
  }

  @Override
  VBox vBoxFromKey(Object vboxId) {
    VBox vbox = cache.lookup(vboxId);
    if (vbox == null) {
      vbox = allocateVBox(vboxId);
    }
    return vbox;
  }

  static VBox allocateVBox(Object vboxId) {
    return cache.cache(new VBox(vboxId, VBox.NOT_LOADED_BODY));
  }
}

class VBox {
  ...
  static final VBoxBody NOT_LOADED_BODY =
    makeNewBody(NOT_LOADED_VALUE, 0, null);

  static VBoxBody makeNewBody(Object value, int version,
                              VBoxBody previous) {
    return new VBoxBody(value, version, previous);
  }
}
```

Listing 3.9: The `RepositoryWrapper`, which wraps the TDR's `Repository` implementation.

In Listing 3.9 on page 52, I provide the final `RepositoryWrapper` that integrates the VBox cache from the previous section. The concrete method `vBoxFromKey(Object)` first looks up the requested VBox from the cache: The expected common case is for this method to find the VBox in the cache and to return it. However, when the application refers an existing key whose VBox is not currently cached, or when the application stores a value in a key for the first time, the lookup operation simply returns `null`. At this point, the infrastructure must create a VBox to represent the transactional memory location being accessed, which is accomplished by the method `allocateVBox(Object)`: This method creates a new VBox instance, caches it, and returns the canonical reference to the VBox, which is obtained from the method `cache(VBox)`. This procedure ensures that if, concurrently, two transactions attempt to allocate a VBox for the same key, they obtain the same VBox instance. Also, references to additional VBoxes that may be created for the same key, due to concurrent allocation requests, will be immediately lost and become garbage collectable.

All VBoxes are instantiated with a special VBoxBody—the `VBox.NOT_LOADED_BODY`—which represents that the contents of the VBox are unknown. If the application's request is to write to the VBox, then the value will be assigned during the commit, when a new VBoxBody with the correct commit version and the correct value will be inserted at the head of the list of bodies. If the application's request is to read from the VBox, then the middleware will find that the value is missing and load it from the TDR, before returning it to the application.

3.5.5 Loading the Contents of a VBox

In the JVSTM, getting the value of a VBox is a wait-free operation that consists in iterating the bodies of the VBox, starting from the most recent, until it finds a body whose version is less than or equal to the current transaction's read version. This operation is guaranteed to succeed, because all accessible versions are kept in memory. When reading a value from a VBox, the current transaction will request a given body from the VBox using the method `getBody(int)`, whose implementation is shown in Listing 3.10 on page 54. The `get()` operation delegates the request to the current transaction. In turn, the transaction—who knows its read version—requests the correct body from the VBox and then returns the value. To improve readability, I have elided other tasks that the method `getBoxValue(VBox)` may perform, such as registering the read in the read set, which are not relevant to the current matter.

Now, getting a value from a VBox must take into account the possibility that the required

```
1  class Transaction {
2    ...
3    Object getBoxValue(VBox vbox) {
4      ...
5      VBoxBody body = vbox.getBody(getReadVersion());
6      ...
7      return body.value;
8    }
9  }
10
11 class VBox {
12   ...
13   Object get() {
14     return Transaction.current().getBoxValue(this);
15   }
16
17   VBoxBody getBody(int maxVersion) {
18     VBoxBody current = this.body;
19
20     while (current.version > maxVersion) {
21       current = current.previous;
22     }
23
24     return current;
25   }
26 }
```

Listing 3.10: The original `get()` method of the VBox. The iteration in `getBody(int)` is bounded and guaranteed to find a suitable VBoxBody that contains the requested value.

body may be absent from main memory. In Listing 3.11 on page 57, I show the revised portion of the operation `getBoxValue(VBox)`, which accounts for the possibility of finding a `NOT_LOADED_BODY`. The special marker `NOT_LOADED_BODY` is always the last element in the chain of bodies, and it represents that there are no more older versions loaded in memory for that VBox.

When a transaction requests a value from a VBox and detects that the required version is not loaded, it triggers the `reloadVBox(int)` operation to read from the TDR the necessary data into main memory. This operation (line 19 of Listing 3.11 on page 57) is performed while holding the VBox's monitor: It fetches from the TDR **all** versions between the oldest loaded version down to the version that is less than or equal to the version that should be seen by the current transaction (line 20). The resulting list of bodies is then placed at the tail of the list of VBoxBodies that the VBox currently holds (line 21). The auxiliary operations used by the reload are shown in Listing 3.12 on page 58 and Listing 3.13 on page 59:

- `getOldestBody()` searches the VBox for the oldest VBoxBody that is not the `NOT_LOADED_BODY`. It returns `null` when the only body in the VBox is the `NOT_LOADED_BODY`.

- `replaceTail()` replaces the existing reference to the `NOT_LOADED_BODY` with the given reference to the new tail, which was obtained from `loadVersionsInRange(int)`. The reference to replace may originate from either the VBox or one of its VBoxBodies, so this method first determines the type of tail replacement to perform (selecting either `replaceTailInVBox(VBoxBody)` or `replaceTailInBody(VBoxBody)`).

- `getStartKey()` determines the first key to use when loading versions: It is either the key of the oldest body loaded in memory, or the *head key* for the VBox if there is no body loaded.

- `loadVersionsInRange(int)` returns the new tail of bodies that should replace the current `NOT_LOADED_BODY`. This method reads from the underlying TDR a sequence of entries starting with the `startKey` (line 5) until either it finds a `RepositoryEntry` whose version satisfies the `requiredVersion` (line 15) or no more entries exist (line 17), in which case is creates a mock `RepositoryEntry` to represent the missing value. In the latter case the application requesting this VBox's value will obtain `null` to represent a missing entry in the TDR for the datum being looked up. Finally, this

method creates the linked list of VBoxBodies that it returns. Notice how the iteration through the list of repository entries starts at the lowest version, due to the use of `addFirst(RepositoryEntry)` when adding entries to the list.

By the end of the reload operation, a VBoxBody that satisfies the transaction's required read version must exist in memory. Such body is obtained simply by invoking again `getBody(int)` (line 10 of Listing 3.11 on page 57): This time there is no need to test again for the `NOT_LOADED_BODY`.

Also very important is that, at any moment, there are never any VBoxBodies missing from the list of VBoxBodies between the one at the head (the VBox's current reference to `body`) and the oldest version loaded. Upholding this invariant is crucial to ensure that no transaction ever reads a wrong version from a VBox: Consider a transaction running in version N that reads from a VBox that has never been cached locally; if it were to reload only the version M needed by N ($M \leq N$), then another transaction running in version P, where $P > N$, could also use version M from this VBox; yet, this would be incorrect, if there was already a version O committed for this VBox such that $P > O > N$.

There are two more noteworthy considerations regarding the loading procedure. One is that changes to the bodies of a VBox must occur in mutual exclusion with other attempts to change any bodies of the same VBox (cf. Section 3.6.5 on page 71). For this reason the `reload(int)` executes holding the VBox's monitor. Otherwise, a concurrent reload for the same VBox could cause some tail of bodies, not containing the required version, to replace another. The other consideration regards what happens if the application code attempts to lookup the value of a nonexistent key from the TDR—a key for which there is no value assigned for the required version. In this case, the application is typically expecting to obtain `null` from the following invocation on the `Repository` interface:

```
get(SOME_NONEXISTENT_KEY)
```

In my approach, I retain this same behavior: When accessing the key `SOME_NONEXISTENT_KEY`, first there will be a VBox allocated for this location, as described in Section 3.5.4 on page 50. Then, when the `reloadVBox(int)` occurs, it will either fail to find any information on the TDR regarding this key (line 8 of Listing 3.12 on page 58), or it will not find an entry for the required version (line 17). In both cases, the operation will set `null` in the body corresponding to version 0. This does not constitute a write to the TDR, but will effectively represent that the current transaction (T_a) read this key, by adding it to

```
1  class Transaction {
2    ...
3    Object getBoxValue(VBox vbox) {
4      ...
5      VBoxBody body = vbox.getBody(getReadVersion());
6      if (body == VBox.NOT_LOADED_BODY) {
7        synchronized (vbox) {
8          vbox.reloadVBox(getReadVersion());
9        }
10       body = vbox.getBody(getReadVersion());
11     }
12     ...
13     return body.value;
14   }
15 }
16
17 class VBox {
18   ...
19   void reloadVBox(int requiredVersion) {
20     VBoxBody tail = loadVersionsInRange(requiredVersion);
21     replaceTail(tail);
22   }
23 }
```

Listing 3.11: The extended `getBoxValue(VBox)`. It accounts for the need to reload the contents from the TDR.

```
1  class VBox {
2    ...
3    VBoxBody loadVersionsInRange(int requiredVersion) {
4      Repository tdr = getRepository().getTDR();
5      RepositoryEntry current = tdr.get(getStartKey());
6      LinkedList<RepositoryEntry> entries = new LinkedList();
7
8      if (current == null) {
9        // special case: the VBox never existed in the TDR
10       entries.addFirst(new RepositoryEntry(0, -1, null));
11     } else {
12       while (true) {
13         entries.addFirst(current);
14
15         if (current.version <= requiredVersion) {
16           break;
17         } else if (current.previousVersion == -1) {
18           // special case: the required version has no value
19           entries.addFirst(new RepositoryEntry(0, -1, null));
20           break;
21         }
22
23         current = tdr.get(makeVersionedKey(getId(),
24                                            current.previousVersion));
25       }
26     }
27
28     // build the new tail
29     VBoxBody bodies = NOT_LOADED_BODY;
30     for (RepositoryEntry entry : entries) {
31       bodies = VBox.makeNewBody(entry.value, entry.version, bodies);
32     }
33     return bodies;
34   }
35
36   void replaceTail(VBoxBody tail) {
37     if (this.body == NOT_LOADED_BODY) {
38       replaceTailInVBox(tail);
39     } else {
40       replaceTailInBody(tail);
41     }
42   }
43 }
```

Listing 3.12: Auxiliary operations for the reload of a VBox. Both `loadVersionsInRange(int)` and `replaceTail(VBoxBody)` run while holding the VBox's monitor exclusively.

```
1  class VBox {
2    ...
3    Object getStartKey() {
4      VBoxBody oldestBody = getOldestBody();
5      if (oldestBody == null) {
6        return makeHeadKey(getId());
7      } else {
8        return makeVersionedKey(getId(), oldest.version);
9      }
10   }
11
12   VBoxBody getOldestBody() {
13     VBoxBody current = this.body;
14     if (current == NOT_LOADED_BODY) {
15       return null;
16     } else {
17       VBoxBody oldest = current;
18       while (oldest.previous != NOT_LOADED_BODY) {
19         oldest = oldest.previous;
20       }
21       return oldest;
22     }
23   }
24
25   void replaceTailInVBox(VBoxBody tail) {
26     this.body = tail;
27   }
28
29   void replaceTailInBody(VBoxBody tail) {
30     VBoxBody oldestBody = getOldestBody();
31     oldestBody.previous = tail.previous;
32   }
33 }
```

Listing 3.13: Auxiliary operations for the reload of a VBox (cont.).

the read set, and it will conflict with any other transaction (T_b) that may concurrently write to this key (if T_b is serialized before T_a and the latter is a read-write transaction). So, the strong consistency guarantees apply, even when reading a nonexistent entry that is being concurrently created.

3.6 Extension for Clustered Environment

Thus far, I have described all that is needed for TMM to support an enterprise application running in a single-node configuration. This decomposition of the solution is intentional. One reason is that it simplifies the exposition. The other reason is because multicore computers with large amounts of available memory are becoming mainstream and they are a natural target hardware for applications developed with the architecture that I describe: The highly parallel workloads benefit from the increasing number of cores, and having large amounts of main memory enables the STM to keep most, if not all, of the VBoxes in cache, thus reducing the number of calls to the repository to fetch data.

Nevertheless, even though a single computer built with this commodity hardware might be enough to run the entire application server with very good performance, there are several other reasons for deploying an application in a clustered environment, such as support for fault tolerance, data replication, scalability, load balancing, and live hardware upgrades. In this section, I show how to extend the middleware to support more than one application server, each running an instance of TMM, while continuing to ensure strong consistency for the transactions. Next, I summarize the changes that must be made to the JVSTM, and then describe how to implement them.

3.6.1 Overview of the Required Changes

The synchronization protocol that I propose leverages on the TDR to make shared application state available on every node, as needed. If the concrete TDR happens to use a distributed topology, I make the arguably legitimate assumption that such TDR should provide the mechanisms required to handle its own data synchronization among multiple nodes. As such, I do not provide any additional state transfer mechanism specifically designed for the application data that is stored in the TDR. I only require that, after any data is committed to the TDR, it will be made available to transactions that start on any application node, henceforth. Consequently, in case of a distributed TDR, its commit operation needs to ensure

that after returning, future transactions will be able to see the committed data, should they request it, regardless of the node in which they begin.

Under the assumption that, once stored in the TDR, application data will be available in any node when requested, the changes required at the STM level to support multiple application servers are:

- To extend the JVSTM's version clock to work globally across all nodes. This global clock will advance upon a successful commit, and provide a unique version number for each commit across all nodes.

- To acquire a cluster-wide commit lock during the commit of a write transaction.

- As part of a valid commit, send a message to the other nodes identifying the VBoxes that have been changed by this commit (not their values), so that each local cache can be invalidated. Total order of the commit messages is ensured, because they may be sent only within the global commit lock. Notice that because VBoxes are transactionally safe, those in the local cache of the committing node are automatically updated during the STM's standard commit and, thus do not get invalidated, as their remote counterparts do.

- When committing a write transaction, after obtaining the global commit lock, ensure that any remote commit messages, regarding versions earlier than the one now attempting to commit, have been duly processed before validating the committing transaction. This may lead to invalidation of local cache entries due to remote writes, which, in turn, may lead to the invalidation of the committing write transaction, if it has read any values that changed meanwhile. This behavior is important to ensure global serialization of write transactions.

- When starting a new transaction, ensure that any remote commit messages that may be pending to process at the moment the transaction attempts to start, have been duly processed before obtaining a read version for this new transaction. This ensures that a transaction begins the in the most recent version known to the node, thereby improving its chances of a successful commit without conflicting with others.

3.6.2 Synchronization Mechanisms

To implement the mechanisms that synchronize each JVSTM instance with the others in the cluster, I use some of the distributed data structures provided by Hazelcast,[14] which, among other features, provides high-level distributed group communication primitives. Namely, I use:

- A *distributed atomic number* to implement a **global lock**. When this lock is free, the (positive) value of the atomic number represents the version of the most recently committed transaction. When the lock is held, its value is negative.
- A *distributed topic* to **broadcast the remote commit messages**. This is a publish/subscribe messaging system that ensures ordered delivery of the messages sent among all members of the cluster. Each remote commit message is sent to this topic while holding the global commit lock, thus ensuring that messages are always received in the order in which the commits occur.
- Another *distributed atomic number* to implement **a shared counter used for assigning unique identifiers to each member of the cluster**. During startup each node atomically reads and increments this counter to obtain a unique number. Each node keeps the counter's value in main memory only and sends it in every remote commit message to identify the node that performed the commit. This way, the subscribers of the distributed topic can easily ignore the commit messages that have originated on their own node.

3.6.3 Committing a Transaction

When using the JVSTM, state changes occur only during the commit of write transactions. These commits are already ordered within one application server because of JVSTM's commit lock. Now, they also need to be globally ordered among the commits from all servers, which is achieved by using a (cluster-wide) global lock. Additionally, the commit operation communicates a *change log* that is then used by the other servers to update their local state. Each change log associates the commit version with the identification of the VBoxes that have changed in that version. The change log is similar to a write set, except that it does not actually contain the values written, only the identification of the VBoxes written.

[14] `http://www.hazelcast.com/`

```
1  class Transaction {
2    ...
3    void commit() {
4      COMMIT_LOCK.lock();
5      try {
6        applyAllRemoteCommits();
7        validate();
8        int newTxNumber = performGlobalCommit();
9        setVersionClock(newTxNumber);
10     } finally {
11       COMMIT_LOCK.unlock();
12     }
13   }
14 }
```

Listing 3.14: The clustered commit algorithm of the JVSTM. It globally orders commits across the cluster using a global lock, which is acquired only after a successful best-effort attempt to validate the transaction locally. The code in gray color is the same from Listing 3.6 on page 46.

Listing 3.14 shows the entry point for the extended commit algorithm, and Listing 3.15 on page 64 shows the additional operations that it uses. The following are the steps required to successfully commit a write transaction:

1. Acquire the local commit lock.
2. Locally apply pending remote commits, if any.
3. Perform read set validation.
4. Acquire the global lock.
5. Locally apply outstanding remote commits, if any.
6. Perform read set validation (only needed if an outstanding remote commit was applied after acquiring the global lock).
7. Commit the changes to the TDR.
8. Broadcast the remote commit.
9. Apply the changes to local VBoxes.
10. Release the global lock.

```
1  class Transaction {
2    ...
3    void applyAllRemoteCommits() {
4      RemoteCommit rc;
5      while ((rc=ClusterUtils.getRemoteCommitMessages().poll()) != null) {
6        rc.apply();
7      }
8    }
9
10   int performGlobalCommit() {
11     int globalClock = ClusterUtils.globalLock();
12     int nextVersion = globalClock + 1;
13     boolean commitSuccess = false;
14     try {
15       if (getVersionClock() != globalClock) {
16         applyRemoteCommits(globalClock);
17         validate();
18       }
19       commitToRepository(nextVersion);
20       RemoteCommit rc = new RemoteCommit(getServerId(), nextVersion,
21                                          this.writeSet.getVBoxIds());
22       broadcastRemoteCommit(rc);
23       writeBack(nextVersion);
24       commitSuccess = true;
25       return nextVersion;
26     } finally {
27       ClusterUtils.globalUnlock(commitSuccess ? nextVersion :
28                                 globalClock);
29     }
30   }
31
32   void applyRemoteCommits(int upToVersion) {
33     RemoteCommit rc;
34     do {
35       rc = ClusterUtils.getRemoteCommitMessages().take();
36       rc.apply();
37     } while (rc.getCommitVersion() < upToVersion));
38   }
39 }
```

Listing 3.15: Additional methods for the clustered commit algorithm. Code in gray color is reused from the previous commit algorithm in Listing 3.6 on page 46.

11. Update the local version clock.

12. Release the local lock.

The new extended version of the commit algorithm serializes transactions globally across the cluster and ensures that the changes made by a commit are visible to all transactions that attempt to commit afterwards, upholding strong consistency. The commit of write transactions still occurs within the local commit lock. This behavior remains unchanged from the JVSTM: Even though a single commit lock may be a potential bottleneck, this lock only affects the commit of write transactions; the upside of using a coarse-grained commit lock is that it is much simpler to reason about the commit algorithm and to understand the implementation.

Before acquiring the global lock—which is more expensive than the local lock—the commit operation first performs a local optimization that consists in applying any remote commits already queued locally (line 6 of Listing 3.14 on page 63) and executing a preliminary read set validation against the information currently available (line 7). This is a best-effort validation: If this preliminary validation fails, then it is possible to avoid the cost of acquiring the global lock, because this transaction is already invalid. However, even if validation succeeds, given that this node has not acquired the global lock yet, it is possible that another node may be concurrently committing a write transaction (but not this node, because of the local lock). So, even if the first validation succeeds, it may still be necessary to revalidate after acquiring the global lock. In any case, the validation procedure follows the JVSTM's original implementation, which checks whether any of the VBoxes in the read set of the committing transaction has a more recent version than the one that was read by this transaction: If so, then the transaction is not valid to commit.

Assuming local validation succeeds,[15] then the method `performGlobalCommit()` (line 10 of Listing 3.15 on page 64) attempts to perform the commit globally. Acquiring the global lock requires a successful *Compare-And-Set* (CAS) to change the current value of the atomic number, from a positive value to `-1`. The lock acquisition method (`globalLock()` shown in Listing 3.17 on page 69) returns the positive value read, which represents the clock version of the most recently committed transaction. If the lock is not available, then this thread must wait until it can acquire the lock: In a trivial implementation of the `globalLock()`, this thread simply yields and retries, but there are alternatives, such as to

[15] Otherwise, the sequential flow is broken due to a `CommitException` thrown inside the method `validate()` and the entire commit ends without success.

implement some kind of back-off algorithm, or to block on a distributed queue of transactions waiting to commit.

After acquiring the lock, it is then possible to determine how many remote commits still need to be applied, by comparing the global clock with the local clock, which contains the version of the most recent locally known commit (line 15 of Listing 3.15 on page 64). The local copy of the version clock is updated whenever either a transaction commits locally or a remote commit is processed (respectively, line 9 of Listing 3.14 on page 63 and line 17 of Listing 3.16 on page 67).

The algorithm then applies all the outstanding remote commits up to the version obtained from the global lock. This operation ensures that the most recent global changes will be accounted for, in case validation needs to be performed again. The entry point of the update operation is the method `applyRemoteCommits(int)` (line 32 of Listing 3.15 on page 64). This method returns only after locally applying the effects of all remote commits up to the version of the global clock. When this method is called, it is theoretically possible that no additional remote commit message is yet available in the local queue of remote commit messages,[16] in which case the method will block until the queue contains more elements to process (line 35). Applying a single remote commit consists in performing two operations:

1. For each locally cached VBox listed in the remote commit's write set, to commit a new VBoxBody with the special value `NOT_LOADED_VALUE` for the corresponding version (line 12 of Listing 3.16 on page 67). Later, when running a subsequent transaction, if any of these values is needed, it will trigger a reload as described in Section 3.5.5 on page 53. However, unlike the case for the `NOT_LOADED_BODY`, these bodies have a known version and will trigger only the reload from the TDR of the value for such given version (detailed ahead in Section 3.6.5 on page 71). When needed, the value is guaranteed to exist already in the TDR, because, in the clustered commit algorithm, writing the changes to the TDR (step 7) precedes broadcasting the commit to others (step 8). Additionally, notice that, for the moment, no other local transaction may be interested in the values of this remote commit, because the local version clock has not been updated yet. Moreover, only locally cached VBoxes are updated, which means that if the write set of this remote commit does not refer any VBox locally cached on the applying node, then applying the write set becomes an empty operation. Such can

[16] Messages may be delayed, but Hazelcast ensures that they are delivered in order. Although possible, a delay is unlikely, because remote commit messages are sent only within the global lock, which this thread now holds.

```
1  class RemoteCommit {
2    final int serverId;
3    final int version;
4    final Object[] vboxIds;
5    ...
6    void apply() {
7      for (Object vboxId : this.vboxIds) {
8        VBox vbox = getCache().lookup(vboxId);
9        // only update locally cached VBoxes
10       if (vbox != null) {
11         synchronized (vbox) {
12           vbox.body = new VBoxBody(version, NOT_LOADED_VALUE,
13                                    vbox.body);
14         }
15       }
16     }
17     setVersionClock(this.version);
18   }
19 }
```

Listing 3.16: Remote commit message.

easily occur in scenarios where the application's workload is split among nodes in such a way that the data accessed in each node do not intersect.

2. To advance the local version clock by updating it with this remote commit's version (line 17). This sets the linearization point for the remote commit: From this point onwards transactions that begin in this node will see this remote commit. Recall that in JVSTM all transactions read the version clock when starting.

After steps 5 and 6, which are required only if the local and global clocks differ after the acquisition of the global lock, the current committing transaction is globally valid. The following step in the algorithm is to store the changes to the TDR via `commitToRepository()`, as described in Section 3.5.2 on page 45.

Next, this commit sends a notification to the other nodes. To do so, it uses the distributed topic to broadcast a remote commit message containing:

- The identifier of the source node.
- This commit's new version.

- The list of identifiers of the VBoxes changed by this commit.

Each commit of a valid transaction entails the broadcast of one remote commit message. The underlying group communication system ensures that these messages are delivered in the order by which they are sent. Given that a remote commit message is broadcast only within the global lock, the resulting order of the commits is well established. After the commit message is delivered to another node, such node may already process it and, eventually, start a new transaction in such version, even while this node is holding the global lock.

After writing back to memory, the commit executes the write back phase of the JVSTM. The difference with regard to the application of a remote commit is that, for the current local commit, the values of the bodies are known and, therefore, they are immediately set, instead of using the special marker `NOT_LOADED_VALUE`.

The next step is to release the global lock. The `globalUnlock(int)` operation (line 16 of Listing 3.17 on page 69) takes a parameter that indicates the value to which the atomic number of the global clock should be set: If the commit was successful, the global clock is set to the `nextVersion`; if something went wrong, a `CommitException` will be thrown, and the atomic number is restored to its original value. Also, for successful commits, the local clock advances (line 9 of Listing 3.14 on page 63) and, finally, the local lock is released (line 11).

3.6.4 Starting a New Transaction

Recall that, when creating a new transaction the JVSTM already reads the local clock to determine the version in which the new transaction will read the values from the VBoxes. So, in a single node, new transactions always begin in the most recent version available, because all write transactions are guaranteed to perform node-local commits (as there is only one node), which naturally advances the local clock.

In the clustered version, however, the local clock advances either by committing a local transaction or by applying a remote one, as shown previously in the description of the clustered commit algorithm. Additionally, it may be the case that, at some point in time, a given node is not performing any write transactions. Nevertheless, it needs to update its clock, so that new transactions can begin in the updated state.

To ensure forward progress in the clustered JVSTM, I extend the algorithm for begin-

```
1  class ClusterUtils {
2    ...
3    static int globalLock() {
4      AtomicNumber lockNumber = getAtomicNumber(GLOBAL_LOCK_NAME);
5      do {
6        long currentValue = lockNumber.get();
7        boolean unlocked = currentValue != -1;
8        if (unlocked && lockNumber.compareAndSet(currentValue, -1)) {
9          return (int) currentValue;
10       } else {
11         globalLockNotYetAvailable();
12       }
13     } while (true);
14   }
15
16   static void globalUnlock(int value) {
17     AtomicNumber lockNumber = getAtomicNumber(GLOBAL_LOCK_NAME);
18     lockNumber.set(value);
19   }
20
21   static void globalLockNotYetAvailable() {
22     Thread.yield();  // alternatively sleep, wait on a queue, etc.
23   }
24 }
```

Listing 3.17: Group communication utilities.

```
1  class Transaction {
2    ...
3    void tryToApplyAllRemoteCommits() {
4      if (!ClusterUtils.getRemoteCommitMessages().isEmpty()) {
5        if (COMMIT_LOCK.tryLock()) {
6          try {
7            applyAllRemoteCommits();
8            setReadVersion(getVersionClock());
9          } finally {
10           COMMIT_LOCK.unlock();
11         }
12       }
13     }
14   }
15 }
```

Listing 3.18: When beginning a new transaction TMM processes any remote commit messages. This ensures that JVSTM's local clock advances, even when there are no local commits occurring.

ning a transaction to process any pending remote commit messages received and to apply their commits locally. I do this by including, at the beginning of any new transaction, an invocation to the method `tryToApplyAllRemoteCommits()`, shown in Listing 3.18, which wraps a call to the existing method `applyAllRemoteCommits()` (defined in line 3 of Listing 3.15 on page 64). With this extension, the beginning of a new transaction can cause state changes in the JVSTM to occur and, therefore, it executes while holding the local commit lock, much like it occurs already during the commit operation. The method `tryToApplyRemoteCommits()` first checks whether there is anything remaining to process in the queue of remote commit messages (line 4) and, if so, tries to acquire the local commit lock without blocking (line 5). If it succeeds in acquiring the lock, it applies all the remote commits that it finds and releases the lock; otherwise, it does nothing. The lock acquisition can fail, because the lock is being held either by a local commit or by another starting transaction. Regardless, some thread will run the procedure required to advance the local clock. An immediate consequence is that the beginning of any transaction remains nonblocking. Forcing all beginning transactions to acquire the same lock would prevent transactions from starting concurrently, which would be extremely penalizing, especially for read-only transactions, because not only are they much faster than write transactions, they are also expected to occur in much larger amounts. If this transaction successfully processes any remote commits then it also upgrades its read version to the most recent value of the

system's clock.

3.6.5 A Review on the Updates to the Shared State

When using the JVSTM, all of the application's shared state is made transactionally safe through the use of VBoxes exclusively. The bodies of any VBox are updated, in mutual exclusion, in the following situations only:

- During the commit of a local write transaction when writing back the values to main memory. This corresponds to the write back phase from the JVSTM, described in Section 3.3 on page 35.[17]
- When reloading the contents of a VBox from the TDR, as shown in Listing 3.11 on page 57.
- During the application of a remote commit, when committing the special `NOT_LOADED_VALUE` to the VBoxes residing in the node's local cache, as shown in Listing 3.16 on page 67.

All of the three cases identified above operate on each VBox in mutual exclusion: Each of them acquires the monitor of the single VBox instance that they intend to modify. As none of these operations manipulates other VBoxes simultaneously, deadlocks due to the acquisition of multiple monitors from different VBoxes cannot occur.

Before the introduction of the remote commits, a reload would be necessary only when the JVSTM detected a `NOT_LOADED_BODY` at the end of the list of bodies, in which case it would trigger the operation `reloadVBox(int)` (line 8 of Listing 3.11 on page 57). In a cluster, however, it is possible that the value requested corresponds to a version that may have been written by another node. In this case, the current node may have the VBoxBody for that version, but its value will be set, initially, to the `NOT_LOADED_VALUE` (line 12 of Listing 3.16 on page 67).

In Listing 3.19 on page 72, I extend the operation `getBoxValue(VBox)` from Listing 3.11 on page 57 to detect this case, and to reload only the specific missing value for the requested version. Note that, unlike the reload triggered by a `NOT_LOADED_BODY`, the

[17] Notice that JVSTM's original `writeBack(int)` operation (from line 8 of Listing 3.2 on page 37), which has not been shown in detail, was redefined to acquire the monitor of each VBox, when writing back the corresponding value from the write set.

```
class Transaction {
  ...
  Object getBoxValue(VBox vbox) {
    ...
    VBoxBody body = vbox.getBody(getReadVersion());
    if (body == VBox.NOT_LOADED_BODY) {
      synchronized (vbox) {
        vbox.reloadVBox(getReadVersion());
      }
      body = vbox.getBody(getReadVersion());
    } else if (body.value == NOT_LOADED_VALUE) {
      vbox.reloadBody(body);
    }
    ...
    return body.value;
  }
}

class VBox {
  ...
  void reloadBody(VBoxBody body) {
    Repository tdr = getRepository().getTDR();
    RepositoryEntry entry = tdr.get(makeVersionedKey(getId(),
                                                     body.version));
    body.value = entry.value;
  }
}
```

Listing 3.19: Final version of the VBox reloading operation. It now accounts for a missing value in any body.

required version is now known. Also, it is not necessary to acquire the VBox's monitor when reloading a missing value for an existing body, because this is a safe operation to perform concurrently, given that it does not change the structure of the list of bodies. In the worst case, the same value for a given VBoxBody could be overwritten multiple times by different transactions: The value of a VBoxBody should be seen as a *value object* [Fowler, 2002].

3.7 Consistency Guarantees

In this section, I describe, informally, the consistency guarantees of the transactions executed by TMM. Given that the proposed design results from extending the JVSTM, I take

JVSTM's consistency as the starting point, and discuss whether it is influenced by each of the extensions that I have added.

As already mentioned in Section 3.3 on page 35, the JVSTM provides opacity. Recall from Section 2.2.1 on page 13, that this correctness criterion is the combination of the following properties:

P1 All committed transactions can be ordered in some sequential history that describes the state observed and produced by each of those transactions, i.e. **serializability**.

P2 The order of temporally nonoverlapping transactions is preserved and their effects appear to occur within the interval in which they executed in real-time, i.e. **strict serializability**.

P3 Transactions always access consistent data, regardless of their outcome (commit or rollback), i.e. **reads are always consistent**.

Together, these properties provide very strong guarantees, which contribute greatly to simplify the programmer's reasoning. Simply embedding the JVSTM in an enterprise application, as described in Section 3.4 on page 39, endows the transactions that the application executes with these properties.

The strong consistency provided by the JVSTM relies mainly on (1) keeping the transaction's read set to validate that no location read changes during the execution of the transaction until it commits (to ensure properties *P*1 and *P*2), (2) reading the version clock when beginning a transaction (to ensure property *P*2), and (3) using versioning for enabling consistent reads throughout a transaction (ensuring property *P*3). Intuitively, as long as any changes retain these behaviors, the same consistency will be upheld.

The extension to enable storing data externally, described in Section 3.5 on page 41, retains all consistency properties—that is, it ensures opacity also. The changes in this extension are completely innocuous with regard to how the consistency properties are upheld: When getting a value from a VBox, the `get` operation always finds a value in memory to return, even if, transparently to the caller, it was necessary to load such value from the TDR. Any VBox that is read will enter the transaction's read set as normal. At commit time, the validation algorithm remains unchanged; there is only the additional invocation of the `commitToRepository` operation, which retains the commit semantics, even in the presence of catastrophic failures, as discussed in Section 3.5.2 on page 45. And, lastly, as

long as there is only a single JVSTM instance, the local version clock is the only clock, which means that any transaction will begin in the most recent version committed.

In Section 3.6 on page 60, I describe the last extension, which adds support for the deployment of multiple application instances in a cluster, each running an embedded JVSTM. In practice, each JVSTM instance works independently, except for the following interactions:

- At node startup, when the JVSTM instance acquires a unique identifier from a distributed atomic number.
- During a transaction, whenever data is loaded into main memory from the TDR.
- At commit time, when acquiring the global lock and broadcasting the commit message.

The first type of interaction is not relevant for consistency as it occurs only during the bootstrap process of a node and, therefore, it does not intervene in the execution of any transaction—I mentioned it only for completeness of the list of interactions.

The second type of interaction is indirect: Although the JVSTM instances do not communicate directly, they share state through the TDR. This state is of two types: application state and infrastructural state. The application state corresponds to the content of VBoxes that is stored at commit time and later loaded into main memory. The infrastructural state corresponds to the version of the global clock, which each node reads when bootstrapping (henceforth, the local in-memory clock advances when applying either a local or a remote commit). In either case, the persistent state is written/updated atomically during the commit of a transaction, i.e. the stored version clock advances at the same instant that the values for that new version become available in the TDR. Therefore, data is always available for any version that may be requested.

The third interaction occurs within a cluster-wide lock, so any state changes will occur in mutual exclusion. Additionally, the decisive validation of the commit occurs after ensuring that the local version clock is up to date with the global clock, so that any changes that have occurred before, in real time, will be taken into account and the transaction will always be ordered after the last write that has occurred globally. The only difference is that when beginning a transaction there is only a best-effort update of the local clock. Recall that, when beginning a transaction, I decided to retain the original behavior of the JVSTM of having a nonblocking start. This has a consequence: It makes it possible, as discussed in Section 3.6.4 on page 68, for a transaction to begin in a read version that is not the most

up to date version, with regard to the global latest commit version. This precludes property $P2$.

Nevertheless, this is not an issue for write transactions. Such transactions, when committing, will obtain the global lock and before running validation will ensure that they have locally advanced the clock up to the most recent committed version globally. So, validation will be performed against the most recent state, and at most it will lead to a restart in the new version. Read-only transactions, though, do not enforce such synchronization so they may effectively execute in a consistent state, but be serialized to a logical point in time that is earlier that their real start time.

So, in summary, when considering a history containing the transactions that execute in all nodes of a cluster, properties $P1$ and $P3$ are always upheld. Property $P2$ is always upheld for the write transactions, as well as for each subset of the history containing only transactions from the same node.

In practice, the only scenario that does not uphold opacity—only because it does not uphold $P2$—occurs when a read-only transaction is allowed to run without seeing the updates that have already been committed, in real-time in another node, before such read-only transaction begins, and this behavior is observed externally. For example, in the following sequence of interactions: Consider a client thread running a write transaction $T1$ in node $N1$. Afterwards, when the same client executes a second transaction $T2$, if $T2$ is read-only, and it happens to run on a node other than $N1$, then it is possible that $T2$ does not see the updates performed by $T1$. This can occur if the node running $T2$ has not updated its local clock with $T1$, by the time $T2$ begins.

If, however, ensuring $P2$ cluster-wide for all transactions is considered essential, here is a simple overview of some of the techniques that might be used:

- To delay the completion of every commit until the committing node receives an acknowledgment from all other nodes, indicating that the commit message has already been fully processed, i.e. the commit has been applied in all nodes. This alternative greatly increases the latency of each write transactions, but it does not affect the latency of the read-only transactions.
- To delay the completion of every commit until the committing node receives an acknowledgment from all other nodes, indicating that the commit message has already been enqueued for processing (even though it might not have been fully processed),

and, in each node, to force the processing of every enqueued commit message before starting any new transaction. This alternative will affect the latency of all transactions. The advantage is that the latency of the write transactions is less affected than in the previous alternative, because it needs only to ensure enqueueing, not really wait for the entire processing of the commit message, which is transferred to the beginning of every transaction.

- To use a load balancer for the cluster, which always sends requests from the same client to the same application server. This alternative is more complex than the previous, because it requires the incorporation of an additional element into the architecture. It will affect the latency of every clients' request, albeit not of the transactions themselves. Additionally, it assumes that each client does not communicate its knowledge of the system to other clients. Otherwise, it would be possible for a client with access to additional information, to realize that some write transaction of another client should already be seen, when it was not.

- To allow the client to inform the transactional system, about what is the most recent version that the client is aware of, so that the transactional system could delay the beginning of this client's transaction until it was up to date with such version. This addresses the limitation of the previous alternative. However, it requires collaboration from the clients, and for the existing transactional API to be extended. It has the advantage of only increasing the latency for those clients that specifically request a given version.

- Ultimately, to force every transaction to acquire the global lock when beginning, so that it could know what is the most recent value of the version clock and then to update to that version before beginning the transaction. This alternative requires the least changes. However, besides increasing the latency of all transactions, it also increases the usage of network bandwidth by forcing every transaction to acquire the global lock, which represents a scalability bottleneck. It might be an acceptable alternative only for systems running extremely long transactions, such that the cost of acquiring the global lock at the start of every transaction is diluted in the transaction's duration.

3.8 Experimental Evaluation

In this section, first I describe the experiments used to evaluate TMM, Then, I present and discuss the results obtained.

One goal, is to compare how the performance of an enterprise application is affected when using TMM versus using a standard implementation approach, that is, without TMM. A major design characteristic of TMM is that it is designed to reduce the number of read accesses to the TDR. Such, coupled with JVSTM's efficiency regarding its read operation, leads to the expectation that the performance for read-dominated workloads should improve, and that, the more read-dominated a workload is, the higher the expected improvement.

Another goal of this evaluation, is to experiment with the use of TMM on top of different TDRs. This experimentation is relevant for two reasons: For one, it should show that TMM, in fact, can be applied regardless of the underlying TDR, at least for the subset of TDRs tested. For another, it should show that the possible improvements obtained for the typical workloads are not specific to a given TDR of choice, and that they occur across different TDR implementations, albeit the actual improvements will, by force, be partially dependent on the performance of the TDR.

Real-world enterprise applications tend to compromise with a specific type of data storage, and usually it is not trivial to reimplement them using another type. Also, the sheer size of the source code makes these applications overly complex and expensive to manipulate, in order to obtain specific performance measurements for different scenarios/workloads.

Considering that using a real-world enterprise application would be a hurdle for several reasons, such as the ones I just mentioned, I opt to follow a common approach to measure performance, whereby I use a controlled benchmark, namely RadarGun, to simulate the workloads, typical of these applications, and measure the resulting throughput.

RadarGun[18] is a benchmarking framework, developed by Red Hat, and created specifically to support the comparison of various Java-based implementations of IMDGs. At its core, RadarGun implements a sequence of steps, called stages in RadarGun's terminology, which are executed concurrently by each of the nodes in a cluster. The stepping between each stage is controlled globally via a master node that is aware of the current stage running in each of the slave nodes. Figure 3.3 on page 78 shows an example of the typical stages provided

[18] `http://radargun.github.io/radargun/`

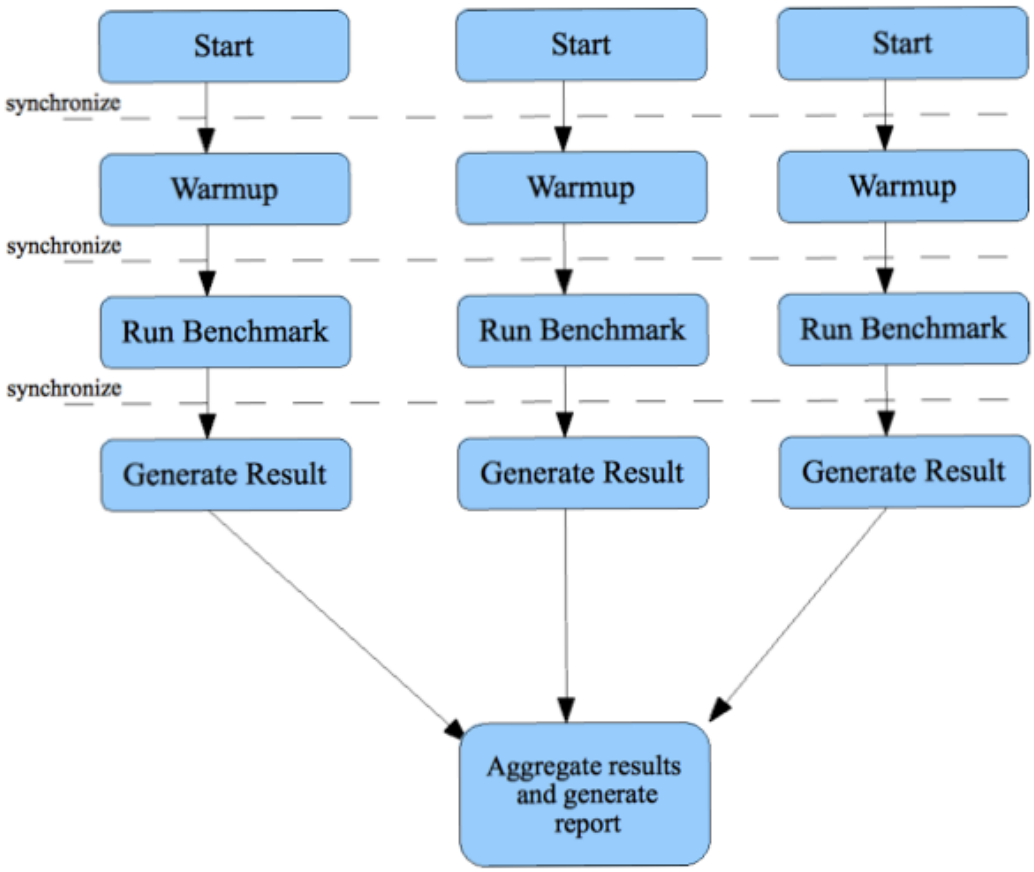

Figure 3.3: Example of stage synchronization in RadarGun. Each stage executes concurrently in each node. Synchronization between stages is controlled by a separate master node. Image source: `https://github.com/radargun/radargun/wiki/Five-Minute-Tutorial`.

by RadarGun: After ensuring that each node is deployed successfully and has joined the cluster, it runs a warm up sequence, followed by the actual performance benchmark. In the end, each node return its statistics to the master node, which then aggregates the results and generates the final report.

RadarGun makes it easy to replace one IMDG implementation with another: It defines a generic interface (`CacheWrapper`[19]) that must be implemented for each IMDG supported. This is done through a corresponding plug-in extension to the framework, one per concrete IMDG instance.

So, RadarGun provides a simple and practical way to test the same application logic on

[19] `https://github.com/radargun/radargun/blob/master/framework/src/main/java/org/radargun/CacheWrapper.java`

top of different IMDGs, which are the most challenging type of TDR against which to test TMM. Much like TMM, IMDGs are also developed with the goal of improving the performance of data access: Data is stored in main memory, so reading and writing data should be as fast as possible, at least much faster than in other types of TDR, such as disk-bound relational databases. Additionally, by running the IMDG embedded into the application (instead of in a separate tier, using a client/server model), the communication cost between the application and the IMDG amounts to method calls in the application code stack, just like in TMM. For these reasons, RadarGun is a good fit to use in the evaluation of TMM.

The `CacheWrapper` is similar to the `Repository` that I defined in Listing 3.1 on page 32, so that the `CacheWrapper` may be seen as a `Repository`, and the corresponding IMDG as a TDR. To integrate TMM into the RadarGun framework, I created a plug-in extension, whose `CacheWrapper` implementation decorates another IMDG, and implements the TMM algorithms described in the previous sections of this chapter. In turn, when TMM needs to access its underlying TDR, it makes the necessary calls on the `CacheWrapper` of the decorated IMDG. This way it becomes possible to test a given IMDG, and then to replace it with its TMM-decorated version, while maintaing the same application logic.

3.8.1 Description of the Experiments

Next, I present two groups of experiments. In the first group, I use a single application node, and vary the number of threads that execute concurrently (from 1 to 48), as well as the ratio of write transactions (from 1% to 20% of the total number of transactions executed). In the second group, I use a cluster of application servers (from 2 to 12), each with a fixed number of threads (4). The first scenario simulates vertical scaling, in which a single powerful server may be upgraded to have more resources, such as memory modules and CPU cores, as needed by the application to serve its clients. The second scenario corresponds to horizontal scaling, in which more parallel resources are added by joining more nodes to a distributed application.

All tests were executed on a single machine with a NUMA architecture containing four AMD Opteron 6168 processors. Each processor contains 12 cores (in two units of 6), thus totaling 48 cores. No other significant processes were executing in the machine during the tests. Furthermore, the system has 128GB of RAM, which is more than enough to distribute among all application servers. During the measurements there were no full garbage collections. The application servers executed with the

following options passed to the Java Virtual Machine: `-verbose:gc -Xmx5000M -Xms3000M -server -XX:+UseConcMarkSweepGC -XX:+AggressiveOpts -XX:NewRatio=5 -XX:SurvivorRatio=2 -XX:ParallelGCThreads=4 -Dhazelcast.operation.thread.count=4`.

Each transaction performs a fixed number (1,000) of random data accesses. Each thread manipulates (reads and writes) disjoint data. This way, there are never semantic conflicts among concurrent transactions. I am aware that a zero-conflict rate does not represent a real-world scenario, albeit not being far from it, by taking into account the conflict rates measured in the real-world applications described in Section 2.4.1 on page 22. Nevertheless, my intention is to measure the maximum possible throughput of the configurations under test, and therefore, the introduction of application-induced contention, which naturally limits performance, is not desirable.

The IMDGs under test are Hazelcast[20] v3.0.2, EHCache[21] v2.6, and Infinispan[22] v5.3.0. Although there are presently a few more plug-in extensions present in RadarGun's code repository, they either have been superseded by newer products (such as JBossCache by Infinispan), or failed to work as intended (such as Coherence), which has limited the experiments to the three aforementioned IMDGs.

Generally, IMDGs can be configured to store data either using full or partial replication among the available nodes. Full replication (replication for short) means that every node will store a copy of the same data, whereas in the case of partial replication (distribution for short), it is usually possible to configure how many replicas of each datum the system as a whole keeps. Replication tends to provide better read performance at the cost of having to synchronize potentially more nodes when committing. The opposite is true for data distribution, where the reads may have to fetch data from remote nodes. There is a myriad of configuration combinations, and these affect the semantics provided to the application. For example, one may consider whether stale reads are allowed (reads from replicas that have not been updated yet), whether written data should be updated synchronously/atomically in every replica, and whether each node may eventually decide to cache data that is frequently accessed.

Taking into account that TMM automatically caches, locally in each node, the data set manipulated by a transaction, and that the workloads under test are read dominated, I

[20] `http://hazelcast.com`
[21] `http://ehcache.org`
[22] `http://infinispan.org`

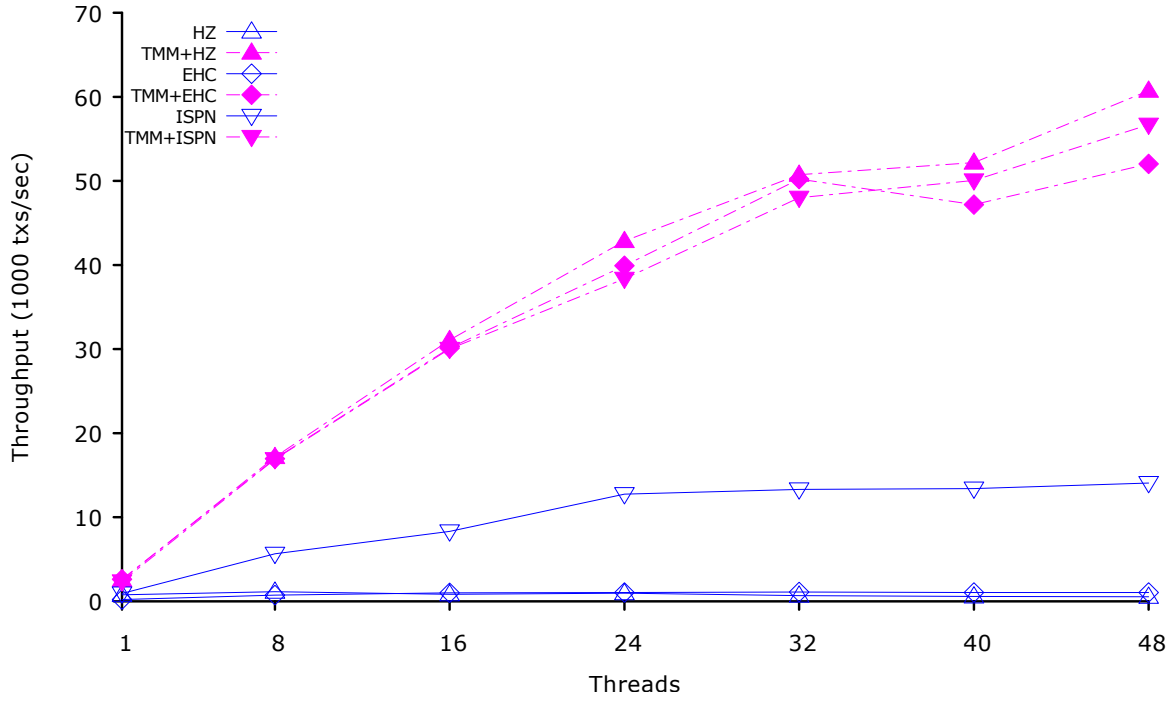

Figure 3.4: Total throughput for the single-node executions using 1% write transactions.

configured all IMDGs to use full replication. The only exception to this is Hazelcast, which, by design, is limited to support only six replicas of each datum. For this reason, in the clustered tests, when using more than six nodes, data will not be fully replicated, in Hazelcast alone.

3.8.2 Results

The results for the single-node executions are shown from Figure 3.4 to Figure 3.6 on pages 81–82. Each plot corresponds to a workload with a different percentage of write transactions. In all plots, the number of threads changes in the horizontal axis and the throughput is represented in the vertical axis (in thousands of transactions per second). Each plot for a given IMDG is labelled with a unique abbreviation: Hazelcast (HZ), EHCache (EHC), and Infinispan (ISPN). Each implementation with TMM decorating an IMDG is labelled with the prefix *TMM+*.

Regardless of the workload, the versions running on top of TMM are clearly better than their counterparts without TMM. This is true, even for write ratios of 20%, which can be regarded as high for the typical workloads of large enterprise applications: In the use cases described in Section 2.4.1 on page 22, FénixEDU and •IST measured consistently throughout

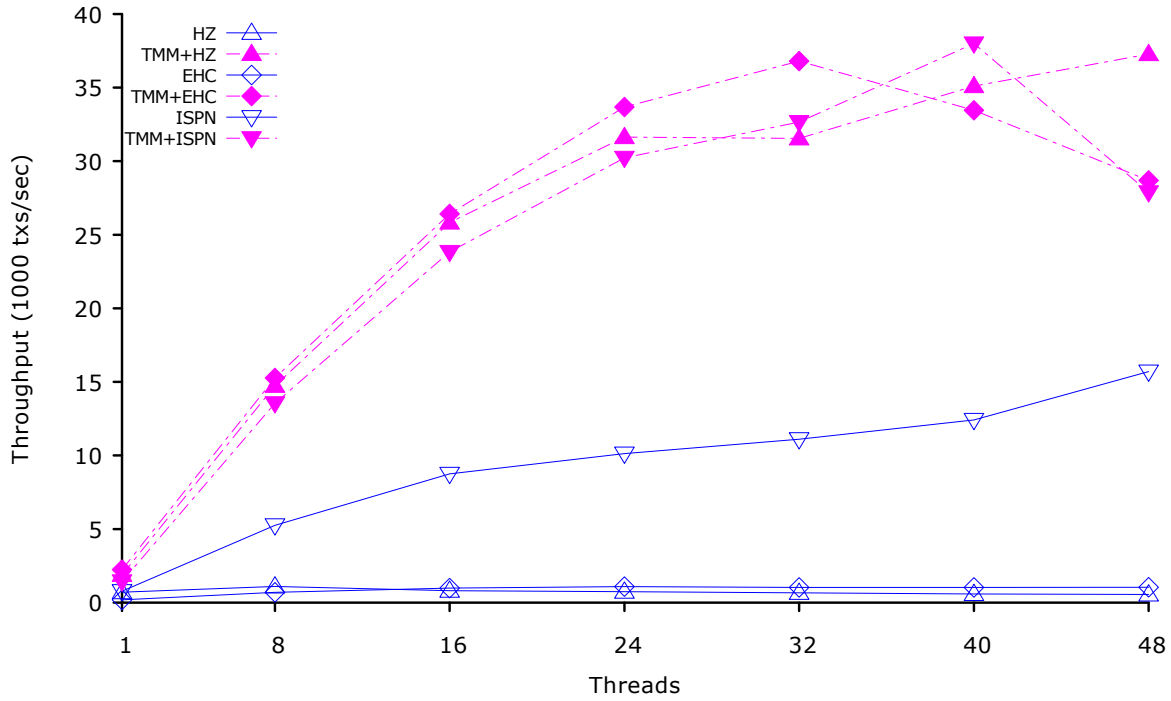

Figure 3.5: Total throughput for the single-node executions using 5% write transactions.

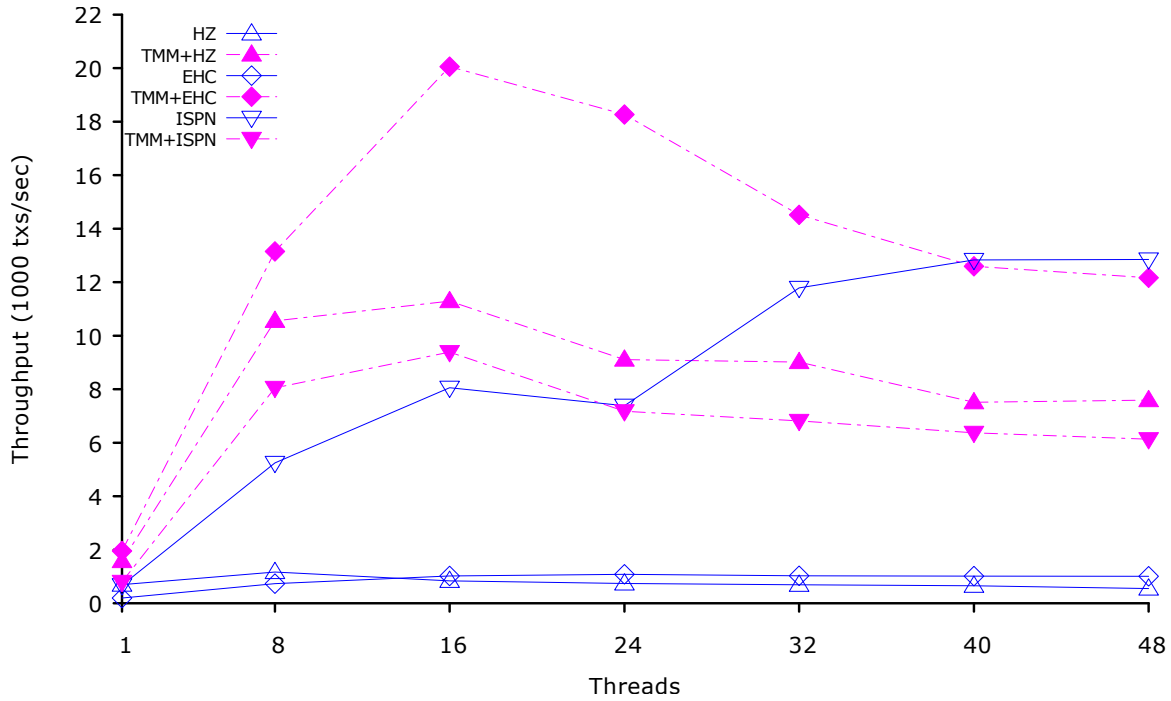

Figure 3.6: Total throughput for the single-node executions using 20% write transactions.

	Minimum and maximum speedup per workload type					
IMDG	1%		5%		20%	
HZ	3.27	111.28	2.68	67.55	2.26	13.80
EHC	12.32	49.29	11.55	35.64	9.96	19.62
ISPN	2.53	4.03	1.78	3.06	0.48	1.53

Table 3.1: Minimum and maximum speedup achieved by TMM, in a single node, when decorating each IMDG. TMM performs, at the minimum, twice as good as HZ (2.26 minimum speedup), and almost ten times better than EHC (9.96 minimum speedup). It looses only for ISPN at 20% write transactions.

several years an average of write transactions below 2%, and the same was reported for other applications.

At 1% write transactions, the best performance improvement ranges from 4 times more throughput (achieved for ISPN at 48 threads), to as high as 111 times more (achieved for HZL at 48 threads). Table 3.1 summarizes the minimum and maximum speedup achieved by TMM when decorating an IMDG, in each of the three workloads. The improvements are very significant, across all workloads. The minimum speedup (that is, the lowest relative improvement), is above 1 for all scenarios except for ISPN at 20%. Still, it is noteworthy that, whereas at 20% write percentage ISPN is able to outperform TMM for a high thread count, the peak throughput achieved TMM+EHC at 16 threads is substantially higher than ISPN's highest throughput. In this scenario at 48 threads ISPN has approximately double the performance of TMM+ISPN. On the one hand, ISPN is able to scale much better than the other IMDGs. On the other hand, at 20% write transactions, for all TMM-based versions, the effect of the single commit lock is noticeable, in that, for a low number of threads the performance increases (approximately, up to 16), but then the relatively higher rate of commits causes the single commit lock not to scale, as the threads begin to have to wait to acquire the lock. In fact, already at 5% write transactions, the effects of using a single commit lock become apparent at 48 threads. Intuitively, as long as the time it takes to commit a transaction is short enough, so that all transactions can access the commit lock without having to wait to acquire it, it will be possible for TMM to scale.

As the total number of concurrent transactions increases—either because the absolute number of concurrent transactions increases for a given percentage of write transactions (moving from left to right in each of the horizontal axis), or the percentage of write transactions increases for a given number of total transactions (moving from one plot to the next)—so will

the number of concurrent commits increase, thereby tending to limit the availability of the single lock for each thread.

Nevertheless, on the overall, and as expected, using TMM in a single node greatly improves the throughput of the application. This is so, even when the comparison is made with other implementations that also perform local memory accesses to read data. After the warm-up stage of the benchmark, the cache of VBoxes is supposed to be loaded with the application's data, which means that during the benchmark stage, the values will be available in this cache, and TMM will not need to contact the TDR during the body of the transaction to get data. Only during commit of a write transaction will the TDR be needed. In spite of the potentially high cost of the single lock approach, taking less time to execute the transaction's body helps to mitigate the commit bottleneck that becomes visible only for higher commit rates.

Let us now consider the second scenario, in which the number of concurrent threads also increases up to 48, but this time by adding more nodes, each using a fixed number of four threads. The results are displayed from Figure 3.7 to Figure 3.9 on pages 85–86. Again, each plot corresponds to a workload with a different percentage of write transactions. The horizontal axis contains the variation in number of nodes tested, and the vertical axis displays the overall throughput (all nodes combined).

Regarding the IMDGs alone, the first aspect worth noting is that they perform quite differently from each other. Both ISPN and EHC are able to scale as the number of nodes increases, albeit at very different rates, with ISPN displaying much better performance. HZ always begins slightly above EHC, but then its performance drops substantially above 6 nodes. This is caused by HZ's design choice of limiting to 6 the maximum number of replicas of each datum. When running at 8 nodes or above, some of the data accesses will necessarily involve remote calls, which is extremely penalizing when compared to a local read. This explains HZ's performance drop. Even though each IMDG performs differently, their performance tendency appears to be little affected from a variation in the percentage of write transactions, that is, each of them displays similar lines in each plot, apart that their overall throughput drops slightly as the percentage of write transactions increases.

With TMM, however, the total thoughput of the system decreases when more nodes are used, except when using ISPN for 1% write transactions. In this case, TMM scales up to 6 nodes, improves on ISPN (doubling the throughput), and it keeps better performance up to 10 nodes. Then, just like in all other workloads, the tendency is for a flat line, with a

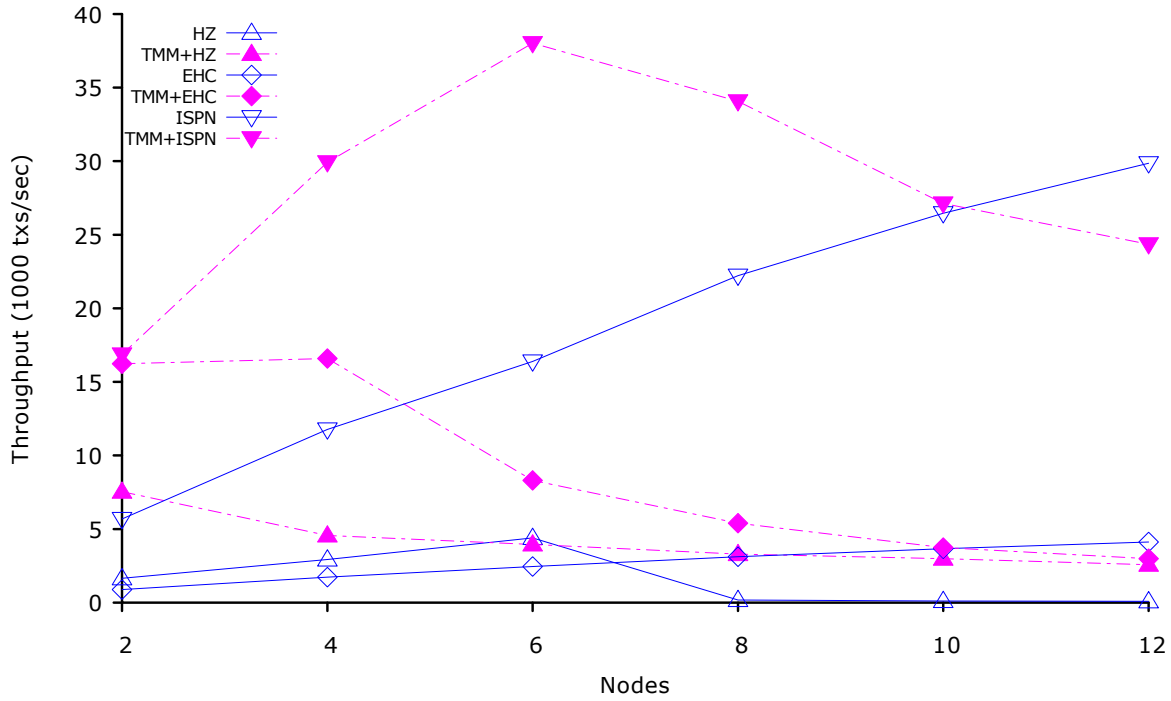

Figure 3.7: Total throughput for the clustered executions using 1% write transactions.

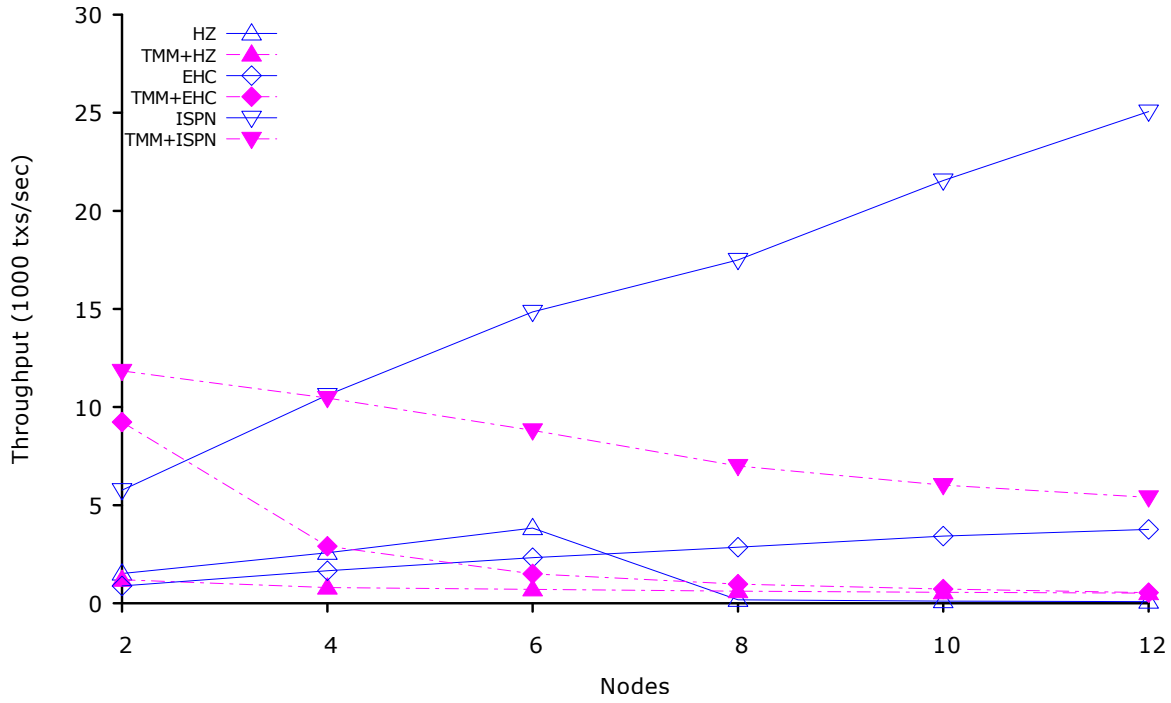

Figure 3.8: Total throughput for the clustered executions using 5% write transactions.

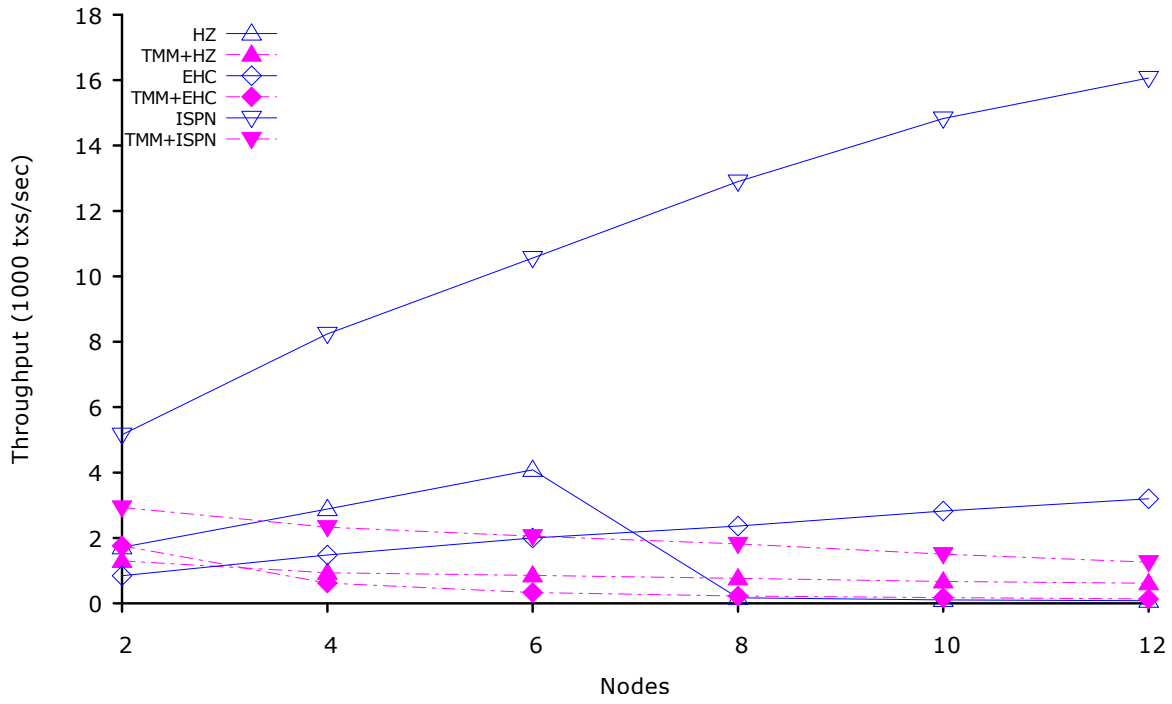

Figure 3.9: Total throughput for the clustered executions using 20% write transactions.

small downwards tendency. The reason for this improvement over ISPN, is related to ISPN being the best performing IMDG of the three, and to the single commit lock: What happens is that, once again, while the duration of the commit operation is short enough to avoid contention, the transactions can progress concurrently. The fact that ISPN has such good performance, when compared to the other IMDGs, allows for TMM+ISPN's commit to take less time, and thus, contention is avoided up to a higher number of threads. When this occurs, the transactions can progress concurrently, and the improved transaction performance greatly enhances overall throughput.

When TMM is using the other IMDGs, their performance affects the commit so much that contention occurs from very early on. Still, it outperforms HZ for 1% write transactions; for 5% and 20%, when running above 6 nodes, it is better only because of the performance drop in HZ, and because of TMM's local cache that prevents read accesses to the TDR.

The problem with the commit of write transactions becomes apparent when considering two opposite workloads: a read-only scenario, and a write-only scenario. In the read-only workload displayed in Figure 3.10 on page 87, it is clearly seen that TMM performance is independent of the underlying TDR. In fact, after warming up, there are no more accesses

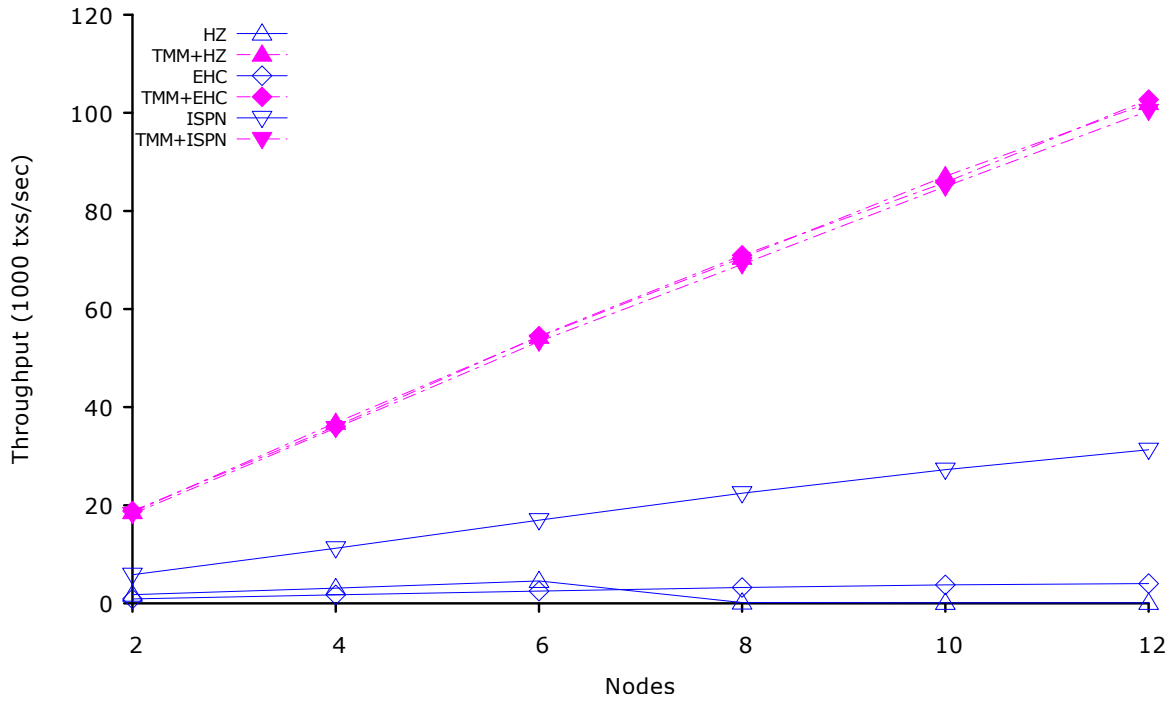

Figure 3.10: Total throughput for the clustered executions using read-only transactions.

to the TDR during the benchmark, because all data is cached. As a result, scalability is linear, and performance is much greater than that of the corresponding IMDGs. On the other extreme, shown in Figure 3.11 on page 88, the performance of all TMM-based implementations plunges. The only relevant difference between the two workloads is in the commit operation, which is empty in the read-only case. Notice that, although each write commit still invalidates data in the other nodes, reloading such data is never required, because each thread is accessing disjoint data.

Lastly, let us consider, for each workload type, the best overall performance achieved by each IMDG, against the best performance achieved by the corresponding TMM implementation, regardless of the application deployment topology. Such comparison is made possible because both groups of experiments ran in the same physical machine, and the total number of threads used in each deployment is identical between the two groups of tests. For example, a clustered execution with N nodes provides as many concurrent threads as the single-node execution at $4 \times N$ threads. Table 3.2 on page 88 presents these values. Except for ISPN at 20% write transactions, the TMM-based implementation always outperforms its counterpart, and most interestingly, the best performance for TMM is always achieved in a single node. Even though there may be several reasons why an application would require deployment

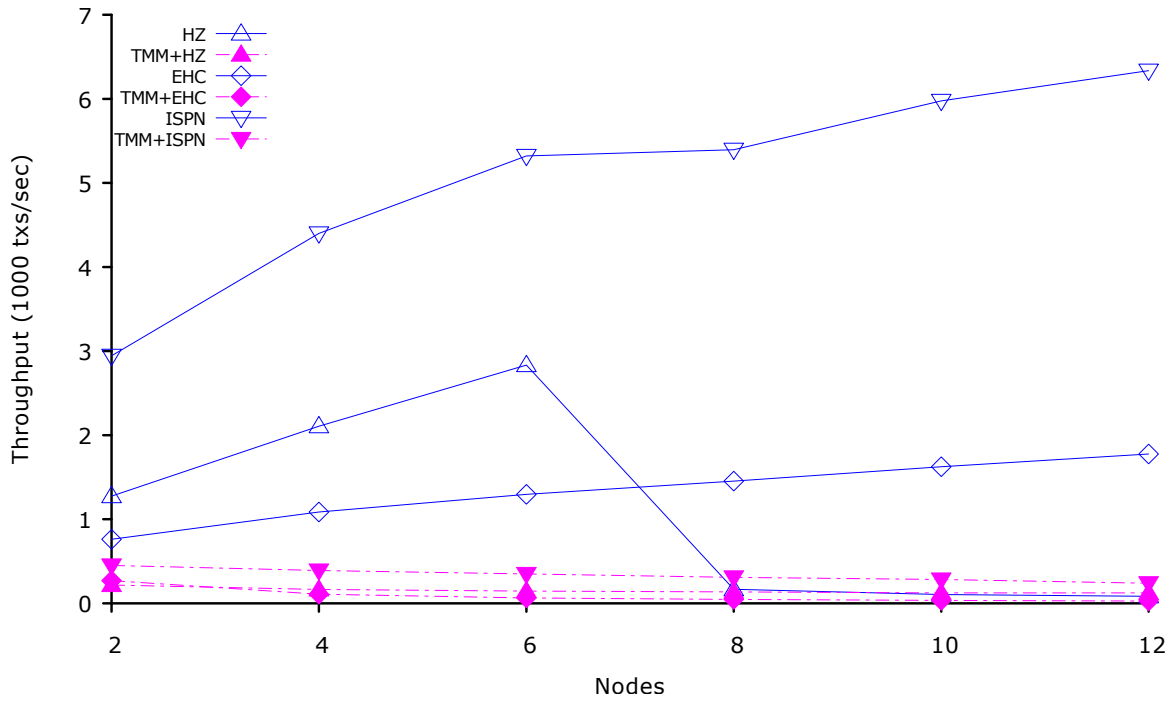

Figure 3.11: Total throughput for the clustered executions using 100% write transactions.

in the distributed setting, at least for those applications where the workload fits in a single powerful computer, a TMM-based implementation clearly offers major improvements, for the most common workloads. Notice that, even assuming continuous scalability of the IMDG solutions, it would take a high number of nodes to even match the performance reached by TMM in a single node, making TMM much more efficient in terms of resource usage.

As an ending remark, it should be noted that in all experiments TMM's transactions execute with the consistency guarantees described in Section 3.7 on page 72, whereas the transactions from the IMDGs, at best, can provide *Repeatable Read* semantics only, which

	Highest overall throughput for each implementation by workload					
Workload	HZ	TMM+HZ	EHC	TMM+EHC	ISPN	TMM+ISPN
1	4394	60728	4116	52025	29859	56684
5	3835	37284	3762	36797	25063	38000
20	4083	11292	3199	20057	16065	9388

Table 3.2: Highest throughput ever achieved by each implementation. The TMM-based implementations are able to obtain their best results using only a single node.

involves different complexities and cost of bookkeeping per transaction, within in each implementation.

3.9 Discussion and Related Work

Snapshot isolation was first defined in [Berenson et al., 1995]. Today, most database vendors support snapshot isolation as the strongest isolation level (confusingly, also named serializable in some database implementations[23]), which, despite exhibiting none of the anomalies that the SQL standard prohibits, is not serializable. Since the broad adoption of this isolation level by database implementations, other researchers have cataloged the weaknesses present in this isolation level [Fekete et al., 2004].

In [Fekete et al., 2005] the authors propose a theory to identify when nonserializable executions of applications can occur under snapshot isolation, which has been used to manually demonstrate that some programs may safely run under snapshot isolation without exhibiting any anomalies. This is done by analyzing possible interferences between concurrent transactions. Additionally, the authors also propose ways to eliminate the identified interferences, but at the cost of manually modifying the program logic of these applications to make it serializable.

Later work [Alomari et al., 2008], has demonstrated that the risks involved in using weaker consistency semantics often do not pay off for the absence of serializability. The effort of ensuring safe executions under snapshot isolation is high, and in many applications the performance penalty for running with full serialization guarantees is low, when compared to the risks of data corruption because of concurrency bugs due to the absence of strong consistency guarantees. Also, these same feelings are expressed anecdotally by software engineers [Shute et al., 2013].

In another work [Jorwekar et al., 2007] its authors propose an automated tool to detect the snapshot isolation anomalies, but to ensure full coverage, they incur in false positives. Also, computing the minimum set of conflicting transaction pairs that require modification is a NP-Hard problem. Regardless, modifying the program code to eliminate the conflicts is still a manual task.

An alternative to changing program code is to change the transactional engine to ensure

[23] PostgreSQL versions prior to 9.1, provided only snapshot isolation when selecting the serializable isolation level.

strict serializability. This is proposed in [Cahill et al., 2009]. The authors describe an algorithm that provides serializability, while showing, in most cases, a performance similar to the one obtained when running with just snapshot isolation. This work, provides evidence that serializability does not incur necessarily in performance problems.

Yet another approach to ensuring program correctness is presented in [Bernstein et al., 2000], which defines a new correctness criteria—semantic correctness—that is weaker than serializability, and describes the conditions in which executions preserve given integrity constraints for each isolation level.

Neither of the approaches previously mentioned focuses on the reduction of the number of database queries. Such requires some form of caching on the application server to avoid database access. In [Perez-Sorrosal et al., 2007], the authors present a versioned object cache for J2EE that reduces the number of database accesses and provides snapshot isolation on top of a relational database only. This work is in line with my purpose of reducing the number of calls to the data repository, via caching of data, as a means to improve performance. The authors show how standard (non-versioned) cache implementations may even further weaken the consistency semantics provided by the data repository, by inadvertently providing inconsistent reads. Their solution imposes that the database must also be able to provide snapshot isolation. They also describe a replication model where each node in the cluster is built from a relational database and a J2EE application server. Their object cache allows for a reduction in the number of database requests, which is akin to the purpose of the cache in TMM. There are, however, some noteworthy differences. First, their middleware does not ensure referential uniqueness: When an object is modified, a new copy is created and stored under a different cache entry. In TMM, even though internally there may exist multiple versions for the values of a given key, from the programmer's perspective there is only a single value accessible, always identified by the same key. A second difference regards to the inherent concurrency of the transactional system. TMM blocks during the commit of a write transaction. Their approach uses a different lock for each object update operation, and this is related to not being able to ensure more than snapshot isolation semantics. A third aspect is that their work is geared towards providing availability and scalability, which led to an architecture of one database per application server, with the cache updates being sent over a dedicated group communication system. However, there is no protocol for dynamically adding a new node to the cluster, which would require database state synchronization to keep up with the rest of the system. In TMM, there is conceptually a single database—the TDR, which can be replicated internally—and the cache in each application server is updated

from this database, allowing for adding and removing application servers at any moment.

There has also been recent work, geared towards providing increased consistency without compromising scalability. For instance, the work from [Couceiro et al., 2009] and [Romano et al., 2009] focuses on extending the JVSTM's design with scalable and fault-tolerant distributed transactional memory algorithms. Their work builds on a group communication system between the nodes, currently implementing an algorithm where only a single member of the cluster is appointed as responsible for applying the effects of the write transactions to the database.

GMU [Peluso et al., 2012] implements a genuine partial replication protocol and it uses a multi-version scheme to ensure that read-only transactions are never aborted. It ensures Extended Update Serializability [Adya, 1999], a consistency criterion weaker than opacity, which is provided by the JVSTM. A distinctive feature of TMM is that it is independent of the underlying TDR, whereas any GMU implementation will need to be specifically tailored to the target system.

TMM clearly improves performance for applications running on single computers with many cores and large heaps. Any performance improvements attained in a clustered environment are, in a sense, a by-product of this design, which I believe can be further improved, by taking into account the specificities of a distributed system in the design of TMM. Some of the technical solutions proposed in the aforementioned works, could be interesting to improve the performance of the clustered TMM, such as the use of vector clocks for overcoming the potential bottleneck of the global clock.

JVSTM was the first and, to the best of my knowledge, the only STM used in the implementation of real-world enterprise applications: FénixEDU [Instituto Superior Técnico, 2011] has been using the JVSTM in a production environment since 2005. However, the solution implemented is hardwired to a single relational database. It relies on the database (1) to store application data, (2) to acquire a cluster-wide lock, and (3) to share cache invalidation messages. The solution that I propose abstracts the underlying database, and can be applied to any type of data storage system. Nevertheless, it still suffers from similar limitations regarding the commit lock.

3.10 Summary

This chapter describes TMM, a new middleware for enterprise applications that leverages on STM technology and recent hardware to be able to provide strong consistency and, for some workloads, to greatly improve application performance, in particular, when executing in a single powerful computer. A key feature of this solution is that transaction control is shifted into the application, and the database is used only as a data storage element.

The chapter begins with an overview of the proposed solution, in which I describe its main elements and how they are combined, to produce the end result. Then, I present the algorithms that implement the proposed design. The presentation is partitioned in three stages, starting from the embedding of the JVSTM into the application. Each stage builds on the previous stage and extends it with additional functionality. In the end, the middleware is capable of supporting the application's needs for transactional operations over shared data that is persisted in a data repository, and it supports clustered deployment of the application server. Each node in the cluster contains its own STM instance. These instances coordinate themselves through a synchronization protocol that invalidates cached entries whenever a commit writes new data, and uses the underlying database strictly for storing and sharing data across the cluster.

To evaluate the proposed middleware, I use a benchmarking framework, RadarGun, which is specifically designed to compare different IMDG implementations, integrated as plug-in extensions. I created a plug in that uses TMM to decorate any other plug in, which enables to easily compare how the same application logic performs when executed on top of each IMDG, as well as when using TMM's plug in to decorate these IMDGs. Even though all TMM-based implementations provide strong consistency, whereas IMDG-based implementations provide at best repeatable read semantics, the results show increased application performance, most notoriously for the single-node scenarios, where depending on the IMDG the best speedups vary from approximately four times to over a hundredfold.

The clustered executions using TMM are clearly affected by the use of a single commit lock. In spite of that, it is noteworthy that the performance improvements in a single node alone are such that, even assuming continuous scalability of any of the IMDGs tested, it would take them several nodes just to reach the performance of TMM. For example, in the 5% write transactions workload, Infinispan would need 18 nodes of 4 cores each to reach the performance of TMM in a single 48-core machine. Notice that the solution to increase

the number of cores per node does not seem to help the IMDGs, as seen by their single-node performance. Taking into account that the recent trend has been to include many cores in a single processor, TMM seems a good fit for upcoming hardware.

Using a single commit lock helps to keep the design of TMM simple and, apparently, it is an acceptable compromise between improving performance and providing strong consistency. Nevertheless, it represents a naive solution for the clustered environment. Moreover, the requirements of some applications may prevent them from running in a single node, thus requiring a clustered deployment. In the next two chapters, I focus on improving the performance of TMM for the clustered environment, by providing an alternative design that enables the commit of write transactions to proceed in parallel with others. First, this requires me to redesign the JVSTM alone to make it nonblocking, which I do in Chapter 4, and then to revisit the ideas from the current chapter to design a new nonblocking middleware extension, which I do in Chapter 5.

4 Lock-free JVSTM

The design of the JVSTM allows it to excel in read-intensive workloads [Carvalho et al., 2008], but raises doubts on its applicability to other types of workloads, where the percentage of time spent committing write transactions increases. Even though during the execution of a write transaction reads and writes of transactional locations are wait-free, the commits of these transactions serialize on a single lock, thereby having the potential of impairing severely the scalability of a write-intensive application.

In this chapter I address this problem by presenting and discussing the implementation of a new and efficient lock-free JVSTM that improves scalability while maintaining the exact same properties for reads, as before. This nonblocking JVSTM allows commits to proceed in parallel during the validation phase, and it resorts to helping from threads that would otherwise be waiting to commit, during the write-back phase. Additionally, I describe a new GC algorithm for the JVSTM, designed to eliminate an existing infrastructural bottleneck.

I perform an evaluation of the new version by comparing it with the lock-based version, as well as with other top-performing STMs, using a test machine with 208 real processor cores.

This chapter provides the following content:

- A discussion of the factors that can affect the performance of the lock-based implementation, motivating for a nonblocking design.
- A new lock-free algorithm for the commit phase.
- A new GC algorithm that asynchronously cleans up unreachable versions and reduces the overhead of starting/finishing a transaction.
- A performance evaluation, comparing the new JVSTM against the lock-based JVSTM and other top-performing STMs.
- A discussion on the related work.

4.1 Motivation for a New Commit Algorithm

The JVSTM was initially developed to support read-heavy applications, in which the number of read-only transactions corresponds to more than 95 percent of the total number of transactions executed. As such, the other 5 percent have a lower probability of overlapping in time and, with the commit phase taking only a fraction of the entire transaction duration, there is an even lower chance for contention in the commit lock. Therefore, lock contention was expected to be low and evaluation of the single-lock approach showed good performance results in the workloads to which it was applied [Cachopo, 2007; Carvalho et al., 2008]. Moreover, an algorithm based on a single lock is simpler to implement and to reason about than alternative approaches, such as fine-grained locking and lock-free. These facts put together justified the initial use of the single-lock approach. Nevertheless, factors such as the **percentage** of write transactions, the **number** of concurrent transactions, and the **duration** of the commit phase can affect the performance of the JVSTM. Next, I present some arguments in favor of a nonblocking implementation.

4.1.1 Locks do not Scale

Different applications may have different workloads regarding read and write transactions and may execute many write transactions in a short period of time, especially in the case of CPU-intensive applications. In turn, this may highly increase the probability of having more than one write transaction trying to commit concurrently. Even if this may not be an issue for machines with a small number of cores, many-core machines are an emerging reality and the number of cores available at an affordable price is growing. It is reasonable to assume that, in a many-core machine, the higher contention on the lock will degrade performance significantly. For example, considering a rate of only 5 percent write transactions, and assuming that all transactions have the same duration, whereas a fully used 20-core computer will be dealing, on average, with one write transaction at any given time, a 100-core computer is expected to be dealing with five concurrent write transactions. Even a small increase in the percentage of write transactions can greatly increase contention.

4.1.2 Lock Convoying

Lock convoying [Blasgen et al., 1979] is a phenomenon that occurs when multiple threads contend repeatedly for the same lock. Typically, this leads to the additional overhead of

repeated context switches and causes overall performance degradation. In JVSTM, this effect is worsened when a transaction needs to restart due to conflicts, because during its re-execution, it will have a higher change of seeing more transactions committing ahead of it, thus increasing the chance for a second conflict to occur.

For any given workload that includes conflicting transactions, the more transactions run concurrently, the higher the probability that a conflict will occur. However, it is not so obvious that this situation may be aggravated when running N transactions in fewer that N processors.

As I have mentioned before, a write transaction has a conflict if some value that it read has already changed when it tries to commit. So, intuitively, the probability of conflict increases with the length of the transaction and the number of concurrent transactions, but it should not depend on the number of available processors, because having fewer processors should slow down all transactions proportionally and, thus, maintain the probability of having transactions committing during the execution of another transaction. Yet, this intuition is assuming that all transactions can proceed without interference. Once a transaction has to wait for the commit lock then the assumption is no longer true.

The JVSTM's commit operation uses a fair locking policy. This means that the lock is granted to the longest waiting transaction, i.e., transactions obtain the lock in the temporal order by which they requested it. Suppose now that a write transaction T_1 wants to commit, and attempts to get the lock. If another transaction T_0 already holds the lock, then T_1 has to wait. **The more transactions there are running, the less processor time T_0 will have to complete its critical section and release the lock**. Thus, there is a higher probability that other write transactions $T_2, \ldots, T_n$ will queue up for the lock as well, because they will be given processor time to execute, whereas T_1 is still waiting. Consider that when T_1 finally gets the lock it fails validation, so that it will immediately release the lock and restart. Releasing the lock and restarting is extremely likely to be much faster than committing T_2 (which is the next in line to hold the lock), thus T_1 will restart and use the version number of T_0's commit (the most recent at the time of the restart). T_1's next commit attempt will be necessarily after $T_2, \ldots, T_n$ commits have been processed, some of which may succeed. In practice, the span of T_1 in terms of versions from its **restart** up to its **commit**, i.e., the time length during which other transactions may commit and lead to yet another conflict in T_1, will effectively be greater than if it had been able to run without lock interference. This was caused by the amount of time that it had to wait for the lock until it could decide that it had to restart, which provided a chance for more transactions to pile up on the commit lock's queue.

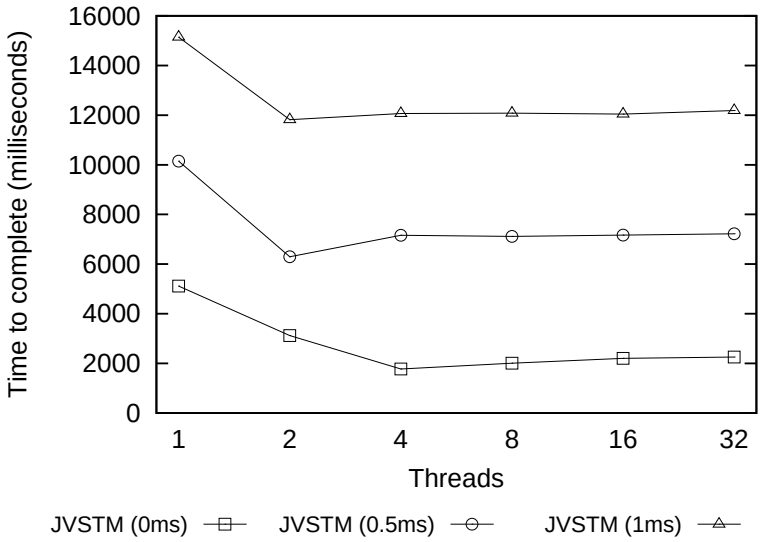

Figure 4.1: Demonstration of the poor scalability of lock-based commit. As the duration of the commit operation increases, the scalability of the application becomes increasingly limited.

In summary, under the circumstances described above, if a transaction restarts, then it faces a higher probability of restarting again.

4.1.3 Increased Lock Duration

During the commit phase, changes have to be written back to their boxes in memory. In this dissertation, I use the JVSTM on top of a data repository, which requires that changes are also written to such repository during the commit. Depending on the repository used, this additional operation can take up to several orders of magnitude longer than just writing back to memory.

One of the assumptions for enabling the use of locks in the first version of the JVSTM was that the commit is a short-duration operation, meaning that the commit lock would be released quickly enough not to cause much contention. The usage scenarios that I target break this assumption. It is even likely for the commit phase to become longer than the body of the transaction, due to the need of remote communication when writing back to the repository, thus dominating the total transaction time. As a result, the increased duration of the critical section contributes to the exacerbation of the other problems that I described as a motivation for a new commit algorithm in the previous subsections.

To illustrate the potential problems with the lock-based approach, I executed a micro-benchmark application—the Array, which will be described in Section 4.4.1.1 on page 121—where the time to commit was artificially altered with the introduction of active delays, while the lock was being held by the committer, in order to simulate a longer commit duration, hypothetically caused by the interaction with a data repository. The results shown in Figure 4.1 on page 98 were obtained running to completion 10^4 read-write transactions with almost no conflicts (each transaction reads 10^3 memory locations and writes to 10 out of 10^6 possible memory locations). The experiment used from 1 to 32 concurrent threads in the same 48-core machine that was already described in Section 3.8.1 on page 79. Each plotted line was obtained adding different delays to the commit operation, namely 0ms, 0.5ms and 1ms per commit. The results show that increasing the commit time naturally leads to an increase in the time that it takes for the single-threaded executions to complete. But the more relevant results are that, in all cases tested, adding more concurrent threads, improves the performance up to a certain number of threads, and then performance remains constant. This is because the executions are becoming serialized by the lock. The longer the delay introduced by the lock, the earlier cease the improvements obtained by adding more threads. I shall revisit this workload in Section 4.4 on page 120 using the lock-free JVSTM to show the scalability of the new algorithm.

Next, I describe the two main modifications that I apply to the JVSTM: First a complete redesign of the commit algorithm and, second, a new GC algorithm. The former is required to turn the JVSTM into a completely lock-free STM. The latter, addresses a performance bottleneck that only manifests under certain workloads with a high thread count; its need only surfaced after experimentally testing the JVSTM in a hardware that enabled for a high level of real parallelism. I describe these two new algorithms in detail in the following sections. Their fully working implementation is available at `https://github.com/inesc-id-esw/jvstm` from commit with SHA-1 checksum `754692dd723ab98e16269911910a420c73c2b5c9`.

4.2 Lock-Free Commit

In this section I describe the new lock-free commit algorithm that replaces the existing lock-based implementation. Conceptually, the algorithm contains the same three steps as before: (1) validate the read set, (2) write back the write set, and (3) make the commit visible to other transactions. In the following subsections I explain how each step is now accomplished in a manner that ensures the properties of lock-freedom.

4.2.1 Validation

The original lock-based commit algorithm uses **snapshot validation**, which, for each element in the read set, checks whether such element's most recent version is still the same that was read during the transaction. Yet, validation can also be performed incrementally by checking the write sets of all transactions that have committed versions greater than the one this transaction started with (**delta validation**). Suppose that transaction T started in version v_1. For any transaction T_i that committed a version greater than v_1, if any element in the write set of T_i is in the read set of T, then T cannot commit, because of a conflict.

By removing the commit lock and, consequently, eliminating any form of dependency between transactions that are committing concurrently, it is now possible that, while T is validating itself, other transactions are also validating themselves, aiming to commit as well. It is, therefore, necessary to order the commits in such a way that each transaction can finish validation and be sure that it is valid to commit.

To do so, I use and extend the functionality of the ATR list that already exists in the JVSTM. The ATR list holds records that contain information about a write transaction that has already committed. To support the delta validation, I extend this list to also include valid, but not yet fully committed records. Thus, a transaction that can get an entry in this list effectively establishes its commit order and, as such, its commit version. The code in Listing 4.1 on page 101 shows how a transaction concurrently validates its read set in a lock-free way.

Each write transaction now has two transaction records: The original `activeTxRecord`, which points to the record that represents the version in which the transaction started; and, the `commitTxRecord`, which is the record that is created to represent the transaction's own commit. To be valid, a transaction needs to check for an empty intersection between its read set and the write set of each record from the `activeTxRecord` onward. This is what the method `validate(ActiveTxRecord)` does: It checks all records from the `lastValid` onward and it returns the last successfully validated record (or throws a `CommitException` if validation fails at any point). Next, the new `commitTxRecord` is created with an incremented commit version number, and the transaction's write set. The `trySetNext(ActiveTxRecord)` is a CAS operation that atomically sets the `commitTxRecord` as the next valid record after the `lastValid`. This is a tentative operation that only succeeds if the next record is still unset. Otherwise, it means that another transaction has won the race for that position, in which case validation resumes from the

```
1  class Transaction {
2    ...
3    void validateCommitAndEnqueue() {
4      ActiveTxRecord lastValid = this.activeTxRecord;
5      do {
6        lastValid = validate(lastValid);
7        this.commitTxRecord = new ActiveTxRecord(lastValid.txNumber+1,
8                                                 this.writeSet);
9      } while (!lastValid.trySetNext(this.commitTxRecord));
10   }
11
12   ActiveTxRecord validate(ActiveTxRecord lastChecked) {
13     ActiveTxRecord recordToCheck = lastChecked.getNext();
14
15     while (recordToCheck != null) {
16       if (!this.readSet.isEmpty() && !validFor(recordToCheck)) {
17         throw new CommitException();
18       }
19       lastChecked = recordToCheck;
20       recordToCheck = recordToCheck.getNext();
21     }
22     return lastChecked;
23   }
24
25   boolean validFor(ActiveTxRecord recordToCheck) {
26     for (VBox write : recordToCheck.getWriteSet()) {
27       if (this.readSet.contains(write)) {
28         return false;
29       }
30     }
31     return true;
32   }
33 }
34
35 class ActiveTxRecord {
36   ...
37   boolean trySetNext(ActiveTxRecord next) {
38     return next.compareAndSet(null, next);
39   }
40 }
```

Listing 4.1: The lock-free validation algorithm.

last known valid record. Thus, commit order is defined by the order in which transaction records enter the ATR list. All transactions that are in the process of committing must obtain a position in this list, after validating themselves. Even if not all transactions are already written back, being in this list enables future committing transactions to check their validity against all those that are already queued for a sure commit.

Notice that this validation algorithm is lock-free: Even though a single transaction may continuously fail to get a commit position, such fact implies that the transactional system as a whole must be making progress, i.e., other transactions are in fact validating and queueing their commit records. Other than the possible starvation caused by the other transactions progressing, there is no mechanism that prevents a transaction from validating itself, regardless of the state of the other transactions.

Additionally, if no commits occur between the start and the commit of a transaction that is attempting validation, then validation is trivial, because that transaction's `activeTxRecord` is the most recent one and `getNext()` returns `null`, allowing for the method `validate(ActiveTxRecord)` to quickly return without having to perform any iteration. Then, the committing transaction only needs to enqueue its commit record with the CAS.

4.2.2 Write Back

Any transaction that has been successfully validated by the previous algorithm, obtains a commit version, which is represented by the commit record that is placed in the commit queue. Figure 4.2 on page 103 shows an example of a possible state for the ATR list. Transactions 9 and 10 have already committed,[24] whereas transactions 11 to 13 are valid but not yet written back. At this point in time, any new transaction that begins will see the state that corresponds to version 10, because it is the last committed version. The figure also shows the two transaction records in use by transaction 12 (thus, assuming that transaction 12 started before transaction 10 had committed).

Writing back each value in the write set consists in adding a new `VBoxBody` to the corresponding VBox. The new `VBoxBody` is placed at the head of the list of bodies, pointing to the previous head. This new body contains the version obtained during validation and the value written by the transaction.

[24] The commit status of a write transaction is stored in a property of its commit record.

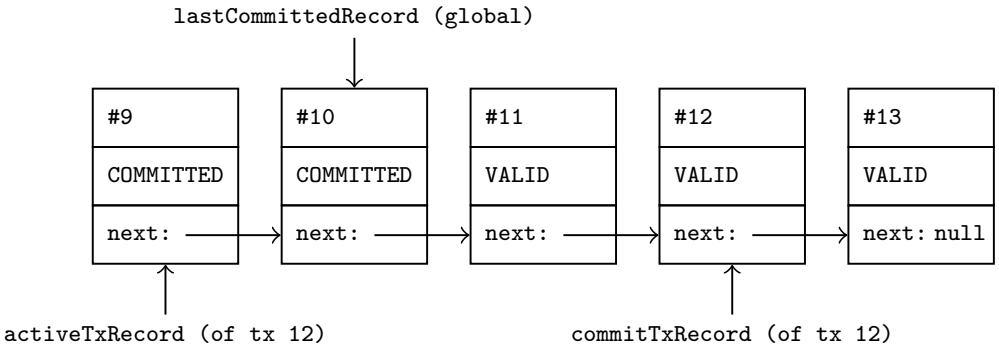

Figure 4.2: The ATR list extended to hold write transactions that are valid but not yet written back.

The JVSTM requires that, within each VBox, the list of versions is kept sorted by decreasing version number. This is necessary for correctness, but it also allows the reads to be fast, as the most often requested value is expected to be the first. If two valid transactions attempt to write back concurrently to the same location, then their writes need to be ordered according to their commit version. This problem did not exist in the lock-based version, because then only one transaction could be in the write back phase, at any given time.

Within the same transaction, the write back of each value can execute concurrently without synchronization, because each value is written to a different location. It is even possible to consider concurrent write backs between different transactions, as long as the written locations do not intersect. However, the latter adds complexity and cost to the algorithm: It requires the ability to determine whether two write sets intersect, which is an expensive operation.

To ensure that write backs to the same location are performed in order, I choose to write back only one transaction at a time, beginning with the oldest valid transaction that is yet to be written back (valid but not fully committed). The algorithm ensures that each transaction is completely written back before proceeding to the next in line. Each valid transaction will help to write back, starting from the oldest valid transaction up to, and including, itself. This way, instead of blocking, transactions that want to commit can be helpful and increase the overall write back throughput. Listing 4.2 on page 104 presents the method `ensureCommitStatus()` that is the entry point for the write back phase. This method, which is invoked immediately after a successful validation, is responsible for ensuring that all transactions up to `this` are written back and fully committed, by the end of the method's execution.

```
1  class Transaction {
2    ...
3    void ensureCommitStatus() {
4      ActiveTxRecord recToCommit = LAST_COMMITTED_RECORD.getNext();
5      while (recToCommit.txNumber <= this.commitTxRecord.txNumber) {
6        helpCommit(recToCommit);
7        recToCommit = recToCommit.getNext();
8      }
9    }
10
11   void helpCommit(ActiveTxRecord recToCommit) {
12     if (!recToCommit.isCommitted()) {
13       WriteSet writeSet = recToCommit.getWriteSet();
14       writeSet.helpWriteBack(recToCommit.txNumber);
15       finishCommit(recToCommit);
16     }
17   }
18 }
```

Listing 4.2: The method `ensureCommitStatus()` is the entry method for the write back phase. When it finishes it is guaranteed that all transactions up to, and including, the target transaction (`this`) have finished their commits.

The method `ensureCommitStatus()` starts by reading the shared reference `LAST_COMMITTED_RECORD` to get the next record to commit. Each iteration of the loop ensures the commit of the given record, i.e. it writes back the corresponding write set and marks the associated transaction as committed. The values to write back are evenly split among several buckets of approximately the same size each. This splitting is done so that the work can be distributed by any concurrent helpers. The write set itself holds the buckets, so `helpCommit(ActiveTxRecord)` delegates the write back operation to the write set instance, configuring it with the version number to use (line 14 of Listing 4.2).

The method `helpWriteBack(int)`, shown in Listing 4.3 on page 105, picks a random[25] bucket to start, and then iterates through each bucket once. For each bucket, it checks whether that bucket is already written back, by checking the `bucketsDone` array. This array contains an `AtomicBoolean` for each bucket that indicates whether the values in the corresponding bucket are already written back. Any helping transaction may find that some buckets are already written back, if some other transaction has helped. For those buckets that are not yet written back, this helping transaction will attempt to write each value to

[25]There is one instance of the randomizer class per thread, to eliminate contention on the random number generator.

```
1  class WriteSet {
2    ...
3    void helpWriteBack(int newTxNumber) {
4      int finalBucket = random.nextInt(this.nBuckets);
5      int currBucket = finalBucket;
6      do {
7        if (!this.bucketsDone[currBucket].get()) {
8          this.bodiesPerBucket[currBucket] =
9            writeBackBucket(currBucket, newTxNumber);
10         this.bucketsDone[currBucket].set(true);
11       }
12       currBucket = (currBucket + 1) % this.nBuckets;
13     } while (currBucket != finalBucket);
14   }
15
16   List writeBackBucket(int bucket, int newTxNumber) {
17     List newBodies = new LinkedList();
18     for (BucketEntry be : this.buckets[bucket]) {
19       VBoxBody newBody = be.vbox.commit(be.value, newTxNumber);
20       newBodies.add(newBody);
21     }
22     return newBodies;
23   }
24 }
```

Listing 4.3: Write back algorithm for a single write set.

```
1  class VBox {
2    ...
3    VBoxBody commit(Object newValue, int txNumber) {
4      VBoxBody currHead = this.body;
5      VBoxBody existingBody = currHead.getBody(txNumber);
6
7      if (existingBody == null) {
8        VBoxBody newBody = new VBoxBody(newValue, txNumber, currHead);
9        existingBody = CASbody(currHead, newBody);
10     }
11     return existingBody;
12   }
13
14   VBoxBody CASbody(VBoxBody expected, VBoxBody newBody) {
15     if (compareAndSwapObject(this, FIELD_OFFSET_BODY,
16                              expected, newBody)) {
17       return newBody;
18     } else {
19       return this.body.getBody(newBody.version);
20     }
21   }
22 }
```

Listing 4.4: Write back algorithm for each VBox.

its respective VBox, and mark the bucket as done. The method `writeBackBucket(int, int)`, iterates through the elements of the bucket invoking `commit(Object, int)` on each VBox with the value and version to commit.

Listing 4.4 presents the methods required at the VBox level. The method `commit(Object, int)` handles the creation of the new `VBoxBody` and inserts it at the head of the list of bodies. Due to concurrency in the write back operation, more than one thread can attempt to write back to the same VBox. As such, the `commit(Object, int)` operation starts by searching (via the method `getBody(int)`) whether there is already a `VBoxBody` with the required version. If such body is found, then it means that another helper transaction has already performed the commit of this value to the box. If the body is not found, then a new `VBoxBody` is created and the method `CASBody(VBoxBody, VBoxBody)` attempts to install it. This operation tries to replace the VBox's current `body` attribute with the new one, using a **single CAS attempt**.[26] If it succeeds, it returns the new

[26]The operation `compareAndSwapObject(Object obj, long offset, Object expected, Object newValue)` atomically replaces `expected` with `newValue`, if the field identified by `offset` in

head of the list. It if fails, then it is because a concurrent helping transaction succeeded, in which case it looks up the required `VBoxBody` to return. Recall that transactions are written back in order, so a failed CAS must necessarily mean that the required write already occurred.

The return value of the method `commit(Object, int)` is always the `VBoxBody` instance that corresponds to the value that should be written back, regardless of whether it was this helper transaction that actually did the write back. Each of these bodies is collected into the `newBodies` list (line 20 of Listing 4.3 on page 105), which, in turn, is stored in the `bodiesPerBucket` array, in the position corresponding to the bucket being written back (line 8). In Section 4.3 on page 112, I describe how these lists are used by the new GC algorithm to eliminate old versions from memory.

Finishing a commit requires publicizing the changes globally, and it must be done only after all values are in place. According to the Java Memory Model [Manson et al., 2005], any write to memory performed before a volatile write of a given location x will necessarily be seen by other threads, as long as they first perform a volatile read of x and the value written for x is seen. The setting of each `AtomicBoolean` in the `bucketsDone` array entails volatile write semantics, thus when another thread sees that a bucket is done (issuing a volatile read), it will surely be able to see the values of that bucket written back.

At the end of `helpWriteBack(int)` the helper transaction can be sure that all buckets of this write set are written back, because either this transaction wrote them itself or checked that some other transaction did it. So, the helper is sure that the helped transaction can be safely committed.

The write back phase is wait-free, because any transaction that enters this phase is certain to finish it in a finite number of steps, in spite of other threads executing any other code concurrently. There are three iterations:

- In the method `ensureCommitStatus()`: The number of iterations performed in this loop is bounded, because there is a finite number of records between the `LAST_COMMITTED_RECORD` and the transaction's own `commitTxRecord`, which was enqueued during the validation phase. Newer records that may enter the queue are added after the `commitTxRecord` and will not be considered in the loop.
- In the method `helpWriteBack(int)`: The number of buckets (`nBuckets`) is fixed

object `obj` contains the value `expected`, and it returns whether the replacement occurred.

per write set and the loop only passes through each bucket once.

- In the method `writeBackBucket(int, int)`: The number of entries per bucket is limited and defined when the bucket is created. There is only one pass per entry.

Finally, the CAS to set the new `VBoxBody` may fail, but in that scenario the written back value is already there, so the operation does not have to be retried. Also, when the CAS fails, finding the written back value to return is a bounded search, because the list of versions may only grow at the head, whereas the search is performed in the opposite direction.

4.2.3 Finishing a Commit

All transactions that help to write back a given transaction must try to finish that transaction, because no transaction really knows whether it was the first one to fully complete the write back, and it cannot depend on another to do it. Nevertheless, after completing the write back operation, making the changes globally visible is a trivial operation with two steps:

1. Set the commit status flag on the commit record.
2. Set the shared reference `LAST_COMMITTED_RECORD` to the record just committed.

These steps are performed in the method `finishCommit(ActiveTxRecord)`, which is shown in Listing 4.5 on page 109. The linearization point occurs when setting the commit flag. This is a volatile write to a boolean variable that is read by future transactions and, thus, guarantees that everything written by this transaction is visible by future transactions. The update of the shared reference to the last committed record is just a helping operation to reduce the search effort of new transactions.

The `finishCommit(ActiveTxRecord)` operation has no synchronization, so it is possible that thread interleaving will cause for some late thread to actually set the `LAST_COMMITTED_RECORD` reference back to a previously committed record, when there is already a more recent committed record. For this reason, whenever a new transaction begins, it obtains the most recent version from the `LAST_COMMITTED_RECORD`, and then it iterates forward, looking for the most recent **committed** record. Recall that a record enters the list immediately after validation succeeds, but before its write back phase, whereas in the lock-based algorithm, the list was updated, in mutual exclusion, only after each successful commit.

```
1 class Transaction {
2   ...
3   void finishCommit(ActiveTxRecord recToCommit) {
4     recToCommit.setCommitted();
5     LAST_COMMITTED_RECORD = recToCommit;
6   }
7 }
```

Listing 4.5: Finishing a commit and publishing changes.

4.2.4 Validation Revisited

As mentioned in Section 4.2.1 on page 100, validation of a transaction T can be performed in two different ways:

- Snapshot validation, which checks if T's read set is still up to date with the current global state. This corresponds to the original validation used in the lock-based commit.
- Delta validation, which incrementally checks each write set committed since T started and looks for an intersection with T's read set. This is the validation introduced for the new commit algorithm.

With regard to performance, either validation technique has advantages and shortcomings: Snapshot validation depends solely on the size of the read set. The time it takes to validate a read set is proportional to its size. Conversely, delta validation is independent of the size of the read set. This holds true as long as the lookup function that determines whether any given VBox is contained in the read set can be executed in constant time. However, delta validation depends on the size and number of all the other write sets committed while the transaction executed. So, delta validation will tend to perform better when the average size of the list of write sets to iterate multiplied by the average write set size is smaller than the average size of a read set to validate.

Whereas the size of the write set is application dependent, the number of the write sets to iterate grows with the number of write transactions. As the number of concurrent transactions increases, so does the cost of delta validation. To tackle this problem, I propose a new validation procedure, which mixes the two techniques. This mixed validation algorithm is presented in Listing 4.6 on page 110.

```
1  class Transaction {
2    ...
3    void validate() {
4      ActiveTxRecord lastSeenCommitted = helpCommitAll();
5      snapshotValidation(lastSeenCommitted.txNumber);
6      validateCommitAndEnqueue(lastSeenCommitted);
7    }
8
9    ActiveTxRecord helpCommitAll() {
10     ActiveTxRecord lastSeenCommitted = LAST_COMMITTED_RECORD;
11     ActiveTxRecord recToCommit = lastSeenCommitted.getNext();
12
13     while (recToCommit != null) {
14       helpCommit(recToCommit); //defined on Listing 4.2 on page 104
15       lastSeenCommitted = recToCommit;
16       recToCommit = recToCommit.getNext();
17     }
18     return lastSeenCommitted;
19   }
20
21   void snapshotValidation(int committedTxNum) {
22     int myReadNumber = getReadVersion();
23
24     if (committedTxNum == myReadNumber) { // immediately valid
25       return;
26     }
27
28     for (VBox read : this.readSet) {
29       if (read.body.version > myReadNumber) {
30         throw new CommitException();
31       }
32     }
33   }
34
35   // Adapted from validateCommitAndEnqueue(), first line excluded.
36   // (cf. line 3 of Listing 4.1 on page 101)
37   void validateCommitAndEnqueue(ActiveTxRecord lastValid) {
38     ActiveTxRecord lastValid = this.activeTxRecord;
39     ...
40   }
41 }
```

Listing 4.6: The mixed validation. The operation `validateCommitAndEnqueue` is adapted from Listing 4.1 on page 101, by removing the first line: The algorithm is the same, except that now it takes the last valid record as a parameter.

Before any validation attempt, the transaction first helps to write back all pending commits already queued. The method `helpCommitAll()` (line 9 of Listing 4.6 on page 110) is similar to the method `ensureCommitStatus()` (line 3 of Listing 4.2 on page 104), except that it helps to commit all queued records and it returns the last one that it helped to commit. Afterwards, a snapshot validation is performed. Notice that, because there is no synchronization, it is possible for other concurrent commit validations to occur, thereby increasing the tail of the ATR list beyond the `lastSeenCommitted` record. Nevertheless, the list of committed versions on each VBox surely must contain at least all the commits up to `lastSeenCommitted`. Thus, snapshot validation, at least, validates up to that point. But, even if snapshot validation succeeds, it is still necessary to ensure a valid commit position, and account for the possibility of concurrent commits meanwhile. So, the last step is to run the delta validation, but this time only from the `lastSeenCommitted` record onwards. The new `validateCommitAndEnqueue(ActiveTxRecord)` method is adapted from the original `validateCommitAndEnqueue()` (line 3 of Listing 4.1 on page 101) to receive the starting record, instead of using `activeTxRecord`.

According to experimental observation, read sets tend to be several times larger than write sets, even though this difference is highly dependent on each application's read and write patterns. As such, even though the mixed validation algorithm may be slower for a low number of transactions, it will scale much better, because it keeps the list of write sets to check small. Moreover, the initial step, in which the transaction helps to write back, is not wasted work as this would have to be done by some transaction anyway.

The method `helpCommitAll()` is lock-free, because a helper transaction may remain forever helping to commit newly enqueued transactions. The method `snapshotValidation(int)` is wait-free, because its loop is bounded by the size of the read set to validate. The method `validateCommitAndEnqueue(ActiveTxRecord)` is lock-free for the same reasons as `validateCommitAndEnqueue()`: Only the successful enqueueing of another transaction can cause a helper to have to retry. So, using the mixed validation algorithm, the validation phase remains lock-free as it was when using only the delta validation described in Section 4.2.1 on page 100.

In Listing 4.7 on page 112, I present the final commit algorithm, which combines the three commit phases that I have described. Taking into account the progress guarantees of each phase, the resulting algorithm is lock-free.

```
1  class Transaction {
2    ...
3    void commit() {
4      if (isWriteTransaction()) {
5        validate();
6        ensureCommitStatus();
7      }
8    }
9
10   boolean isWriteTransaction() {
11     return !writeSet.isEmpty();
12   }
13 }
```

Listing 4.7: The top-level method in the lock-free commit.

4.3 Asynchronous Elimination of Unused Versions

After replacing the original commit algorithm with the one that I have just described, all operations that the JVSTM implements are now nonblocking and they ensure system-wide progress—the JVSTM is completely lock-free. This includes the original GC algorithm for removing old versions, which I overviewed in Section 3.3.1 on page 38. However, during the initial tests that I performed while developing the lock-free commit algorithm, I discovered a performance bottleneck in the GC algorithm, which still prevented the scalability of the transactional system. In the following I present the problem and describe the replacement GC algorithm that I developed to address the problem.

In short, the problem is that the shared counter used in each ATR does not scale when used intensively. The following factors contribute to the bottleneck:

- Very short transactions.
- Read-dominated workloads.
- Increased number of concurrent transactions in many-core machines.

Given that every transaction increments/decrements these counters at the beginning/end, the more transactions there are, the more concurrent updates there will be to the counters, which is worsened when the number of transactions that can effectively run in parallel increases. Also, if the transactions are short, the percentage of time spent beginning

and finishing increases, when compared to the time spent in the transaction's body. Most importantly, in scenarios where read-only transactions prevail, most transactions will be using the same version, thus trying to modify the same counter. I verified, experimentally, that the current mechanism for cleaning old versions causes a performance degradation in read-dominated scenarios, when there are above 20 short transactions running in parallel.

Taking this into account, I developed a new algorithm to eliminate unused versions, which runs in a separate thread and still uses the ATRs, but does not require the records to have any counters. The general idea is to inform the GC task about all transactions currently running, and what is the record currently in use by each of these transactions. With this information, the algorithm can identify asynchronously which versions are no longer accessible, and eliminate them. Instead of keeping an updated list with an entry per transaction, the system keeps a list with an entry per thread: This design is best under the assumption that the creation of a new thread occurs fewer times than the creation of a transaction, and leverages on the fact that each thread executes transactions sequentially. In the rest of this section I present the details for the lock-free GC algorithm.

The `TxContext`, shown in Listing 4.8 on page 114, is a **per-thread transaction context** that holds a reference (`oldestRequiredRecord`) to the record currently in use by the thread's running transaction. Each thread's context is accessible via the thread-local variable `threadTxContext`. There is also a shared list of all existing contexts (`allTxContexts`): The GC task (described ahead) repeatedly iterates through this list, identifying which is the oldest record in use by any transaction, and cleaning all records from the last cleaned up to the oldest in use.[27] Each context also contains a Java `WeakReference` to the thread that created it (`owner`). This reference allows the GC task to detect when a thread has died, so that unnecessary contexts can be removed from the shared list.

Adding a new `TxContext` to the shared list is a simple add operation on a nonblocking list. This needs to be done **only once for every new thread that starts a transaction for the first time** and is automatically guaranteed in the initialization method (`initialValue()`) of the `threadTxContext`. Afterwards, each thread reuses its own context, regardless of the transaction that is being executed in the thread. **When a new transaction starts**, it determines its read version, and **it must set the corresponding ATR in the context's `oldestRequiredRecord`**. **When a transaction finishes it must set that record to `null`**. This is fundamental to allow the GC algorithm to compute which is the oldest record in use—

[27] The shared list of contexts is initially populated with a context that is not associated with any thread, so that list is never empty, and the GC algorithm will always keep at least one context in the list.

```
1  class TxContext {
2
3    // A TxContext per thread
4    static final ThreadLocal<TxContext> threadTxContext =
5            new ThreadLocal<TxContext>() {
6      @Override protected TxContext initialValue() {
7        TxContext newCtx = new TxContext(Thread.currentThread());
8        allTxContexts.enqueue(newCtx);
9        return newCtx;
10     }
11   };
12
13   // All TxContexts available to the GC task.
14   // The GC algorithm requires this list to be non-empty,
15   // so the first context is special and is explicitly added.
16   static TxContext allTxContexts = new TxContext(null);
17
18   volatile ActiveTxRecord oldestRequiredRecord;
19   WeakReference<Thread> owner;
20   TxContext next;
21
22   TxContext(Object owner) {
23     this.owner = new WeakReference(owner);
24   }
25
26   void enqueue(TxContext ctx) {
27     TxContext current = this;
28     while (true) {
29       for (; current.next != null; current = current.next);
30
31       if (compareAndSwapObject(current, FIELD_OFFSET_NEXT,
32                                null, ctx)) {
33         return;
34       }
35     }
36   }
37 }
```

Listing 4.8: The `TxContext`. Its main goal is to keep a per-thread registry of the oldest record required by the transaction running in each thread.

```
1  class GCTask implements Runnable {
2    ...
3    ActiveTxRecord lastCleanedRec;
4
5    GCTask(ActiveTxRecord lastCleanedRec) {
6      this.lastCleanedRec = lastCleanedRec;
7    }
8
9    void run() {
10     while(true) {
11       ActiveTxRecord rec = findOldestRecordInUse();
12       cleanUnusedRecordsUpTo(rec);
13     }
14   }
15 }
```

Listing 4.9: Entry point for JVSTM's new GC algorithm.

the oldest required record—by all transactions.

Due to the concurrency between the GC algorithm and the start of a new transaction, after setting the `oldestRequiredRecord` a transaction must check again whether there is a newer record, and, if so, upgrade to it, simply by setting it as the `oldestRequiredRecord`. This double-check sequence is required to ensure that a transaction never uses a record that may have been removed concurrently by the GC. Notice that a write transaction that commits will mark its `commitTxRecord` as committed (line 4 of Listing 4.5 on page 109), before setting the `oldestRequiredRecord` to `null`. This ensures that, if the GC task decides that a record is inaccessible, then new transactions will have to see the newer record, and use it. The GC algorithm is summarized in Listing 4.9.

The GC task is launched in a *daemon* thread that runs an infinite loop: It first finds the oldest record in use, and then it cleans all records up to that record. The GC task is created before the JVSTM begins to process transactions, so the `lastCleanedRec` is initialized with a sentinel record that represents the initial state of the transactional system. The `lastCleanedRec` will advance as the GC cleans up old records.

The algorithms for `findOldestRecordInUse()` and `cleanUnusedRecordsUpTo()` are shown in Listing 4.10 on page 116, and they depend on the following invariant:

Apart from changing to `null`, the `oldestRequiredRecord` for each con-

```
1  class GCTask implements Runnable {
2    ...
3    ActiveTxRecord findOldestRecordInUse() {
4      ActiveTxRecord alreadyCommitted = LAST_COMMITTED_RECORD;
5      for (ActiveTxRecord next = alreadyCommitted.getNext();
6           (next != null) && next.isCommitted();
7           next = next.getNext()) {
8         alreadyCommitted = next;
9      }
10
11     ActiveTxRecord oldRec_1 = getOldestRecord(Integer.MAX_VALUE);
12     if (oldRec_1 == null) {
13       return alreadyCommitted;
14     }
15
16     ActiveTxRecord oldRec_2 = getOldestRecord(oldRec_1.txNumber);
17     return (oldRec_2 != null) ?
18             oldRec_2 : oldRec_1;
19   }
20
21   ActiveTxRecord getOldestRecord(int minVersion) {
22     ActiveTxRecord oldRec = null;
23     TxContext currentCtx = TxContext.allTxContexts;
24
25     while (currentCtx != null) {
26       ActiveTxRecord rec = currentCtx.oldestRequiredRecord;
27       if ((rec != null) && (rec.txNumber < minVersion)) {
28         minVersion = rec.txNumber;
29         oldRec = rec;
30       }
31       currentCtx = currentCtx.next;
32     }
33     return oldRec;
34   }
35
36   void cleanUnusedRecords(ActiveTxRecord upToThisRec) {
37     while (this.lastCleanedRec.txNumber < upToThisRec.txNumber) {
38       final ActiveTxRecord toClean = this.lastCleanedRec.getNext();
39       this.lastCleanedRec = toClean;
40       cleanersThreadPool.execute(new Runnable() {
41         public void run() {
42           toClean.clean();
43         }});
44     }
45   }
46 }
```

Listing 4.10: Algorithm for detection and cleaning of unused records.

> **text never changes backwards—that is, after being set to a record R_a the `oldestRequiredRecord` will no longer be set to another record R_b, such that R_b`.txNumber` $< R_a$`.txNumber`.**

This invariant ensures the GC that once a transaction has set a record R in its context it will never need to access another record older than R. This is guaranteed to be so, because a transaction always starts on the latest committed record and never downgrades to an older version. Therefore, after a transaction T_1 finishes, any future transaction T_2 that executes on the same thread will necessarily use a version that is not older than T_1`.getReadVersion()`.

With the invariant in mind, the method `findOldestRecordInUse()` starts by computing the most recent record `alreadyCommitted` (lines 4 to 9 of Listing 4.10 on page 116) by reading the current `LAST_COMMITTED_RECORD` and advancing as much as possible through the list. Should the remainder of the method fail to find any record, it is safe to clean up to the `alreadyCommitted` record, because any new transaction will necessarily start from that record onwards.

The search for the oldest record in use requires two passes over the list of contexts to get the oldest record of each pass. The method `getOldestRecord(int)` returns the record with the lowest `txNumber` from the list, or it returns `null` if it does not find any record in the list of contexts with a number lower than `minVersion`. If the first pass returns `null`, then the `alreadyCommitted` record is the safest point to clean up to, and the method returns. Otherwise, a second pass is executed to ensure that, due to concurrency, it did not miss a thread that was starting a transaction with a record older than `oldRec_1`, which could occur for any context C_i read before reading the context C_{min} in which `oldRec_1` was found. Such event is only possible if C_i`.oldestRequiredRecord == null`. The case of C_i`.oldestRequiredRecord.txNumber` $<$ C_{min}`.oldestRequiredRecord.txNumber` is impossible and for C_i`.oldestRequiredRecord.txNumber` $\geq$ C_{min}`.oldestRequiredRecord.txNumber` a second pass will never detect a record older than C_{min}, because of the invariant. The second invocation of `getOldestRecord(int)` can be optimized, because actually it only needs to check the contexts up to C_{min}.[28] For clarity of the algorithm, such optimization is not shown in the code in Listing 4.10 on page 116. In the end, after the second pass, if a record older than `oldRec_1` was found, then that record is returned; otherwise `oldRec_1` is the oldest.

After the identification of the oldest record in use, the method

[28] Any significant context ahead would be detected in the first pass.

`cleanUnusedRecordsUpTo(ActiveTxRecord)` simply iterates and cleans all records from the last one cleaned (`lastCleanedRec`) up to the oldest in use (`upToThisRec`). It uses a pool of worker threads to do the actual cleaning work, handing off one record to clean to each thread in the pool. Cleaning each record (line 42 of Listing 4.10 on page 116) is performed as initially described in Section 3.3.1 on page 38: The only change required is that the list of bodies to iterate is stored in the `bodiesPerBucket` array, which was produced during the write back phase of the commit, and is accessible directly through the record (see line 8 of Listing 4.3 on page 105).

The method `getOldestRecord(int)` performs an additional task, which I did not show until now to keep the presentation of the GC simpler: Given that the method already iterates over all contexts, I use it also to identify and remove from the list those contexts that are no longer associated with a thread. Such removal can occur only after a thread has been destroyed. This is detected, because the `WeakReference` in the slot `owner` becomes `null`. The full algorithm for the method `getOldestRecord(int)` is shown in Listing 4.11 on page 119: It keeps an additional reference to the context preceding the one being iterated and, in case it detects that the current context is no longer owned by a thread, it removes it by updating the previous reference to point to the context after the current. The removal only occurs if it can ensure that the list of records will not be left empty (line 10 of Listing 4.11 on page 119).

The GC algorithm that I present here does not use counters to keep information about which records are still required, and it executes on threads that run separate from those running the transactions. The biggest improvement it provides is that contention is virtually eliminated when starting and finishing a transaction: Transactions just write to the `oldestRequiredRecord` of the **context of their own thread** in addition to reading from or writing to the `LAST_COMMITTED_RECORD`, which is negligible when compared to the overheads of the previous algorithm, caused by the races to atomically update the counters. Conversely, because it runs asynchronously with the transactions, this algorithm may not eliminate unused versions as soon as the previous algorithm did and, therefore, it can cause memory to be used for longer than necessary. Also, it will require more memory to maintain the data structures required by the GC.

The GC algorithm is lock-free and all other operations of the JVSTM remain, at least, lock-free. The entire GC algorithm is nonblocking and due to its asynchronous nature, it does not interfere with the progress of the threads running transactions. The exception is the first transaction that each thread executes: The trans-

```
1  class GCTask implements Runnable {
2   ...
3   ActiveTxRecord getOldestRecord(int minVersion) {
4      ActiveTxRecord oldRec = null;
5      TxContext currentCtx = TxContext.allTxContexts;
6      TxContext previousCtx = null;
7
8      while (currentCtx != null) {
9        if ((currentCtx.owner.get() == null)
10           && (currentCtx.next != null)) {
11         removeCtx(previousCtx, currentCtx);
12         currentCtx = currentCtx.next;
13         continue;
14       }
15
16       ActiveTxRecord rec = currentCtx.oldestRequiredRecord;
17       if ((rec != null) && (rec.txNumber < minVersion)) {
18         minVersion = rec.txNumber;
19         oldRec = rec;
20       }
21       previousCtx = currentCtx;
22       currentCtx = currentCtx.next;
23     }
24     return oldRec;
25   }
26
27    void removeCtx(TxContext previous, TxContext toRemove) {
28      if (previous == null) { // 'toRemove' is the first record
29        TxContext.allTxContexts = toRemove.next;
30      } else {
31        previous.next = toRemove.next;
32      }
33    }
34 }
```

Listing 4.11: The detection and removal of destroyed threads occurs while iterating the contexts' list. The code in gray is the same from Listing 4.10 on page 116.

action will run the method `initialValue()` when trying to get the thread's context to update its `oldestRequiredRecord`. In turn, this will run the lock-free method `enqueue(TxContext)`, which can contend with other threads that are also running their first transaction. The GC algorithm is not wait-free, because it performs iterations that may suffer from starvation:

- In line 5 of Listing 4.10 on page 116, records are appended to the list when write transactions commit.
- In line 25 of Listing 4.10 on page 116, contexts are appended to the list when new threads run their first transaction.

The iteration performed in line 37 of Listing 4.10 on page 116 is bounded, and the work performed by the threads spawned to clean each record is also bounded by the number of bodies to clean. Nevertheless, the main GC thread does not even wait for those threads to complete: It is a fire-and-forget mechanism.

4.4 Experimental Evaluation

In this section, I evaluate the performance of the lock-free JVSTM from two different perspectives:

- First, I compare it with the original lock-based version. This evaluation shows that, as the lock contention starts to become a problem, using the lock-free commit can improve the overall application performance. This is best seen when the duration of the commit amounts to a relevant part of the overall transaction.
- Second, I compare it with other top-performing STM implementations, namely LSA and TL2, which use blocking algorithms. Here, the lock-based implementations still attain the overall best performance, especially for a lower thread count. However, in spite of not being the top-performer for a small number of threads, as the parallelism increases the JVSTM improves performance whereas the others decrease, leading to an identical performance of all STMs, in some workloads.

4.4.1 Lock-Free Versus Lock-based JVSTM

I use three benchmarks to evaluate the scalability of the lock-free JVSTM. For each one, I measure the overall application performance (time to complete or throughput, depending on the benchmark), the average commit time per transaction, and the total number of transaction restarts due to conflict. These tests were executed on an Azul Systems' Java Appliance[29] with 208 cores, running a custom Java HotSpot(TM) 64-Bit Tiered VM (1.6.0_07-AVM_2.5.0.5-5-product), and using from 1 up to 256 threads, for both the lock-based and the lock-free implementations of the JVSTM. I first provide a description of each benchmark used, together with its relevant characteristics for this evaluation. Then, I present the results and discuss them.

4.4.1.1 Description of the Benchmarks

Array. This benchmark is a custom micro-application that I designed specifically to test the JVSTM. It is highly configurable, and very useful to stress test the JVSTM, but it does not implement any real-world behavior. It can be configured through several execution parameters, such as the number of transactions to execute, how many of them can run concurrently, the number of reads and writes per transaction, the number of read-only transactions, write transactions, and so forth. The application initializes an array of integer transactional locations with a given size, and then executes a set of transactions that can read from and write to random positions of the array. This benchmark is useful because the body of a transaction can be finely tuned and comprises only transactional accesses, meaning that the results are unaffected by complex application logic: As such, this benchmark is more sensitive to changes in the JVSTM.

The goal is to use the Array benchmark to simulate a highly concurrent and write-intensive scenario: Given a conflict rate sufficiently low, each transaction should have a high probability of committing without the need to restart. To do so, I configure the application to create an array with 10^6 positions. The benchmark runs to completion 10^4 read-write transactions, where each transaction randomly reads from 10^3 positions and randomly writes to 10 positions. In fact, as the number of concurrent transactions increases, it is expected that the conflicts also increase, even if only slightly.

[29] http://www.azulsystems.com/

Lee-TM. The Lee-TM benchmark [Watson et al., 2007] implements the Lee algorithm, applied to automatic circuit routing and it measures how long it takes to lay down tracks on a circuit board. Each iteration of the algorithm—laying a track between two points—has two phases: In the **expansion phase** it performs a breadth-first search to mark all positions with their relative distance to the starting point, until it reaches the destination point. Then, the **backtracking phase** connects the destination to the origin, via the shortest path that was found during the expansion. This benchmark has a Java implementation[30] that models the circuit board as two shared bi-dimensional arrays of integers (one array for each side of the board). Each transaction attempts to establish a route between two points of the board. The transactions read from the shared array during the expansion phase, but mark the distances in a transaction-local copy of the array. The shared array is modified only during the backtracking phase to mark the track. To use the JVSTM I simply had to wrap each position of the two shared arrays with a VBox.

Even though there is potentially high parallelism in this application, the way that the Lee algorithm works causes many false conflicts between transactions: During the expansion phase, the transactions read a large number of positions from the shared board, causing them to conflict with any concurrent transaction that has laid a track over one of the positions read. However, there is a true conflict only if the tracks cross each other, i.e., if the board positions used during the backtracking phase intersect.

The solution proposed to eliminate these false conflicts [Watson et al., 2007] is to perform the *early release* [Herlihy et al., 2003b] of some positions from the read set: Taking into account that during the backtracking phase the algorithm also reads the positions over which it backtracks, it is possible to remove from the read set of the transaction all positions used during the expansion phase and just consider, at commit time, the ones that are added during the backtracking phase. The JVSTM provides an API that is ideal for this situation. The method `commitAndBegin()` atomically commits the current transaction and begins a new one: If the committing transaction is a read-only transaction, this operation is equivalent to a complete release of the read set up to the point of the `commitAndBegin()` operation. In effect, because the expansion phase writes to local data only, this phase corresponds to a read-only transaction in the JVSTM, which is sure to never fail. Thus, to eliminate false conflicts and improve the benchmark's scalability, the JVSTM implementation of the Lee-TM benchmark additionally uses transactions in the form shown in Listing 4.12 on page 123. With this modification, this benchmark is expected to represent a possible

[30] http://apt.cs.man.ac.uk/projects/TM/LeeBenchmark/

```
1  class LeeRouter {
2    ...
3    void connect(...) {
4      boolean success = false;
5      try {
6        Transaction.begin(READ_ONLY);
7        // expansion phase
8        Transaction.commitAndBegin(READ_WRITE);
9        // backtracking phase
10       success = true;
11     } finally {
12       if (success) {
13         Transaction.commit();
14       } else {
15         Transaction.abort();
16       }
17     }
18   }
19 }
```

Listing 4.12: Pseudocode for the method that connects two points in the Lee-TM benchmark. The transactional operation is implemented in JVSTM using an atomic `commitAndBegin()` operation to perform the early release of the read set and, thus, avoid false conflicts during the backtracking phase.

real-world application that is highly scalable even though all concurrent operations perform updates.

For my tests with Lee-TM, I used the memory circuit board (*memboard*) described in [Ansari et al., 2008]. This is the most complex circuit available in terms of number of routes to process, modeling a 600 by 600 circuit board with 3101 routes to establish.

STMBench7. This is another benchmark commonly used to evaluate STM implementations. The STMBench7 [Guerraoui et al., 2007] simulates the workload of a real-world object-oriented application: The benchmark implements a shared data structure, consisting of a set of graphs and indexes that model the object structure of complex applications. It supports 45 different operations, varying in category from simple to complex, and three predefined workloads with different read/write ratios: read-dominated (90/10), read-write (60/40), and write-dominated (10/90). The performance measurement is the throughput, given as the average number of operations executed per second.

The adaptation of this benchmark to the JVSTM, initially by [Cachopo, 2007] and then

by [Rito and Cachopo, 2010], replaces the shared data structures with their equivalent implementations, using VBoxes to hold the shared values. Additionally, the benchmark manipulates collections of objects transactionally: These collections are implemented with purely functional data structures and there is a single VBox to hold the instance of a collection.

This benchmark is very difficult for STMs, because of its low scalability. The most complex operations either cause structural modifications to the object graph or perform long traversals, reading nearly all the objects. As I want to increase the potential for parallelism in the write-dominated scenario, I run the tests with structural modifications and long traversals disabled. Nevertheless, as the results will show, the benchmark still exhibits a high conflict rate in the write-dominated workload, preventing scalability.

4.4.1.2 Results

Array. Before delving into the results obtained by each benchmark with the Azul System's JVM, let us first revisit the motivating example from Section 4.1 on page 96. There, I showed in Figure 4.1 on page 98 the scalability limitations of the single-lock approach by increasing the duration of the commit algorithm in the Array benchmark. In Figure 4.3 on page 125, I present again the previous results, together with the results obtained when using the lock-free JVSTM under the same test conditions. In both versions of the JVSTM the single-thread executions take roughly the same time to complete. This is expected, because all the transactions are executed in sequence, and the added delay affects all transactions equally. However, as soon as the number of concurrent threads starts to increase, the differences between the two algorithms become clearly noticeable: In the lock-free JVSTM, the time to complete all the operations of the benchmark reduces continuously as the number of concurrent threads increases, and it tends towards the baseline time. In fact, when 10,000 transactions execute in 32 concurrent threads, there will be an average of 312.5 transactions executed per thread. These will still execute in sequence, so that a 1ms delay in each, will cause an overall delay to complete of at least 312.5ms, which is barely noticeable in the plot. These results demonstrate that, as expected, when the commits of write transactions overlap each another, the lock-free design is required to ensure scalability of the application.

Now, let us turn to the results obtained in Azul System's JVM. Figure 4.4 on page 126 displays the results for the Array benchmark. Bear in mind that Azul's machine has many more cores, but each core is much slower than each of the 48 cores used before, so that total time to complete will start much higher. In both the lock-based and lock-free versions,

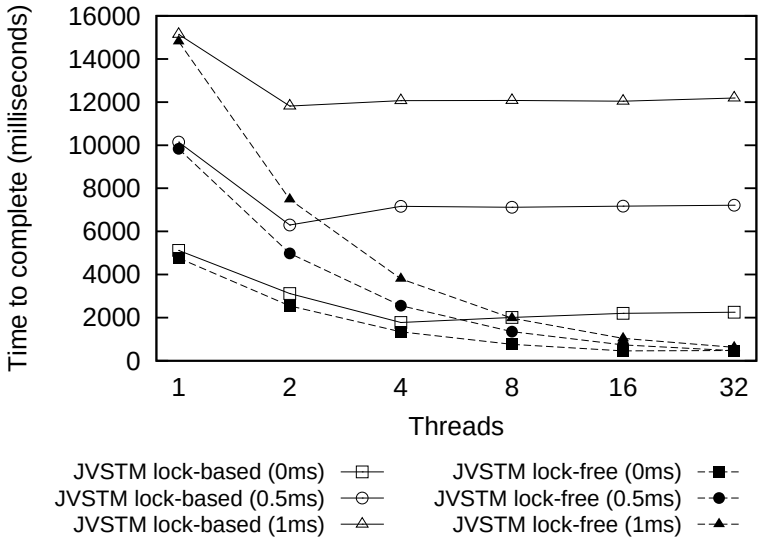

Figure 4.3: Comparison of the commit algorithms using added delays. Unlike the lock-based commit, the nonblocking version allows the application to scale when more threads are added.

the time to complete the benchmark initially decreases as the number of threads increases, meaning that both versions are being able to process concurrent transactions in parallel. However, in the lock-based version, the time to complete starts to increase when running more than 8 threads, whereas in the lock-free version it remains low. This phenomenon coincides with the point in which the average commit time of the lock-based version starts to increase exponentially.

Apparently, above 8 threads, the number of concurrent transactions has become high enough, so that their commits overlap one another, causing the transactions to have to wait increasingly longer to acquire the lock. I draw this conclusion based on the following three observations taken from the results:

- In the lock-based version, the time to complete increases above 8 threads.
- In both versions, the number of restarts is identical, up to 64 threads, meaning that the probability of conflict is also identical.
- In the lock-based version, the average time to commit increases exponentially[31] above 8 threads, e.g. at 64 threads this time in the lock-based version is already over a hundredfold that of the lock-free version.

[31] In all benchmarks, the plots for the average commit time use a logarithmic scale for the vertical axis.

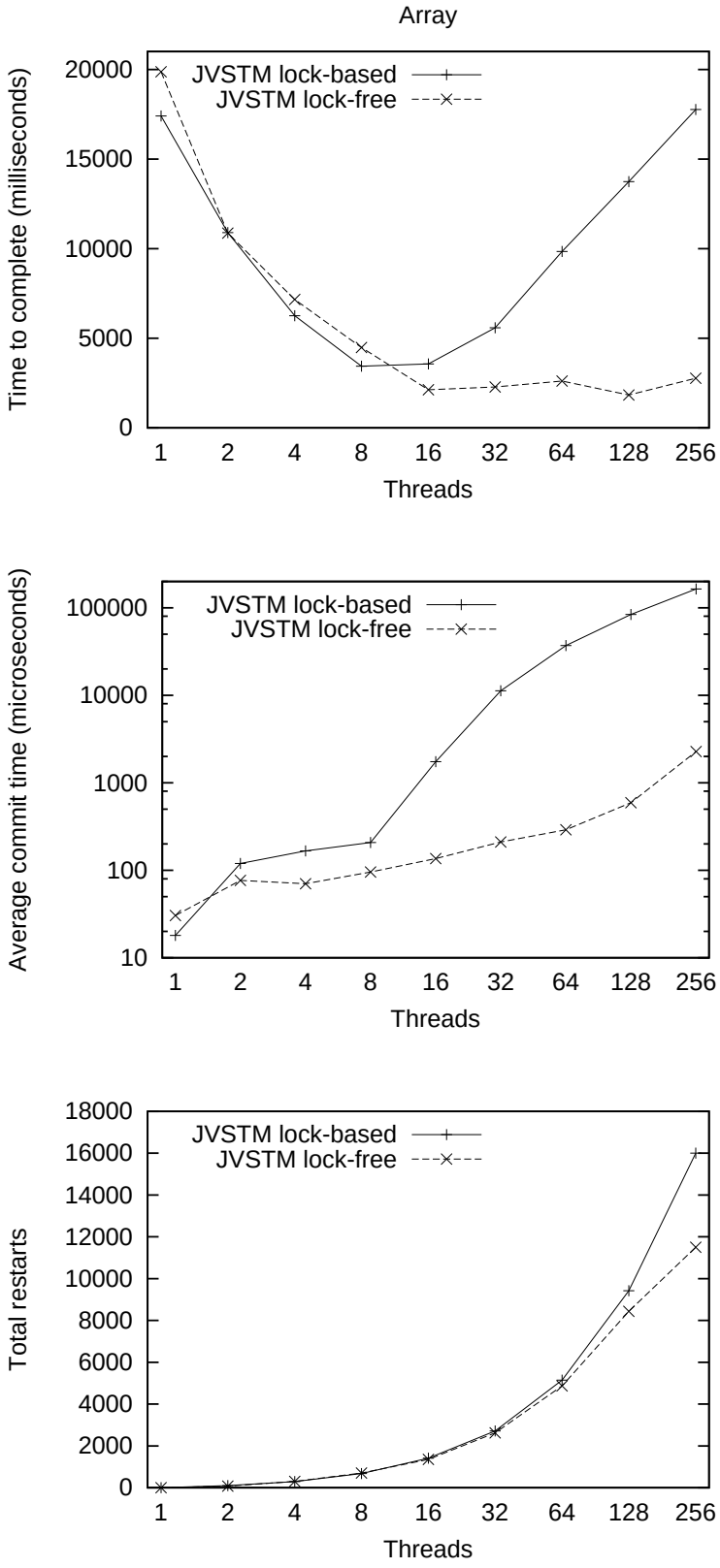

Figure 4.4: The results for the Array benchmark in the Azul VM.

For 128 threads and above, the difference in the total number of restarts tends to increase, with the lock-based version having approximately 40 percent more restarts than the lock-free. This is caused by the effect of lock convoying that I described in Section 4.1.2 on page 96. Notice that, for 256 threads, multiprocessing is necessarily involved, because the test machine has 208 cores. The added cost of context switching, and even the probability that a thread holding the commit lock may be preempted, contribute to increase the lock time of each commit, with the associated pernicious effects I already mentioned.

In the lock-free commit algorithm, each transaction can perform validation independently of the others. This means that invalid transactions can detect their condition much earlier than if they had to wait for the commit lock: In the lock-based version, not only do these transactions take longer to restart, they also take up lock-time preventing other transactions from committing. Moreover, because the validation includes the helping mechanism, which writes back pending commits ahead in the queue, it makes the restarted transactions see a more recent version than they would if they simply restarted without helping, which improves the odds of a successful commit after the restart.

Also in the lock-free version, the lowest time to complete the benchmark is obtained at 128 threads. In spite of that, the times to complete change very little above 16 threads: At first glance, this is unexpected, because, ideally, the lock-free version should continue to improve, at least up to the number of physical processors available—some increase at 256 threads is expected, as there is already context switching involved. This benchmark performs a very high rate of memory accesses, which could cause it to reach the limits of the machine's available memory bandwidth. Apart from the possibility of a physical limitation, however, there are two factors contributing to this behavior:

- As the number of concurrent transactions increases, so does the size of the commit queue—the number of transactions to write back in the ATR list—which potentially increases the amount of work that each transaction needs to do in the helping algorithm. This affects the validation and the write back phases of the commit.
- In this workload the probability of conflict increases with the number of transactions. This can be seen in the increasing number of restarts. So, in fact, the application needs to perform more transactions due to restarts, as the concurrency increases.

Both aspects contribute to prevent the further reduction in the time to complete the workload for the lock-free version. In Table 4.1 on page 128 I present the average number

	Transactions/second by number of threads								
JVSTM version	1	2	4	8	16	32	64	128	256
Lock-based	574	927	1644	3101	3202	2276	1539	1413	1463
Lock-free	503	928	1438	2389	5335	5529	5696	10047	7763

Table 4.1: Total transactions per second in the Array benchmark. It includes the transactions processed that had to restart due to a conflict.

of transactions per second that each version is capable of processing, **including** the transactions that restart due to conflict. This provides an idea of the relative performance of the two JVSTMs: In the lock-free version, the number of transactions per second increases up to 128 threads—this value only drops when running with more threads than the number of available processors. The performance of the lock-free version peeks at 128 threads, processing just over seven times more transactions per second than the lock-based version, and clearly taking more advantage of the parallel processing power available than its lock-based counterpart. Taking into account that the workload is the same, the reason why the lock-based version processes so many fewer transactions is due to the time it spends waiting for the lock acquisition.

Lastly, the overhead of the new algorithm is visible for low thread counts: Up to 8 threads, the lock-based version is capable of processing more transactions per second, which results in a lower time to complete the benchmark. This behavior is expected: While contention on the commit lock is low or even inexistent (as is always the case for a single thread), the lock-based version should perform better, because its commit algorithm is shorter.

Lee-TM. The workload in this benchmark is quite different from the Array: One distinguishing feature, in the Lee-TM, is that each transaction performs transaction-local computations instead of strictly accessing shared state, which causes the proportion between the duration of the transaction body and its commit to be much higher in the Lee-TM than in the Array. Also, the commits tend to be small because, even though the transactions are relatively big, the final size of their read and write sets is only the number of positions in the circuit board that are required for connecting two points (due to the early release of most of the read set). This means that the improvements to the commit phase brought about by the lock-free implementation are much less visible in the overall performance measurements of this benchmark.

In fact, the results for the Lee-TM benchmark, present in Figure 4.5 on page 129, show

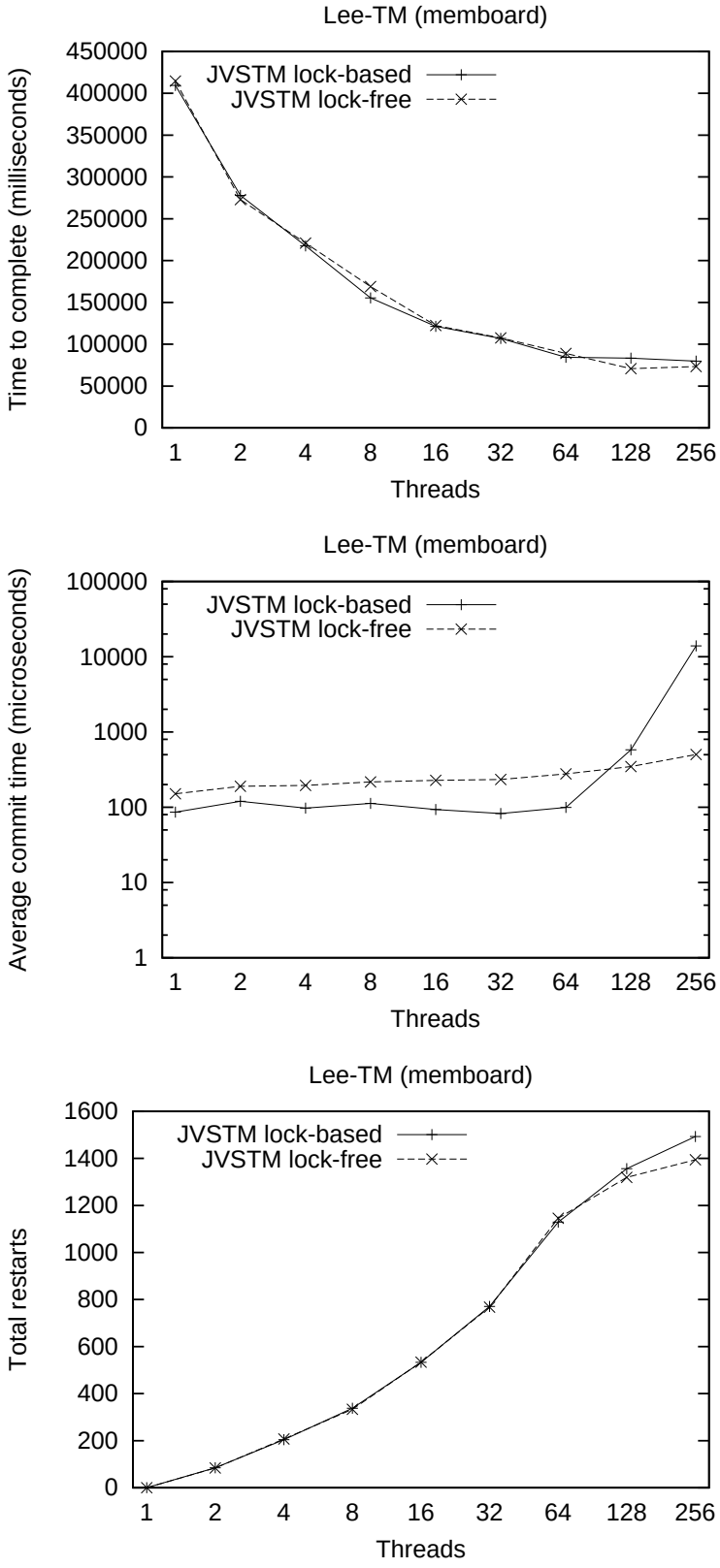

Figure 4.5: The results for the Lee-TM benchmark in the Azul VM.

that both versions of the JVSTM display similar times to complete for the same number of threads. As with the previous benchmark, however, when considering only the average time to commit, the lock-based version starts with a lower commit time (up to 64 threads, in this workload), and then, the commit time grows exponentially, whereas in the lock-free the commit time remains almost the same. Like with the previous benchmark, there is clearly a threshold for how much contention the lock-based commit can take: After a certain level of contention is reached (which can vary with the application), the duration of the commit in the lock-based version skyrockets.

There is also another factor that affects why the exponential growth in the commit time is not noticeable in the plot that shows the total time to commit. In Lee-TM the transactions are not uniform in size, and the longest paths to lay correspond to the longest running transactions, because the breadth-first search needs to go much deeper. The longest track to lay dominates the benchmark and, as such, no matter the amount of parallelism available, the benchmark will never take less time to complete than the time it takes to lay the longest track. For the memboard circuit, the longest track takes between 60 and 70 seconds to lay (accounting for any restarts necessary) and the best time to complete the entire benchmark corresponds to the lock-free version at 128 threads: about 71 seconds. So, the scalability of this benchmark is limited, and it is not expected that increasing the number of concurrent transactions will allow for further reducing the time it takes to complete the benchmark.

The number of restarts is also very similar between the two JVSTMs, as in the previous benchmark, again showing some reduction (this time less significative) in the lock-free version for the largest number of threads. This seems to suggest a correlation between the average commit time and the probability of conflict: When the average commit time increases exponentially, the probability of conflict also increases, leading to more restarts. This is in agreement with the idea that lock convoying contributes to performance degradation.

STMBench7. The results for this benchmark are available in Figure 4.6 on page 132. The most relevant difference to the previous benchmarks is that neither version of the JVSTM is able to scale when the number of threads increases. The STMBench7 is generally a highly conflicting benchmark, and the write-dominated workload presents a very high number of restarts, even for low thread counts.

Table 4.2 on page 131 shows the average percentage of restarts. It takes into account that the same transaction might restart several times before successfully committing: Each percentage corresponds to the ratio between the number of executions due to a restart and

	Percentage of transaction restarts by number of threads								
JVSTM version	1	2	4	8	16	32	64	128	256
Lock-based	0	6.95	18.56	35.62	54.48	69.35	79.26	87.98	94.4
Lock-free	0	7.06	18.96	36.06	54.18	69.58	79.68	87.54	91.51

Table 4.2: Percentage of restarts in the STMBench7 benchmark. The percentage of restarts in the write-dominated workload increases very rapidly and it prevents scalability.

the total number of transaction executions (including the restarts). At 16 threads already more than half the transactions executed correspond to a transaction restart and at 256 threads almost all transactions executed (above 90 percent) are the result of a restart. Every time spent performing a transaction that restarts is wasted work for the measurement of the overall throughput, so these values explain the decrease in performance for a higher number of threads. Unlike in the Array benchmark, in STMBench7 contention in the commit lock is a lesser problem, because of the amount of restarts caused by conflicting operations.

Like in the Lee-TM benchmark, due to the proportion between the size of the transaction's body and its commit, the effects of the increased average commit times, occurring above 32 threads, are barely visible in the overall throughput plot. Still, all the aspects that I have mentioned in the previous benchmarks, comparing the lock-based with the lock-free version still apply here also, namely:

- In the lock-based version, the average commit time increases exponentially above a given threshold, whereas it does not in the lock-free version.
- The difference in the number of restarts is more pronounced under multiprocessing, and appears in association with the increase in the commit time.
- Apart from the differences in commit times, the performance of the two versions is identical.

Summary. The lock-free implementation effectively explores the parallelism, which contributes to the reduction of the average commit times, most notably in scenarios where contention affects the commit lock. The degree to which this improvement is visible in the overall performance of the application depends on the workload. For transactions whose body executes for much longer than the commit, or for highly conflicting workloads (in which

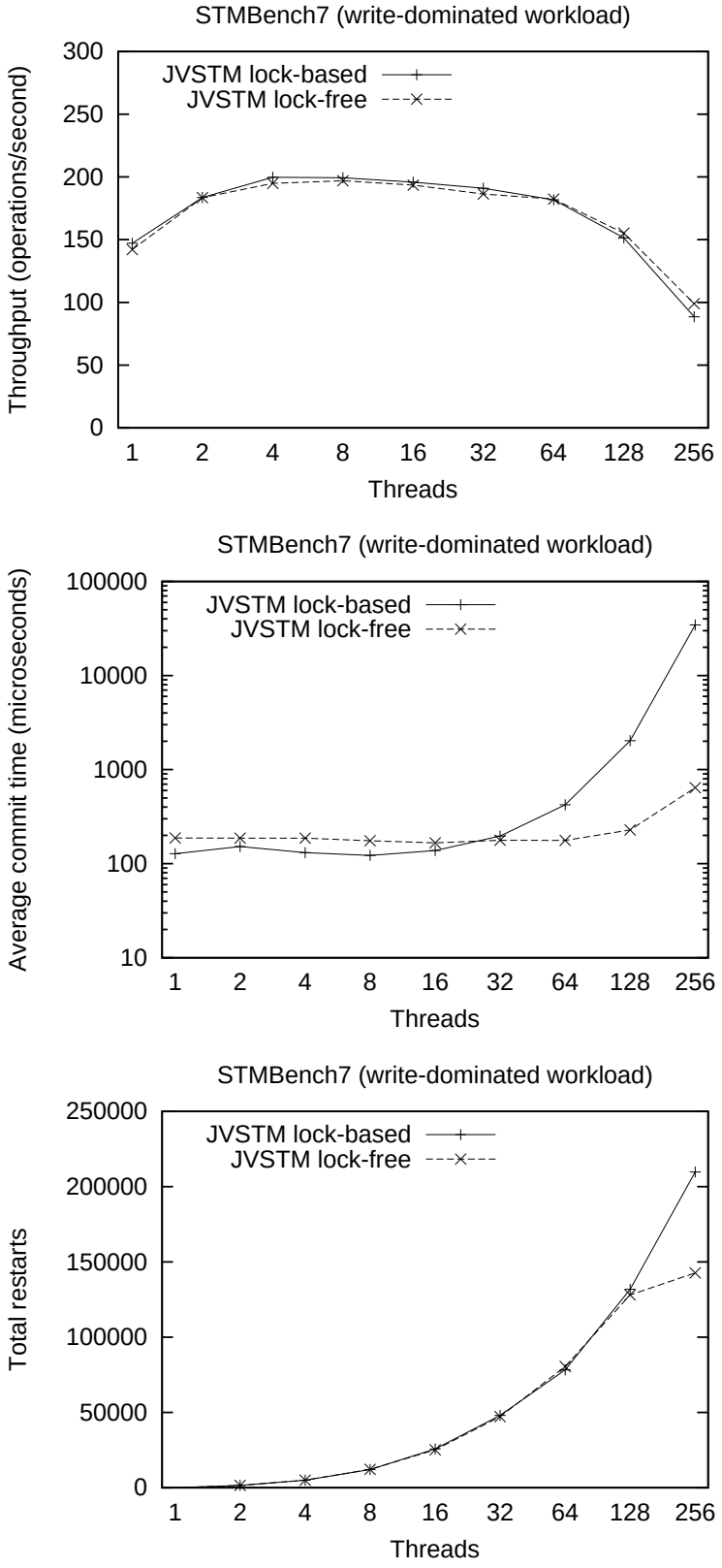

Figure 4.6: The results for the STMBench7 benchmark in the Azul VM.

a close-to-serial execution performs as good as, or better than, parallelizing the execution), the lock-based version is still a valid choice, and it can even exceed the performance of the lock-free version, for a low number of threads. Otherwise, the lock-free version can improve performance and reduce the commit times by better exploring the available parallelism, and, in a multiprocessing scenario, by reducing the amount of transaction restarts that are typically caused by lock convoying. As I have mentioned in Section 4.1.3 on page 98, if the commit phase additionally needs to write to a data repository, there is the potential for greatly increasing the commit duration, so much so that it can become comparable to, or even dominant of, the total duration of the transaction. Under these circumstances, I expect that the improvements brought about by the lock-free version should become even more visible and relevant.

4.4.2 Lock-Free Versus Top-Performing STMs

Most of the modern and top-performing STM implementations use blocking designs, relying on locks to ensure an atomic commit operation. This approach has revealed to be better in practice, in part due to its simplicity, and also because careful design has kept the blocking portion of the algorithms to a minimum. Yet, a blocking approach may present scalability problems as we move into many-core computers.

The direct comparison between different STMs can be a difficult task for several reasons, such as the use of different programming languages, the need for customized execution environments, the lack of common benchmarks, and the existence of non-comparable design decisions. Deuce [Korland et al., 2010] is a Java framework for STMs: Its goal is to provide a common API for adding STM support to an application, with minimal intrusiveness. Deuce requires only that programmers add an `@Atomic` annotation to the methods that should be executed within a transaction. Internally, Deuce provides mechanisms to support plugging STM algorithms, which are invoked when either the application performs any access to a memory location within a method annotated with `@Atomic` or when the application calls or returns from such methods.

Thanks to the Deuce project,[32] there are Java implementations of both the TL2 [Dice et al., 2006] and the LSA [Riegel et al., 2006] algorithms that can be used out-of-the-box. Therefore, in this section, I present a comparison between TL2 and LSA implemented in Deuce version 1.3.0, and the lock-free JVSTM.

[32] `https://sites.google.com/site/deucestm/`

There is a key design difference between Deuce and JVSTM, which makes comparing them difficult. Due to its goal to provide a transparent integration of the STM with the applications, Deuce must instrument every memory access done within an `@Atomic` method, regardless of whether such access is shared or local to the transaction. This can easily lead to over-instrumentation of application code [Dragojevic et al., 2009] leading to performance loss. This is more noticeable in complex applications, whose longer transactions tend to perform more computations using local unshared memory, such as Lee-TM and STMBench7 [Carvalho and Cachopo, 2011]; previous benchmarks have shown performance differences negatively affecting Deuce up to a hundredfold [Fernandes and Cachopo, 2011a].

Much to the contrary, JVSTM's design principle is that the programmer must explicitly identify shared mutable state and make it transactionally safe, by wrapping it with a VBox. Therefore, in JVSTM there is an indirection handler— the VBox—to the actual data. Deuce does not assume a versioned model, and expects each transactional location to contain the data itself: It does not support the indirection required by JVSTM. As such, I chose not to implement JVSTM in Deuce. Although technically feasible, it is not a trivial task, and it would subvert some design choices of either Deuce or JVSTM, which I believe would not contribute to make the comparison fairer.

To keep the comparison fairer, however, I limit my evaluation to benchmarks whose transaction bodies only perform shared accesses. This way the set of memory accesses automatically instrumented by Deuce is the same as that which requires protection through VBoxes. Also, Deuce was configured to use an existing runtime optimization for read-only transactions that allows them to skip read set validation, which is also the standard behavior in JVSTM.

The Deuce project already provides three micro-benchmarks that are often used to evaluate STM implementations. They correspond to three different implementations of integer sets, namely using a linked list, a skip list, and a red-black tree. For these three benchmarks the amount of code executed within each transaction is very small, and the transactions need to manipulate only the set's data structure. This causes the transactional locations managed by either STM to be the same. The common interface `IntSet` supports three operations: Insertion, Deletion and Search.

The performance measurement is the throughput as the number of operations executed per second. They executed in the same hardware configuration as the ones from the previous section and tests run up to 192 concurrent transactions. All tests use a fixed-size set with

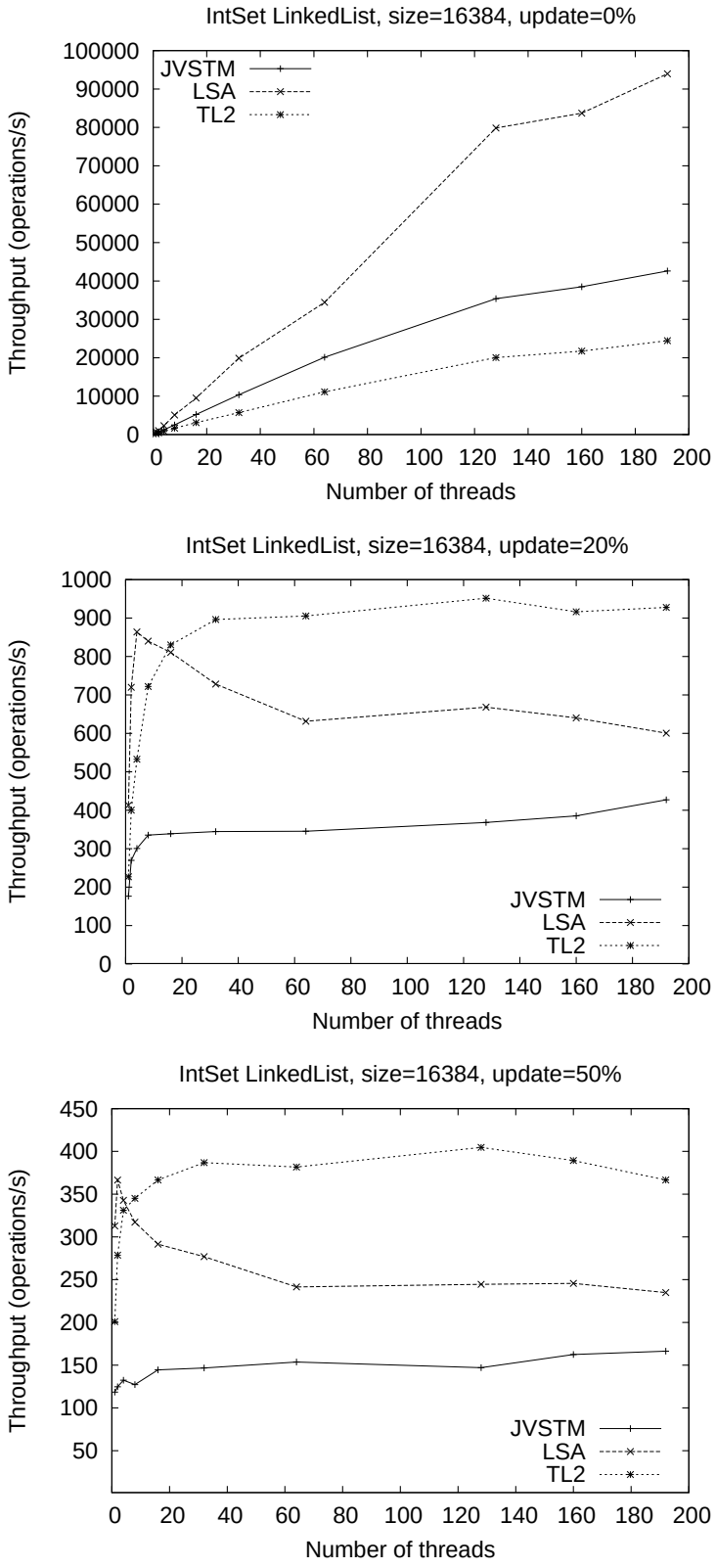

Figure 4.7: The results for the linked list benchmark in the Azul VM.

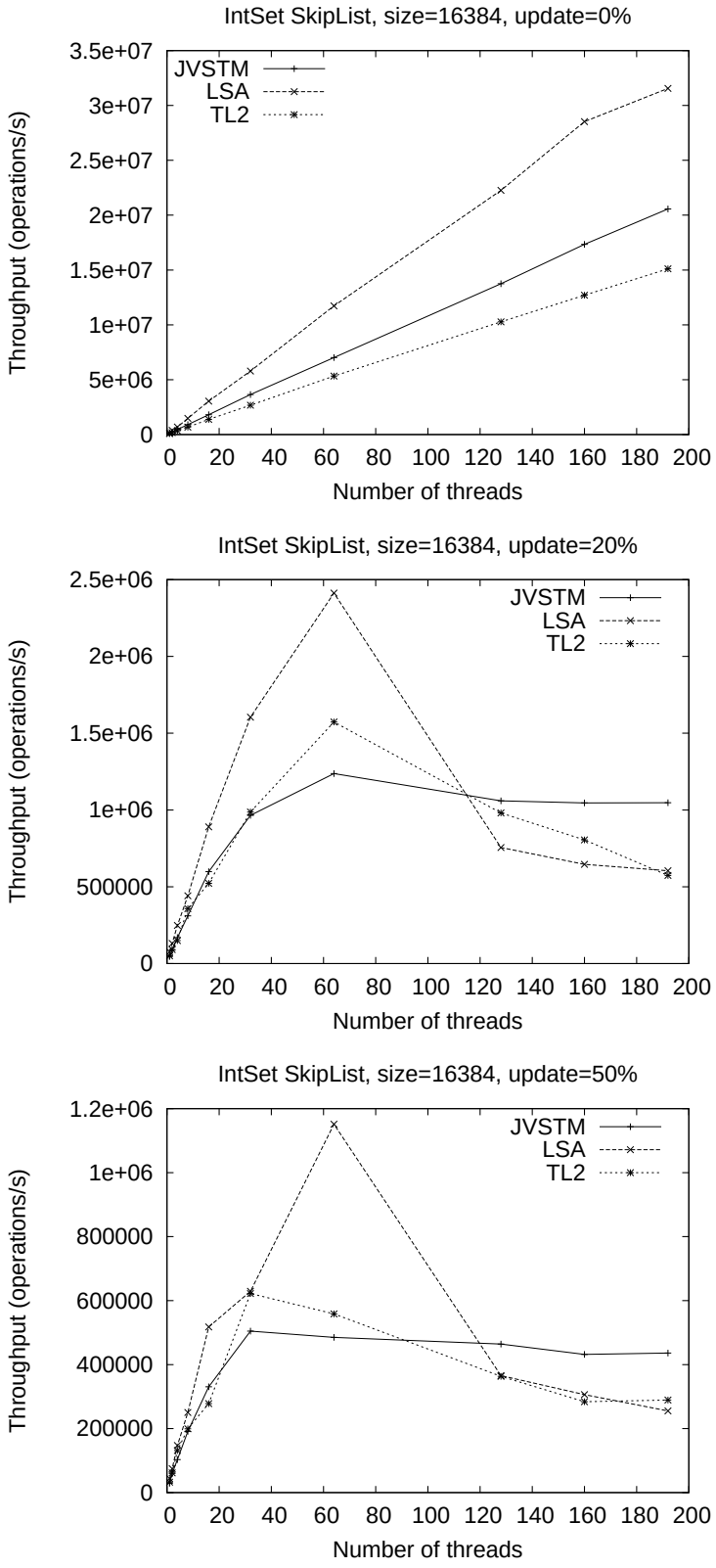

Figure 4.8: The results for the skip list benchmark in the Azul VM.

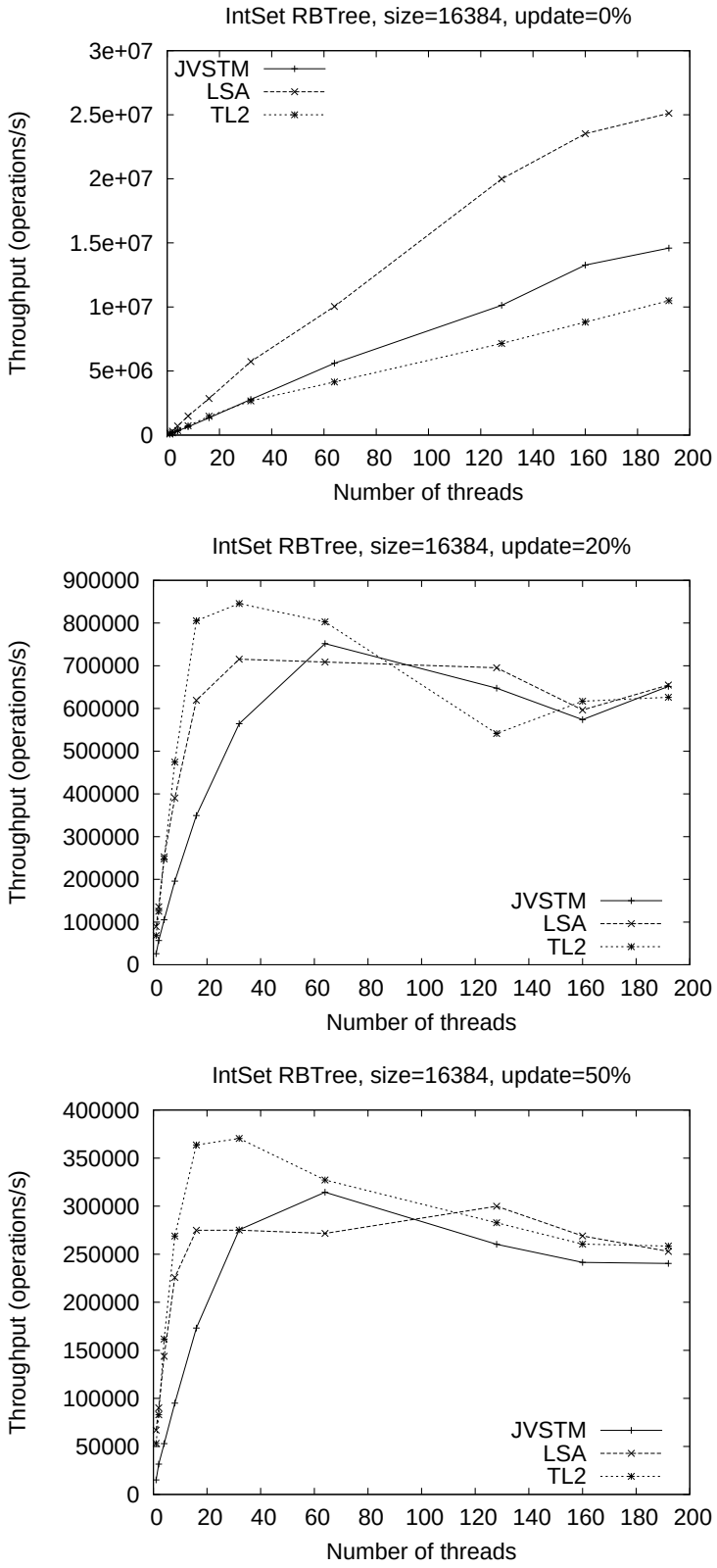

Figure 4.9: The results for the red-black tree benchmark in the Azul VM.

16,384 elements, and vary the ratio of updates. I tested with read-only transactions (only searches, no updates), then with 20 percent updates, and, finally 50 percent updates. The write operations alternated between insertions and deletions, so that the total size of the set was constant over time.

From Figure 4.7 to Figure 4.9 on pages 135–137 I show the results for the three implementations when running from 1 up to 192 parallel transactions. In all read-only tests the JVSTM outperforms TL2, and does worse than LSA. As expected, given the absence of conflicts, all implementations steadily increase the throughput as the number of threads increases, albeit at different rates.

Considering updates, changing the ratio of write-transactions from 20 to 50 percent reduces the overall throughput in all cases, but does not influence the performance of each STM relative to the others. In general, memory accesses within write transactions are more expensive for JVSTM than Deuce, because, whenever reading any location, JVSTM needs to access the transaction's local write set first, to search for a transaction-local write, whereas Deuce just gets the value from memory after cheaply ensuring that it is still valid. To a large extent this accounts for the reason why JVSTM does not achieve a higher maximum throughput in write transactions. The JVSTM performs the worst in the linked list benchmark, showing approximately 30 percent of the throughput of LSA and 50 percent of the throughput of TL2. In the skip list benchmark, JVSTM starts off from behind, but as it keeps a sustained throughput it overtakes the other implementations beyond approximately 120 threads. The throughput of LSA and TL2 spikes at around 60 threads (more noticeable in LSA), but then degrades quickly, most probably due to lock contention. In the red-black tree benchmark, all STMs perform similarly, tending to stabilize approximately on the same throughput from 60 threads onward. Overall, the JVSTM implementation maintains a sustained performance over a growing number of parallel transactions, whereas the lock-based versions have initial higher throughput but then reduce performance for higher thread counts.

4.5 Discussion and Related Work

Even though the initial proposals for STMs used nonblocking designs (from lock-free to obstruction-free), their implementation presented large overheads, making them less performing in practice than simpler blocking designs [Ennals, 2005]. In fact, the original proposal of JVSTM already contemplated a lock-free approach to the commit [Cachopo, 2007]

and hinted at a helping-based solution. However, the preliminary experiments had a limited extent, and showed worse performance than the single-lock approach, for the workloads tested. Most, if not all, of the most recent and current top-performing STMs block some threads to ensure exclusive access to some critical region during certain operations (e.g., TL2 [Dice et al., 2006], RingSTM [Spear et al., 2008], TinySTM [Felber et al., 2008b], NOrec [Dalessandro et al., 2010], and McRT-STM [Saha et al., 2006]), typically the commit operation.

The original JVSTM followed a similar approach, using a single lock to ensure exclusive access to the commit operation of write transactions. Unlike other approaches, however, in the JVSTM read-only transactions do not use any locks, and they are wait-free. Using a single lock has the advantage of being simple to implement and very fast when the lock is uncontended, but has the disadvantage of preventing concurrent commits that could proceed in parallel because the committing transactions accessed unrelated data.

STMs such as TL2 allow such concurrent commits by acquiring locks for all the transactional locations accessed by the transaction (and later releasing them once the commit is concluded), but this approach has overheads that are typically proportional to the size of the read set (R) plus the size of the write set (W). Another problem of these blocking designs is the duration of the critical region, which depends on what needs to be done within that region. The original JVSTM validates the transaction and does the write back of the write set with the lock held, thereby spending time that is proportional to $R + W$ in the critical region. Likewise for TL2 that needs to validate the transaction after acquiring the locks. The LSA algorithm [Riegel et al., 2006] can be applied to either blocking or nonblocking STMs. There is an implementation that is obstruction-free [LSA-STM, 2013], but the one available in Deuce is actually blocking.

The commit algorithm proposed in this chapter allows for concurrent commits without incurring into the overheads of acquiring locks. In this design, validation is lock-free and is performed by each thread individually, racing for a position in the commit queue. The write back phase imposes ordered writes among the committing transactions, with transactions that have a lower version number having to be fully committed first, but a transaction does not block waiting for other transactions to finish. Instead, transactions help to write back other transactions that are ahead of them, thus accelerating the commit of older transactions. The write back phase follows a wait-free algorithm. When multiple concurrent write transactions attempt to commit, there is the possibility that they execute redundant work when they help to write back the same write set. To minimize this effect, the write set of each

transaction is split into buckets, and the helper uses randomization to access the buckets.

NOrec [Dalessandro et al., 2010] uses a lock-free validation mechanism similar to the one I described, but it does not allow any concurrent commits, thus limiting scalability. RingSTM [Spear et al., 2008], on the other hand, supports concurrent commits, but only insofar as the write sets do not overlap. If this occurs, the committing transactions have to write back serially, and there is no helping mechanism installed for this case, meaning that it becomes blocking.

The JVSTM's helping mechanism completely avoids any blocking behaviour, because it maintains the sequential ordering of the write back of all valid transactions. Besides, it maintains intact the properties of the original JVSTM design in what regards read-only transactions, as well as the behavior of reads and writes to transactional locations during the execution of a transaction. There are other proposed STM designs that include helping mechanisms, but they incur performance penalties in the reads: In [Fraser and Harris, 2007] a transaction that reads may need to help other transactions that are writing to that location before it can know the value. In [Marathe and Moir, 2008] the entire read set is validated every time a new record is accessed. To the best of my knowledge, the commit algorithm that I propose is unique in this regard, being the first to use helping to reduce the time of the commit of a transaction, by doing the commit of a single transaction concurrently by as many cores as those available to commit on the system.

The problem of garbage collecting unused versions is seldom addressed in the context of STMs, perhaps because the most common STMs do not support multi-versioning. [Perelman et al., 2010] shows that no STM can be optimal in the number of previous versions kept. In fact, the JVSTM may keep more versions than are actually required. Such can occur, e.g., when a long-running transaction is still running with an old version, and there have already been many updates to the same object. This will cause the JVSTM to keep many versions of the same location, until the old transaction finishes and allows for record cleaning, whereas, in fact, the only versions required could simply be the oldest and the newest versions of the object, because no other transaction was running at the moment. However, and contrary to what is stated by [Perelman et al., 2010], JVSTM is MV-permissive, because it keeps all versions that any running transaction may require. This holds true for both the previous GC algorithm and the new one.

4.6 Summary

In this chapter I present and discuss the modifications required to make JVSTM a completely lock-free STM. The new implementation shows improvements over the lock-based design for large thread counts by eliminating lock contention, and provides results that compete with, and sometimes exceed, some of the top-performing lock-based implementations.

I begin by presenting the motivation for redesigning the commit algorithm, and I describe the circumstances that can affect negatively the performance of the lock-based implementation, thereby arguing in favor of a nonblocking approach.

Next, I present the concrete algorithms: I provide a description of the new commit algorithm and discuss its nonblocking properties. The new lock-free commit algorithm enables write transactions to proceed in parallel, by allowing them to run their validation phase independently of each other, and by resorting to helping from threads that would otherwise be waiting to commit, during the write back phase. I also present a new GC algorithm to dispose of old unused versions, which eliminates the scalability issues identified with the original implementation. The new GC allows for asynchronous identification of the unreachable versions and minimizes its interference with the rest of the transactional system.

To evaluate the lock-free JVSTM, I perform two analysis: One comparing the lock-based with the lock-free implementation; another comparing the lock-free JVSTM with two top-performing STMs. I use benchmarks that are often used to evaluate STM implementations and a custom benchmark to better analyze the features of the lock-free design. The possibility to use a test machine with a high number of real cores enabled me to obtain results that provide high levels of confidence about the real scalability of the algorithms for high thread counts. Regarding the effects of the lock-free commit on the overall performance of applications, there are mixed results, showing that execution patterns and domain-specific logic can have a great influence on the outcome.

I end the chapter, with a discussion of the related work, focusing on various blocking versus nonblocking approaches to STM design.

5 Nonblocking TMM

In this chapter I combine the ideas presented in the two preceding chapters to design *a nonblocking alternative to TMM*: NbTMM. From Chapter 3, I reuse the design of the middleware solution. NbTMM continues to rely on STM to manage transactions, and to use a TDR strictly for storing data externally. The goal is to retain the same qualities of the solution proposed initially, while improving its scalability, especially by eliminating the commit bottleneck. To this effect, I use the lock-free JVSTM implementation, described in Chapter 4, and I apply the same programming techniques, such as thread helping, to redesign TMM, using exclusively nonblocking algorithms. The resulting middleware retains the same transactional consistency guarantees and enables transactions to execute in parallel, even during the commit of write transactions, while ensuring global ordering and without using any locking mechanism that could prevent individual thread progress.

The implementation of NbTMM is nonblocking, insofar as it abstracts from the implementation of its two underlying components—the TDR and the distributed communication infrastructure. To qualify the algorithms in NbTMM as nonblocking, I make the following assumptions regarding the underlying operations that it invokes: (1) every method call eventually returns, and (2) the network communications do not fail. I understand that ensuring these properties may not be trivial, but I do not delve into these aspects in this dissertation. I use the operations of the components underlying NbTMM much in the same way as any programmer uses a *Compare-And-Set* (CAS) operation to achieve synchronization. By this I mean that, even tough these operations may internally use blocking synchronization (like the CAS does), the programmer can use them abstracting from this fact, as long as some assumptions hold.[33] The assumptions that I make are not a strict requirement of the solution: Their purpose is merely to help to reason about the properties of the algorithms in NbTMM, independently of any other concerns related to the properties of the underlying components. To take those properties into account would not preclude the algorithms that I will present; rather it would require them to be extended to account for additional scenarios.

[33] For the CAS, the relevant assumptions are that the operation will complete (either way), and that it appears to execute atomically (the hardware ensures that the CAS is not interrupted halfway through).

The remainder of this chapter presents the design of NbTMM and its experimental evaluation. The presentation sequence of each element follows a structure similar to that of Chapter 3: I start from the integration of the lock-free JVSTM into the enterprise application, and incrementally add the required functionality, which provides support for storing data externally, and to operate in a clustered environment. For the experimental evaluation, I use, once again, the RadarGun benchmarking framework with the same workloads, this time to compare application throughput when using NbTMM versus TMM, and versus the standard implementation approaches that use IMDGs directly.

The fully working implementation of all algorithms described in this chapter, just as in Chapter 3, is materialized within the Fénix Framework, and it is available at `https://github.com/smmf/fenix-framework` from commit with SHA-1 checksum `657ebe3ea2367368442322734ee56c7c046e9222`.

5.1 Overview of the Solution

The design of NbTMM follows the same model that was proposed in Section 3.2 on page 33. In particular, some elements of TMM can be used for NbTMM, exactly as described in Chapter 3. Namely:

- The integration of the lock-free JVSTM in an enterprise application occurs as described in Section 3.4 on page 39, because the new version of the JVSTM retains the same API.
- The caching mechanism described in Section 3.5.3 on page 49 is the same, because it depends solely on the VBox API, which remains unchanged.
- The allocation of new VBoxes described in Section 3.5.4 on page 50 is also the same, because the structure of VBoxes did not change, as well.

In the current section, I present a summary of the most noteworthy aspects of NbTMM's design. These will be detailed in the following sections, when presenting each part of the solution.

Data mappings in external storage are immutable. Data stored in the TDR are never overwritten with different values: Once some value V is written for a given key K, then any other possible writes to K must necessarily be of V. In fact, each version of application data is

written only once; multiple writes for a given key (of the same value) occur only for NbTMM's infrastructural data.

Storage keys are unique. The keys used to store each write set entry are unique, so that transactions may write their write sets concurrently to the TDR with the guarantee that they will never be writing to overlapping keys.

Only the committer persists the write set. The changes produced by a valid write transaction—that is, the data contained in the write set—are written to the TDR only once, and by the thread executing such transaction.

All nodes commit all write transactions, using helping within each node. From a high-level perspective, the commit sequence of each write transaction follows these steps:

1. Perform local validation. If the transaction is invalid, commit fails here.
2. Persist the write set;
3. Broadcast a commit request;
4. Process the commit request's queue in sequence until self commit request has been processed. If it was successfully processed at this point, then end commit; otherwise return to step 1.[34]

Only the committer has access to its read set. Commit requests do not include the read set. This is because sending the entire read set would lead to a very large payload in the commit requests. Instead, validation of the read set is performed only by the originator of the request, before the request is broadcast, and the commit request contains just the version in which the transaction is valid. Consequently, between the validation of request R_a and the actual processing of the commit request by each node, some other request R_b, enqueued meanwhile, may cause the validation status of R_a to become undecidable. As a result, the same commit request may have to be sent more than once by the committing thread. NbTMM includes techniques to minimize the need to resend the same request multiple times. The most relevant of them consists of sending in each commit request the set of identifiers of *benign commit requests*: Benign commit requests are requests that, although not committed at the time the commit request was sent, were seen by the commit request's sender, and are

[34] Step 2 is skipped when repeating the commit sequence, as will be made clear during the detailed presentation of the algorithm.

known not to conflict with its request. Should they meanwhile become committed ahead of this request, they will not prevent it from committing as well. To the best of my knowledge, this technique is unique to NbTMM.

Nodes decide commit status independently. At commit time, each write transaction broadcasts a commit request, such that every node in the cluster can deterministically and independently decide whether the request is valid when it is seen. The write back phase of the commit procedure is also performed independently on each node, using the helping mechanism described in Section 4.2.2 on page 102. In the node from which the associated commit request originated, the new values are written to their corresponding VBoxes. In all other nodes, the value set for each written VBox is the `NOT_LOADED_VALUE`.

5.2 Extension for Persistent Data

Following a structure analogous to that of Section 3.5 on page 41, this section presents the design of the elements in NbTMM that add support for working with data stored externally, enabling NbTMM to be used with a single-node application. I first present, in Section 5.2.1, how data is mapped to the TDR, followed by the algorithms to store and load such data, respectively in Section 5.2.2 on page 147 and in Section 5.2.3 on page 148.

5.2.1 Data Mapping

When a write transaction attempts to commit, one of its earliest steps will be to store the write set in the TDR. This step is performed exclusively by the thread of the committing transaction, and it occurs even before broadcasting a commit request. The goal of this design is to reduce the number of accesses to the TDR. Also, it prevents having to send the full write set as part of the commit request: Only the identifiers of each VBox in the write set are sent (not their values), to allow for each node to perform invalidation of local cache entries, as needed.

When storing the write set, it is not known yet whether the transaction will be globally valid to commit, and consequently the commit version is also not known. Therefore, the data mapping described in Section 3.5.1 on page 43 cannot be used. Instead, a committing transaction generates a commit identifier (*commitId*) from a *Universally Unique Identifier* (UUID). It then uses this identifier to generate unique keys in which to store each entry from the write

set. In NbTMM, each VBoxBody is mapped to the TDR using the function *makeVersionedKey* similar to the one described in Chapter 3, where instead of using a *version* it uses a *commitId*. As described before, this function must return unique values (*versionedKeys*) for each unique pair of arguments. This is trivial to ensure, given that the *commitId* is already a unique value.

Unlike the solution used for TMM, in NbTMM the value stored for each *versionedKey* is simply the actual value written in the transaction for the given VBox. There is no need to store a RepositoryEntry (cf. Listing 3.5 on page 44), because there is yet no version associated.

In a later stage of the commit procedure, which I will describe ahead in Section 5.3.3 on page 161, there will be another key written to the TDR that maps between a transaction version and the corresponding commit identifier. NbTMM keeps in the TDR a mapping between every committed transaction version and its corresponding commit identifier, where the key is the transaction version and the value is the *commitId*. This way, it becomes possible to retrieve the value of a VBox for a given version, by first converting the version into a *commitId*, and then to use the pair (*vboxId*, *commitId*) to retrieve the required value. The mapping between a version and its *commitId* is immutable and, in practice, each NbTMM instance should keep a cache of this information, to avoid requesting it from the TDR.

Notice that *headKeys* have been eliminated in NbTMM.

5.2.2 Persisting Changes

Taking into account that, in the lock-free JVSTM, any thread performing the commit of a write transaction may help to execute the write back of another commit, and that one goal is to minimize the number of calls to the TDR, then it is best to store the write set in the TDR before entering a stage of the commit operation that involves thread helping.

The solution proposed in Section 3.5.2 on page 45, which sends the write set to the repository after validation, can also be applied to the lock-free JVSTM, with minor adaptations. As such, the `persistWriteSet(UUID)` operation, shown in Listing 5.1 on page 149, runs early during the commit—immediately after local validation, and before broadcasting the commit request—and it is executed solely by the thread of the committing transaction, without interfering with any other operation.[35] The implementation of `persistWriteSet(UUID)` is analogous to `commitToRepository(int)` from line 16 of Listing 3.6 on page 46: The

[35]Ideally, concurrent executions of `persistWriteSet(UUID)` will progress without any contention whatsoever, but it will depend on whether the TDR in use supports it.

only differences are that it takes a `commitId` instead of a `txNumber` and it does not invoke `updateTxNumber(Repository, int)`. Now that *headKeys* are not used, the operation equivalent to `putWriteSet(Repository, int)` is much simpler and it appears embedded directly in `persistWriteSet(UUID)`.

NbTMM keeps a mapping between transaction versions and commit identifiers, both in main memory and in the TDR. Each helper thread will set these mappings, when helping any given transaction to commit, unless the mappings are detected already to be set at the time that the helper attempts to do it. This can be seen in `mapTxVersionToCommitId(int, UUID)` in Listing 5.13 on page 169.

5.2.3 Loading the Contents of a VBox

The concerns here are identical to those expressed in Section 3.5.5 on page 53: It is necessary to consider the possibility that a requested value may be missing from main memory, and so to reload it from the TDR. In NbTMM, however, this must be accomplished in a nonblocking fashion. Hence, it is no longer possible to read the value(s), and then, simply use mutual exclusion to update the contents of the VBox. Consequently, the reload algorithm must now take into account that the contents of the VBox may change concurrently, because of either a commit or another reload.

When describing TMM, I introduced the operation `getBoxValue(VBox)` in two steps: First, in Listing 3.11 on page 57, when describing the extension for persistent data and later, in Listing 3.19 on page 72, when describing the extension for clustered environment. Here, I rely on the previously introduced concepts of `NOT_LOADED_VALUE` and `NOT_LOADED_BODY`, and present the complete version of `getBoxValue(VBox)` at once, even tough some of the code is needed only in a clustered configuration.

Simply put, when the reload operation is triggered because a `NOT_LOADED_BODY` is found, it will generate the tail of bodies to be added at the end of the current list of VBoxBodies in memory, just like before. However, when replacing the reference to the `NOT_LOADED_BODY` with a reference to the new tail, the algorithm will use a CAS operation to do so. If the CAS succeeds, the reload is complete; otherwise, the reload knows that some other thread placed some other tail, in which case it must reevaluate the state of the VBox and, if needed, to retry the tail replacement, taking into account the new tail that now exists. When the reload operation is triggered by a `NOT_LOADED_VALUE`, then it is safe to

```
1  class Transaction {
2    ...
3    void persistWriteSet(UUID commitId) {
4      Repository tdr = getRepository().getTDR();
5
6      boolean success = false;
7      try {
8        tdr.beginTransaction();
9
10       // Similar to putWriteSet(Repository, int) from Chapter 3
11       for (WSEntry<VBox, Object> wsEntry : this.writeSet) {
12         Object vboxId = wsEntry.getVBox().getId();
13         Object newValue = wsEntry.getValue();
14
15         tdr.put(makeVersionedKey(vboxId, commitId),
16                 (newValue == null ? NULL_VALUE : newValue));
17       }
18
19       success = true;
20     } catch (Exception e) {
21       throw new CommitException(e);
22     } finally {
23       try {
24         if (success) {
25           tdr.commitTransaction();
26         } else {
27           tdr.rollbackTransaction();
28         }
29       } catch (Exception e) {
30         throw new CommitException(e);
31       }
32     }
33   }
34 }
```

Listing 5.1: The `persistWriteSet(UUID)` operation. It is similar to `commitToRepository(int)` from line 16 of Listing 3.6 on page 46, except that it uses *commitIds* and it writes *versionedKeys* only.

```
1  class Transaction {
2    ...
3    Object getBoxValue(VBox vbox) {
4      ...
5        synchronized (vbox) {
6          vbox.reloadVBox(getReadVersion());
7        }
8      ...
9    }
10
11   static Map<int, UUID> TX_VERSIONS_MAP = ...;
12
13   static UUID getCommitIdForVersion(int version) {
14     return TX_VERSIONS_MAP.get(version);
15   }
16 }
17
18 class VBox {
19   ...
20   void reloadVBox(int requiredVersion) {
21     // same as reloadVBox(int) from Listing 3.11 on page 57
22   }
23
24   void reloadBody(VBoxBody body) {
25     Repository tdr = getRepository().getTDR();
26     UUID commitId = Transaction.getCommitIdForVersion(body.version);
27     body.value = tdr.get(makeVersionedKey(getId(), commitId));
28   }
29 }
```

Listing 5.2: The nonblocking version of `getBoxValue(VBox)`.

just reload and set the missing value on the already existing body. As mentioned in Section 3.6.5 on page 71, this operation is safe because it does not change the list of bodies, and the value of a VBoxBody should be seen as a *value object*.

Listing 5.2 shows the new operation `getBoxValue(VBox)`, which simply eliminates the synchronization monitor that was used to protect the operation `reloadVBox(int)`. The algorithm for `reloadBody(VBoxBody)` is identical to its previous version, except that it uses the `commitId` instead of the `version`. The operation `getCommitIdForVersion(int)` is a trivial lookup on an in-memory map that keeps the association between commit identifiers and their corresponding version. This mapping is updated when committing a write transaction, as will be shown ahead (cf. Listing 5.13 on page 169). The operation `reloadVBox(int)`

remains the same from line 19 of Listing 3.11 on page 57. However, some of its underlying operations are defined differently. These changes are shown in Listing 5.3 on page 152 and Listing 5.4 on page 153:

- `loadVersionsInRange(int)` uses an algorithm similar to the one from line 3 of Listing 3.12 on page 58: It iteratively reads the values for the VBoxBodies with versions in the range [*low*; *high*[, where *low* is the version that is immediately less than or equal to the `requiredVersion`, and *high* is the lowest version already loaded in memory.[36] The main difference is that it uses `commitId`s instead of the `version`s to build the keys to read from the TDR.
- `replaceTailInVBox(VBoxBody)` and `replaceTailInBody(VBoxBody, VBoxBody)` are identical, in that they both attempt to replace, atomically, the existing reference to the `NOT_LOADED_BODY` with the given tail. In case the reference to update changes meanwhile, they both call `updateAndReplaceTailInBody(VBoxBody, VBoxBody)` to find the newest reference to the `NOT_LOADED_BODY`, and then continue reattempting the update via `replaceTailInBody(VBoxBody)`. The only difference between the two replacement operations is that they change references in different object types, respectively a VBox and a VBoxBody.
- `updateAndReplaceTailInBody(VBoxBody, VBoxBody)` obtains the current reference to the `NOT_LOADED_BODY` and decides whether the CAS is still needed: It will be needed if, after eliminating from the tail those VBoxBodies that have meanwhile appeared in memory, the remaining tail still contains any versions to append. The operation `findSubTail(VBoxBody, VBoxBody)` is used to filter the versions that are already present in the VBox's sequence of bodies.

5.3 Extension for Clustered Environment

This section presents the elements that add support for using NbTMM in a clustered environment. I start by introducing NbTMM's distributed architecture and its synchronization mechanisms in Section 5.3.1 on page 154. Next, I describe the commit algorithm in Section 5.3.2 on page 155. The helping operation that is used in each node to commit transactions is addressed separately in Section 5.3.3 on page 161. Finally, in Section 5.3.4 on

[36] Notice that the iteration runs from *high* to *low*, and recall that the expectation is that the required version tends to be at the head of the list of versions.

```
class VBox {
  ...
  VBoxBody loadVersionsInRange(int requiredVersion) {
    Repository tdr = getRepository().getTDR();
    LinkedList<Pair<Object, Integer>> entries = new LinkedList<>();
      Cons.<Pair<Object, Integer>> empty();

    int versionToLoad;
    VBoxBody oldestBody = getOldestBody();
    if (oldestBody == null) {
      versionToLoad = getVersionClock();
    } else {
      versionToLoad = oldestBody.version - 1;
    }

    while (true) {
      UUID commitId = getCommitIdForVersion(versionToLoad);
      Object value = tdr.get(makeVersionedKey(getId(), commitId));

      if (value != null) {
        Pair<Object, Integer> pair =
          new Pair<Object, Integer>(value == NULL_VALUE ? null : value,
                                    versionToLoad);
        entries.addFirst(pair);
        if (versionToLoad <= requiredVersion) {
          break;
        }
      } else if (versionToLoad == 0) {
        // special case: the required version has no value
          entries.addFirst(new Pair<Object,Integer>(null, 0));
        }
      }
      versionToLoad--;
    }

    // build the new tail
    VBoxBody bodies = NOT_LOADED_BODY;
    for (Pair<Object, Integer> pair : entries) {
      bodies = VBox.makeNewBody(pair.first, pair.second, bodies);
    }
    return bodies;
  }
}
```

Listing 5.3: Auxiliary operations for the nonblocking reload of a VBox.

```
1  class VBox {
2    ...
3    void replaceTailInVBox(VBoxBody tail) {
4      VBoxBody current = this.body;
5      if (current != NOT_LOADED_BODY
6          || !compareAndSwapObject(this, FIELD_OFFSET_BODY,
7                                   NOT_LOADED_BODY, tail)) {
8        updateAndReplaceTailInBody(this.body, tail);
9      }
10   }
11
12   void replaceTailInBody(VBoxBody body, VBoxBody tail) {
13     VBoxBody current = body.previous;
14     if (current != NOT_LOADED_BODY
15         || !compareAndSwapObject(body, FIELD_OFFSET_PREVIOUS,
16                                  NOT_LOADED_BODY, tail)) {
17       updateAndReplaceTailInBody(body.previous, tail);
18     }
19   }
20
21   void updateAndReplaceTailInBody(VBoxBody current, VBoxBody tail) {
22     current = getOldestBody(current);
23     tail = findSubTail(current, tail);
24
25     if (tail != NOT_LOADED_BODY) {
26       replaceTailInBody(current, tail);
27     }
28   }
29
30   VBoxBody findSubTail(VBoxBody head, VBoxBody tail) {
31     while (tail != NOT_LOADED_BODY && tail.version >= head.version) {
32       tail = tail.previous;
33     }
34     return tail;
35   }
36 }
```

Listing 5.4: Auxiliary operations for the nonblocking reload of a VBox (cont.).

page 172, I describe how the previous helping is integrated also with the start of every new transaction.

5.3.1 Synchronization Mechanisms

Initially, NbTMM relied also on Hazelcast's distributed topic for publishing commit requests to all members of the cluster. After the first experimental results, however, this design proved to be a scalability bottleneck, due to the amount and payload of commit request messages exchanged using the distributed topic. The NbTMM's ability to process commits faster was being limited indirectly by the speed with which Hazelcast was able to reliably broadcast messages to all members of the cluster. Some experiments ensued and, after considering other messaging infrastructures, such as JGroups,[37] the final choice was to use ZeroMQ.[38] To this choice, contributed not only my own experimental performance results,[39] but also an evaluation by Dworak et al. [2011].

ZeroMQ (ZMQ) is a high-performance asynchronous messaging library designed for large scale distributed applications. ZMQ bases its API on a generalization of the concept of sockets, which can represent a many-to-many connection between endpoints. On top of the basic API, ZMQ also provides several message-based communication patterns, for the most common usages, such as publish-subscribe, request-reply, and push-pull.

Figure 5.1 on page 155 shows the distributed architecture of NbTMM. Each Node ($Node_1..Node_n$) represents a clustered NbTMM instance running in an application server. To broadcast a commit request to every other node, each node only needs to send its request to the Sequencer, using a Push-Pull messaging pattern. In turn, the Sequencer is waiting until some node pushes a commit request. Then, internally, the Sequencer orders every request that it receives, in any given order, and then broadcasts the requests to every node, using a Publish-Subscribe messaging pattern. This design guarantees that all nodes will receive all commit requests in the same order, determined by the Sequencer. Each node subscribes to the commit requests published by the Sequencer, and upon receiving a commit request, the node just adds it to a queue of requests to process.

Next, I will detail the algorithm followed by each node when committing a transaction. Similarly to the design of TMM, in NbTMM I assume that after any data is written to the

[37] http://www.jgroups.org/
[38] http://zeromq.org/
[39] https://github.com/smmf/pubsub-bench

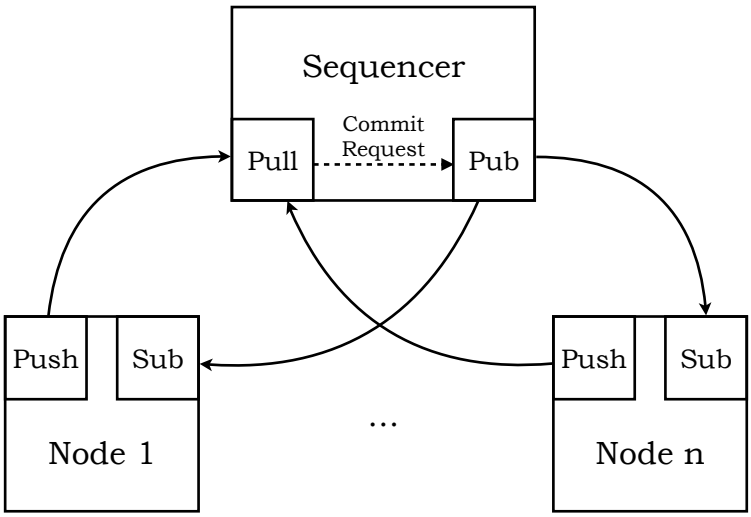

Figure 5.1: Distributed architecture of NbTMM. It uses a dedicated component to receive all commit requests, which are then published in the same order to all nodes.

TDR, it will be made available to other nodes as requested. This allows me to assume that, when a commit request is broadcast to all nodes, data written to the TDR will already be accessible for the other nodes to read, if needed. In practice, this is necessary so that when a commit request is processed in a node, thereby creating a new version, and, afterwards, some transaction begins in this new version, any data that it may need to reload from the TDR will be available.

5.3.2 Committing a Transaction

Listing 5.5 on page 156 shows the algorithm of the cluster-wide commit operation. It performs the following steps in a loop until the transaction is either successfully committed or found invalid.

1. Perform read set validation, after processing pending commit requests, if any (line 6).
2. If this is the first attempt:
   (a) Create a new commit request with a unique identifier (line 8).
   (b) Persist the write set to the TDR (line 9).
3. For subsequent commit requests, update the original request (line 11).
4. Broadcast the commit request (line 13).

```
1  class Transaction {
2    ...
3    void commit() {
4      CommitRequest myRequest = null;
5      do {
6        validate();
7        if (myRequest == null) {
8          myRequest = makeCommitRequest();
9          persistWriteSet(myRequest.getId());
10       } else {
11         myRequest = updateCommitRequest(myRequest);
12       }
13       broadcast(myRequest);
14       helpToCommit(myRequest.getId());
15
16       // At this point this.lastProcessedRequest refers to this
17       // transaction's request. If it was decided as VALID, then
18       // the transaction must be fully committed.
19     } while (this.lastProcessedRequest.getValidationStatus()
20              != ValidationStatus.VALID);
21   }
22 }
```

Listing 5.5: The main algorithm of the nonblocking commit. It keeps attempting to commit until it either succeeds or the transaction becomes invalid. The validation is performed locally, exclusively.

5. Process all pending commit requests until own request is processed (line 14).

When committing a write transaction, the first operation to perform is to check the transaction's validity against the most recent state known by the node. To do so, the operation `validate()`, detailed in Listing 5.6 on page 158, first processes any pending commit requests (line 4), to update the local state of the node. The operation `helpCommitAll(CommitRequest)` will iterate all existing requests, starting from the one last processed by this transaction.[40] This corresponds to the first helping stage, where a transaction may help others to commit. The bulk of the work is done by the operation `handle(Transaction)` in line 17. This operation is defined in the `CommitRequest`, and it will be described in detail in the following section. When `helpCommitAll(CommitRequest)` completes, it returns the last request

[40]The value of `this.lastProcessedRequest` is initialized during the start of a new transaction, as described in Section 5.3.4 on page 172.

that it helped to commit, which is used to update this transaction's reference to its `lastProcessedRequest`. After having processed as many commit requests as possible, `validate()` performs a snapshot validation as described in line 21 of Listing 4.6 on page 110. If validation fails, `snapshotValidation(int)` will throw a `CommitException`, which in turn, terminates the commit entirely. If the validation succeeds, the transaction is upgraded, by updating its read version to the version of `lastSeenCommitted` record.

Assuming that this is the first commit attempt, a new commit request will be created. The detailed contents of a commit request will be presented ahead. For now, it suffices to mention that a commit request contains the necessary information to allow any node to independently and deterministically decide whether the commit is valid. In case this is not the first attempt to commit this transaction, then all that is necessary is to update the contents of the commit request. Notice that the commit identifier of a transaction does not change when reattempting to commit. A commit request holds a unique commit identifier, which is used to store this transaction's write set persistently in the TDR, during the first commit attempt. The operation `persistWriteSet(UUID)` has been presented already in Listing 5.1 on page 149.

Next, the commit request is broadcast to all nodes (cf. `broadcast(CommitRequest)` in Listing 5.7 on page 158). This operation uses the underlying communication infrastructure to send the commit request to all nodes. The most important guarantee provided by the operation `ClusterUtils.sendCommitRequest(CommitRequest)` (line 5) is that all nodes will always receive in the same order all commit requests sent. Before sending the commit request, however, a local map of commit requests (`COMMITS_MAP`) is updated with the current request and the associated owning transaction. Later, when handling incoming commit requests, this will allow for the receiver to recover the transactional context for the requests of transactions from the local node. In practice, this enables the helper transactions to access the helped transaction's write set, allowing for the local node's cache to be immediately updated with the write set values, whereas caches in the remote nodes will not have access to the full write set and, thus, will only be able to invalidate their corresponding cache entries, by committing a `NOT_LOADED_VALUE` to the VBoxes.

After the commit request is broadcast, the commit phase enters the second helping stage: The sender thread will now help to process all commit requests in order, until it completes handling its own commit request, which, assuming that the underlying communications do not fail, will eventually appear in the list of commit requests to process. The operation

```
class Transaction {
  ...
  CommitRequest validate() {
    this.lastProcessedRequest =
      helpCommitAll(this.lastProcessedRequest);

    ActiveTxRecord lastSeenCommitted = LAST_COMMITTED_RECORD;
    snapshotValidation(lastSeenCommitted.txNumber);
    upgradeTx(lastSeenCommitted);
  }

  CommitRequest helpCommitAll(CommitRequest current) {
    CommitRequest previous = null;

    do {
      previous = current;
      current.handle(this);
      current = getNextRequestMaybeNull(current);
    } while (current != null) {
    return previous;
  }

  void upgradeTx(ActiveTxRecord newRecord) {
    setReadVersion(newRecord.txNumber);
  }
}
```

Listing 5.6: Validation before broadcasting commit request. Validation is a local-only operation, performed before broadcasting the commit request. Other pending commit requests are processed first, to ensure that validation occurs against the most recent state.

```
class Transaction {
  ...
  void broadcast(CommitRequest request) {
    COMMITS_MAP.put(request.getId(), this);
    ClusterUtils.sendCommitRequest(request);
  }
}
```

Listing 5.7: The `broadcast(CommitRequest)` operation. Each node keeps a map of its own commit requests and associated transactions. Later, this will allow a helper transaction to recover the context of the helped transaction, when handling a commit request from the local node.

```
1  class Transaction {
2    ...
3    void helpToCommit(UUID myRequestId) {
4      CommitRequest current = this.lastProcessedRequest;
5      do {
6        current = getNextRequest(current);
7
8        current.handle(this);
9        this.lastProcessedRequest = current;
10
11       if (current.getId().equals(myRequestId)) {
12         return current;
13       }
14     } while (true);
15   }
16 }
```

Listing 5.8: The `helpToCommit(UUID)` operation. It processes all commit requests, in order, until it completes the processing of its own commit request, identified by `myRequestId`.

`helpToCommit(UUID)`, shown in Listing 5.8, invokes the helping of each request. It starts from the last commit request that this helper transaction has already helped, and it continues forward until it eventually processes its own request (detected in line 11). Again, `handle(Transaction)` is responsible for the actual helping of a single commit request.

The main difference between `helpCommitAll(CommitRequest)` and `helpToCommit(UUID)`, apart from the loop termination condition, is that the former obtains the next request to handle using `getNextRequestMaybeNull(CommitRequest)`, whereas the latter uses `getNextRequest(CommitRequest)`. In the first case, when there are no more requests in the queue, the method simply returns `null`, whereas in the second case, at least one more request must arrive (this transaction's own request that was just broadcast), so this operation will never return `null`. Any delay here corresponds, at most, to the amount of time that it takes for the request to travel to the Sequencer and back.

After returning from `helpToCommit(UUID)`, the `lastProcessedRequest` of the transaction is guaranteed to reference the transaction's own request. The main `commit()` operation can now read this request's validation status to determine whether it was set as valid (line 19 of Listing 5.5 on page 156). If so, then this commit is fully processed and the `commit()` operation returns; otherwise it repeats the commit procedure described in this

section.

While executing the commit steps described above, whenever the transaction helps to process a commit request, it is notified of the request's validation status—the `handle(Transaction)` operation receives a reference to the transaction, so that it can call back to notify the transaction of the request's validation outcome—either `VALID` or `UNDECIDED`. For a request that **is not committed**, that is its status is `UNDECIDED`, this transaction will check whether it would have made this transaction invalid if such request had committed. If committing the request would not invalidate this transaction, then the request is considered benign to this transaction, and it is added to a set of benign commit requests. When sending its own commit request, the transaction includes the set of benign commit requests.

Other transactions are themselves concurrently attempting to commit and, once `UNDECIDED`, they will resend their requests, which may end up ahead of this transaction's request, in which case, if successfully committed, they would make this transaction's request `UNDECIDED`. By broadcasting to all nodes that some other requests are benign, this transaction is improving the shared knowledge and the likelihood of its commit request being considered as `VALID` still.

Consider the following illustrative scenario: At a given moment there are n concurrent write transactions attempting to commit ($T_1 ... T_n$). Let us assume that these transaction are all locally valid in the most recent version of the system, V_1. Now, they all attempt to commit by broadcasting their own commit request, which mentions that each transaction is valid in V_1. Eventually, these n requests end up queued in every node, in some given order.

The (lucky) first request enqueued will be trivially `VALID` and commit successfully. However, all other $n - 1$ commit requests will be `UNDECIDED`: This is because, when they are analyzed in each node, the most recent version will be V_2 already (generated by the commit of the first request), and they are only valid up to V_1. If each failed transaction was simply to repeat the commit loop, then it would locally revalidate against V_2 and, either become invalid or upgrade its read version to V_2, in which case it would eventually resend the request.

The problem here is that, at most, only one of all the $n - 1$ requests will be able to commit,[41] whereas the others will be `UNDECIDED`, once again. The key insight is that the transactions owning each of the $n - 1$ requests can take the opportunity, when processing

[41] They may all fail to commit, in the eventuality that another commit request from the thread that was able to commit the first transaction, enters again ahead of all others.

the queue of commit requests, and validate themselves against the write set of the requests that they see: Any of these requests, although `UNDECIDED` at the present moment, may appear committed next time. In the event they appear committed, this transaction's commit request will provide the information on whether they are compatible, i.e. benign, such that their commit does not invalidate this request's own commit.

In Listing 5.9 on page 162, I show how to extend the commit algorithm to support this behavior. The transaction's property `undecidedRequests` stores the `UNDECIDED` commit requests seen by this transaction, and the property `checkedRequestIds` stores the commit identifiers of the requests already checked, to avoid checking them multiple times. The transaction instance is notified of each processed request's outcome through a call to either `notifyValid(CommitRequest)` or `notifyUndecided(CommitRequest)`. Whenever a `VALID` request is seen, it can be removed from the requests to check (`undecidedRequests`), because it will necessarily already be taken into account during the validation procedure, as it corresponds to a committed version. The requests that are `UNDECIDED`, are added to the `undecidedRequests`, unless they have already been checked before. Then, the `commit()` operation, before creating or updating the commit request to broadcast, computes the set of benign request identifiers. The operation `selectBenignRequestsToSend()` iterates the undecided requests, yet unchecked, and checks whether their write set intersects this transaction's read set (line 20): Those requests, whose write set does not intersect with this transaction's read set are returned and added to the commit request. Immediately after, the `undecidedRequests` maps is cleared. This is because the algorithm avoids sending the same information multiple times: Each node keeps a map of benign requests for each request, and each request sent will contain only the newly found benign requests.

5.3.3 Helping to Commit a Transaction

The commit request is the core element of the commit operation. Its structure is shown in Listing 5.10 on page 163. During the creation of a commit request (line 31 of Listing 5.9 on page 162) a unique identifier is automatically generated and assigned to the `id` field. When updating the commit request (line 34), the identifier remains the same. Additionally, the commit request holds the following attributes:

- `validTxVersion`: Represents the transaction's current read version (after a successful `validation()`). This request will be immediately valid to commit, if when it is

```
class Transaction {
  ...
  Map<UUID, CommitRequest> undecidedRequests = new HashMap<>();
  Set<UUID> checkedRequestIds = new HashSet<>();

  public void notifyValid(CommitRequest request) {
    this.undecidedRequests.remove(request.getId());
  }

  public void notifyUndecided(CommitRequest request) {
    if (!this.checkedRequestIds.contains(request.getId())) {
      this.undecidedRequests.put(request.getId(), request);
    }
  }

  Set<UUID> selectBenignRequestsToSend() {
    Set<UUID> benignIds = new HashSet<>();
    for (CommitRequest request : this.undecidedRequests.values()) {
      this.checkedRequestIds.add(request.getId());
      if (validAgainstWriteSet(request.getWriteSet())) {
        benignIds.add(request.getId());
      }
    }
    return benignIds;
  }

  void commit() {
      ...
      Set<UUID> benignRequestIds = selectBenignRequestsToSend();
      if (myRequest == null) {
        myRequest = makeCommitRequest(benignRequestIds);
        persistWriteSet(myRequest.getId());
      } else {
        myRequest = updateCommitRequest(myRequest, benignRequestIds);
      }
      this.undecidedRequests.clear();
      ...
  }
}
```

Listing 5.9: Computing benign commit requests. These are sent along with the commit request to reduce the number of retries required to commit a transaction.

```
1  class CommitRequest {
2    enum ValidationStatus {
3      UNSET, VALID, UNDECIDED;
4    }
5
6    // Attributes set by the sender.
7    UUID id;
8    int validTxVersion;
9    Set<Object> writeSet;
10   boolean isWriteOnly;
11   Set<UUID> benignCommits;
12
13   // Attributes set in each destination node.
14   CommitOnlyTransaction transaction;
15   final AtomicReference<ValidationStatus> validationStatus =
16     new AtomicReference<>(ValidationStatus.UNSET);
17   final AtomicReference<CommitRequest> next = new AtomicReference<>();
18 }
```

Listing 5.10: Structure of a `CommitRequest`.

analyzed, the most recent committed version is still `validTxVersion`.

- `writeSet`: This holds the set of VBox identifiers for the VBoxes written during the transaction.
- `isWriteOnly`: Indicates whether the read set of the transaction is empty. Write-only transactions will always be valid to commit at any point in time.
- `benignCommits`: Set of identifiers of commit requests that, even if committed ahead of this request, will not affect this request's validity.

The commit request attributes just described contain all the information that is sent when broadcasting a commit request.[42] Additionally, in each node, when a request is received, the communications thread responsible for receiving commit requests sets the reference to a `transaction`, by calling `assignTransaction()`. This operation, shown in Listing 5.11 on page 164, consults the `COMMITS_MAP` to determine whether the request corresponds to a transaction from the current node. This will decide the type

[42] When resending a commit request, the attributes `writeSet` and `isWriteOnly` are not included. Each node keeps this information for requests that are considered `UNDECIDED`, just like it does with the set of benign requests per request.

```
class CommitRequest {
  ...
  void assignTransaction() {
    Transaction tx = COMMITS_MAP.remove(this.commitid);
    if (tx != null) {
      this.transaction = new LocalCommitOnlyTransaction(this, tx);
    } else {
      this.transaction = new RemoteCommitOnlyTransaction(this);
    }
  }

  void handle(Transaction helper) {
    try {
      this.tx.commit();
    } catch (CommitException e) {
      // discard exceptions from invalid commits
    } finally {
      if (getValidationStatus() == ValidationStatus.UNDECIDED) {
        helper.notifyUndecided(this);
      } else if (getValidationStatus() == ValidationStatus.VALID) {
        helper.notifyValid(this);
      }
      ClusterUtils.tryToRemoveCommitRequestFromHead(this);
    }
  }
}
```

Listing 5.11: The main operations in a `CommitRequest`.

of `CommitOnlyTransaction` (`COT`, for short) that will be set in the `transaction` attribute. A `COT` is a special-purpose transaction that re-enacts the commit section of a JVSTM transaction. Its commit is different from the regular commit described in the previous section. The commit of a `COT` does not involve any distributed communication, and it will perform only the node-local commit of the given transaction, which may include helping other transactions ahead of it to complete, if necessary. The difference between `LocalCommitOnlyTransaction`s and `RemoteCommitOnlyTransaction`s is that the former have access to the instance of the originating transaction (`tx`), and so they will be able to use the full write set to apply changes in memory when committing this transaction, whereas the latter will just set `NOT_LOADED_VALUE`s in each VBox identified in the commit request's `writeSet`. In the following, I describe the commit algorithm of a `COT`, noting where due, any differences between its two subtypes.

There are two other attributes in the `CommitRequest`, whose value is only meaningful on the nodes receiving the request: The `validationStatus` is always initialized to `UNSET` and, during the handling of the request, it will transition to either `VALID` or `UNDECIDED`. The `next` is a reference to the following `CommitRequest` to process. In each node, the queue of commit requests is maintained using a strategy similar to the one used for the JVSTM's ATR list: The structure is a singly-linked list of requests, where each request holds a reference to the next request. The communications infrastructure keeps a reference to the first and last requests received. When a new request arrives it is added to the tail. When a request is handled, the system tries to atomically advance the reference at the head (using a CAS), if and only if, there is already another request and the request just handled is still at the head.

As noted earlier, the commit request's `handle(Transaction)` operation is the entry point for executing a transaction's request to commit. This method, shown in Listing 5.11 on page 164, invokes the `commit()` of the `COT` assigned to the request. The `COT`, which is a subtype of `JVSTM`'s `Transaction` type, will transparently activate the commit machinery of the lock-free JVSTM. The `handle(Transaction)` operation, silently discards any `CommitException` thrown due to a failed validation. Instead, after executing the `commit()`, it notifies the helper transaction, of the outcome, by invoking either `notifyValid(CommitRequest)` or `notifyUndecided(CommitRequest)`. Lastly, it tries to remove the current commit request from the head of the list of commit requests to process. Notice that any transactions that may still refer an older commit request will always be able to access all commit requests by iterating through their `next` reference. The commit request currently at the head is relevant only for new transactions, as will be explained in Section 5.3.4 on page 172.

As described in Section 4.2 on page 99, the lock-free commit of a transaction starts with the validation phase. If it succeeds, then the transaction's `commitTxRecord` is enqueued and then written back, possibly with the help of other committing transactions.

The commit of a `COT` follows a similar sequence. The main difference is that, whereas in the single-node JVSTM, each transaction would race with the others to validate itself and enqueue its `commitTxRecord` in the ATR list, now the validation of the same `COT` can be performed by any transaction that is helping to commit. This means that the `COT`'s validation, and the creation and enqueueing of its `commitTxRecord` must take helping into account.

All COTs are processed in the same order by all helping transactions to determine their validation status—the order used is the order of enqueueing of their commit requests. So, when validating a COT, all COTs ahead have already been either decided VALID and enqueued for write back, or discarded as UNDECIDED. This means that if a COT is deemed VALID to commit, its commit version can be easily determined by looking at the version of the last record enqueued for write back. Naturally, the algorithm must account for the case where another helper concurrently enqueues a given COT's commitTxRecord for write back, so as to not mistakenly enqueue the same record multiple times.

The validation algorithm relies on the following two invariants:

I1 The commitTxRecord of a COT is always set before setting the COT's validation status as VALID.

I2 A validation status VALID is always set before attempting to enqueue the corresponding commitTxRecord for write back.

The validation of a COT considers two cases: A write-only COT is immediately valid to commit; for a read-write COT it is necessary to check whether its read version is still up to date or, if not, to check whether each of the commitTxRecords enqueued after the COT's read version correspond to commits of transactions that are benign. The validation procedure of a COT ends when either (1) the status is determined to be UNDECIDED and a CommitException is thrown, or (2) the status is determined to be VALID and the commitTxRecord has been successfully enqueued for write back. The write back phase is the same as described in Section 4.2.2 on page 102.

First, I will describe, in Section 5.3.3.1, the handling of the write-only case, which is common to both cases. Then, in Section 5.3.3.2 on page 170, I describe the additional work performed in the read-write case.

5.3.3.1 The Write-Only Case

Listing 5.12 on page 167 shows the main validation algorithm for a COT, which simply detects the type of request and calls the appropriate action. Given that write-only transactions are always valid to commit, the operation validateWriteOnly() needs only to assign the commitTxRecord (line 14), to set the validation status (line 16), and to enqueue the record for write back (line 17). To decide the version of its commitTxRecord, the validation gets

```
1  class CommitOnlyTransaction extends Transaction {
2    ...
3    void validate() {
4      if (this.commitRequest.getIsWriteOnly()) {
5        validateWriteOnly();
6      } else {
7        validateReadWrite();
8      }
9    }
10
11   void validateWriteOnly() {
12     ActiveTxRecord existingRec = getLastEnqueuedRecord();
13     if (this.commitTxRecord == null) {
14       assignCommitRecord(existingRec.txNumber + 1, getWriteSet());
15     }
16     this.commitRequest.setValid();
17     enqueueForWriteBack(existingRec, this.commitTxRecord.getWriteSet());
18   }
19
20   ActiveTxRecord getLastEnqueuedRecord() {
21     ActiveTxRecord existingRec = LAST_COMMITTED_RECORD;
22     ActiveTxRecord nextRec = existingRec.getNext();
23
24     while (nextRec != null) {
25       existingRec = nextRec;
26       nextRec = nextRec.getNext();
27     }
28     return existingRec;
29   }
30
31   void assignCommitRecord(int txNumber, WriteSet writeSet) {
32     ActiveTxRecord record = new ActiveTxRecord(txNumber, writeSet);
33     compareAndSwapObject(this, FIELD_OFFSET_COMMIT_TX_RECORD,
34                          null, record);
35     % this.commitRequest.setValid();
36   }
37
38   abstract WriteSet getWriteSet();
39 }
```

Listing 5.12: Validation of a write-only `CommitOnlyTransaction`.

the most recently enqueued record (line 12) and uses the next version number to create the `commitTxRecord`: Regardless of whether `existingRec` is already written back, this is the record after which to enqueue the next record.

It may be the case that when executing `assignCommitRecord(int, WriteSet)` the CAS operation fails (line 33). This means that another helper has concurrently performed the enqueueing. Notice that the test in line 13 prevents two `commitTxRecord`s for the same `COT` from being enqueued: By transitivity, the two invariants mentioned above ensure that assigning a `COT`'s `commitTxRecord` must be done before attempting to enqueue it. Also, the value of `this.commitTxRecord` is read (line 13) only after reading the most recently enqueued record (line 12). So, when the `commitTxRecord` is seen as `null`, the `existingRec` obtained cannot contain already this `COT`'s `commitTxRecord`. When calling `assignCommitRecord(int, WriteSet)`, the write set is obtained using the `COT`'s abstract operation `getWriteSet()`. A `LocalCommitOnlyTransaction` will return the full write set, whereas a `RemoteCommitOnlyTransaction` will return the write set whose values are all the `NOT_LOADED_VALUE`. During the write back phase, this will affect the values stored in the VBoxBodies committed to each VBox identified in the write set. The write back algorithm, however, will be unaware of this difference.

The operation `enqueueForWriteBack(ActiveTxRecord, WriteSet)`, shown in Listing 5.13 on page 169, performs two main tasks: First it ensures the required mappings for the transaction's version, and then it attempts the actual enqueueing. Executing `mapTxVersionToCommitId(int, UUID)` is necessary to enable the reload of VBoxes (cf. Section 5.2.3 on page 148). Notice that this last operation is responsible for maintaing the mapping both in memory and in the TDR, and that it minimizes the accesses to the TDR, by first checking whether the mapping is already present in memory. If so, then it means that it must also be in the TDR already. When multiple transactions attempt to help to enqueue the same record, this helps to prevent having many transactions store the same mapping to the TDR. The operation `mapTxVersionToCommitRecord(int, ActiveTxRecord)` is a simple in-memory mapping that will allow, later, for the operation that validates read-write transactions to immediately fetch the `ActiveTxRecord` for a given version (cf. line 4 of Listing 5.14 on page 171). The operation `trySetNext()` was defined in Listing 4.1 on page 101: It is attempted only if the `commitTxRecord` to enqueue is the logical immediate successor of the given record.[43]

[43]Otherwise, the required enqueueing was already performed. For example, the given record may not be the immediate predecessor, when the operation `validateWriteOnly()` initially obtains the `existingRec`, already after some other helper has enqueued the current `commitTxRecord`.

```
1  class CommitOnlyTransaction extends Transaction {
2    ...
3    void enqueueForWriteBack(ActiveTxRecord rec, WriteSet writeSet) {
4      int version = this.commitTxRecord.txNumber;
5      mapTxVersionToCommitId(version, this.commitRequest.getId());
6      mapTxVersionToCommitRecord(version, this.commitTxRecord);
7      if (rec.txNumber + 1 == this.commitTxRecord.txNumber) {
8        rec.trySetNext(this.commitTxRecord);
9      }
10   }
11 }
12
13 class Transaction {
14   ...
15   void mapTxVersionToCommitId(int version, UUID commitId) {
16     if (getCommitIdForVersion(version) == null) {
17       persistTxVersionToCommitId(version, commitId);
18       TX_VERSIONS_MAP.putIfAbsent(version, commitId);
19     }
20   }
21 }
```

Listing 5.13: Enqueuing of a valid `CommitOnlyTransaction` for write back.

The details of the operation `persistTxVersionToCommitId(int, UUID)`, used by `mapTxVersionToCommitId(int, UUID)`, are not shown. This operation is similar in structure to `persistWriteSet(UUID)` from Listing 5.1 on page 149, except that it performs a single `tdr.put(version, commitId)` to store the mapping between the `version` and the `commitId`.

5.3.3.2 The Read-Write Case

The algorithm for validation of a read-write `COT` is shown in Listing 5.14 on page 171. In comparison to the validation of a write-only `COT`, the additional aspects to take into consideration have to do with determining the validation status of the `COT`. If the transaction's validation status is seen already set (line 8 and line 11), then the operation just acts accordingly, either ensuring it gets enqueued for write back or throwing a `CommitException`, respectively for the `VALID` and the `UNDECIDED` status. If the validation status is yet `UNSET` (line 13), then the helper must first determine the validation status of the request. To do so, this helper iterates (line 15) all commit records following the record for which the transaction is known to be valid (`existingRec`, first obtained in line 4), and checking whether all more recent commit records correspond to commits of requests that are listed as benign in this transaction's commit request. If so, then this `COT` is set `VALID`; otherwise, it is set `UNDECIDED` and the `CommitException` is thrown. This verification loop must also account for the possibility that, meanwhile, another helper concurrently enqueues a record for this request, which means that it is necessarily valid and no more checking is needed (line 17).

Notice that, unless the operation `validateReadWrite()` throws an exception, it will always attempt to execute `enqueueForWriteBack(ActiveTxRecord, WriteSet)` (line 31), even when the validation status is already seen as having been set by another helper. The general idea when designing a helping algorithm is that each participant must do its own part in ensuring that every sequence of an algorithm gets accomplished. It would be wrong to assume, just because some other transaction already set the validation status of this request, that it has already enqueued the `commitTxRecord`, even though, probably it already has. But, if so, then this transaction will detect it when attempting itself the enqueueing of the record. The same reasoning applies to any part of the helping algorithm, whenever a direct dependency between two steps cannot be established.

```
1  class CommitOnlyTransaction extends Transaction {
2    ...
3    void validateReadWrite() {
4      ActiveTxRecord existingRec =
5        getCommitRecordForVersion(this.commitRequest.getValidTxVersion());
6
7      ValidationStatus status = this.commitRequest.getValidationStatus();
8      if (status == ValidationStatus.VALID) {
9        existingRec = getForwardRecord(existingRec,
10                                      this.CommitTxRecord.txNumber - 1);
11     } else if (status == ValidationStatus.UNDECIDED) {
12       throw new CommitException();
13     } else {          // ValidationStatus.UNSET
14       ActiveTxRecord nextRec = existingRec.getNext();
15       while (nextRec != null) {
16         UUID commitId = getCommitIdForVersion(nextRec.txNumber);
17         if (commitId.equals(this.commitRequest.getId())) {
18           break;
19         } else if (!this.commitRequest.getBenignCommits()
20                     .contains(commitId)) {
21           this.commitRequest.setUndecided();
22           throw new CommitException();
23         }
24         existingRec = nextRec;
25         nextRec = nextRec.getNext();
26       }
27       assignCommitRecord(existingRec.txNumber + 1, getWriteSet());
28       this.commitRequest.setValid();
29     }
30
31     enqueueForWriteBack(existingRec, this.commitTxRecord.getWriteSet());
32   }
33
34   ActiveTxRecord getForwardRecord(ActiveTxRecord record, int version) {
35     while (record.txNumber < version) {
36       record = record.getNext();
37     }
38     return record;
39   }
40 }
```

Listing 5.14: Validation of a read-write `CommitOnlyTransaction`.

```
1 class Transaction {
2   ...
3   void applyPendingCommitsAndUpgradeTx() {
4     this.lastProcessedRequest =
5       helpCommitAll(ClusterUtils.getCommitRequestAtHead());
6
7     upgradeTx(LAST_COMMITTED_RECORD);
8   }
9 }
```

Listing 5.15: The update algorithm that runs when starting a new transaction. It helps to process any pending requests and then upgrades the transaction's own read version.

5.3.4 Starting a New Transaction

In NbTMM, the beginning of a new transaction shares the same concerns that were described in Section 3.6.4 on page 68, namely the effort to begin the transaction in the most recent version possible, and the need to advance the node's local state, even when there are no other committing write transactions to aid in such task. So, when beginning any transaction the system invokes the operation `applyPendingCommitsAndUpgradeTx()`, shown in Listing 5.15: It executes the same helping that is performed when committing write transactions, by calling `helpCommitAll(CommitRequest)`. This time the call uses the commit request at the head of commit requests queue and, when this call completes, it will initialize the transaction's reference to the `lastProcessedRequest`. This ensures that, from this point onwards, the transaction will iterate all commit requests that further appear between its start and its completion, which allows it to improve the list of possible benign transactions to send in its own commit request, in case this is a write transaction.

After processing any pending requests, the transaction upgrades its own read version to the most recent version. Notice that the transaction's read version will include *at least* any version produced by requests that it may have helped to process. However, it is possible that the transaction upgrades to an even more recent version if between the completion of line 4 and the read of the `LATEST_COMMIT_RECORD` in line 7, some new request arrives and is processed by another helper.

5.4 Experimental Evaluation

I repeated the experiments from Section 3.8 on page 77, this time using NbTMM. By using the same test environment with exactly the same execution configurations, the results obtained for NbTMM are directly comparable with the results obtained previously for the other implementations. First, I present and discuss the results comparing TMM with NbTMM, and then I compare NbTMM with the same standard implementation approaches used in Chapter 3.

Globally, the results show notorious improvements over the lock-based design, most visible in the clustered scenarios. Nevertheless, there are still some scalability limitations when stressing the commit algorithm in the clustered configurations (with higher write percentages or higher number of nodes). To better understand the source of these limitations, I perform additional measurements, in particular:

- I instrument NbTMM's commit algorithm to analyze how each phase performs, and show that the performance of the underlying TDR greatly affects the commit algorithms's overall performance, by dominating the total execution time, and increasing its latency by considerable amounts of time. This is caused by delays in the execution of the underlying components that are also aggravated when the size and amount of messages exchanged increases—a consequence of increasing the workload's write percentage.
- To support the conclusions just drawn about the effect of the TDR on the performance of the commit algorithm, I developed and tested the implementation of a *Mock* TDR that eliminates all TDR-related communication costs. The resulting algorithm, which still uses all elements of the commit algorithm, including the communication with the Sequencer, scales continuously for all workloads tested, albeit at different rates, depending on the workload, convincingly showing that the algorithms in NbTMM effectively scale, in spite of being limited by the performance of the underlying components.

I complete the experimental evaluation with two more groups of experiments: First, because I consider important to perform a working demonstration of NbTMM clustered in more that one physical computer, I deploy the experiments in a real cloud computing grid, namely FutureGrid, using one node of the grid to execute each application server. Lastly, I experiment with using data distribution in the TDR, instead of data replication. Even though I have argued before, in Section 3.8.1 on page 79, that this configuration is less fair to the

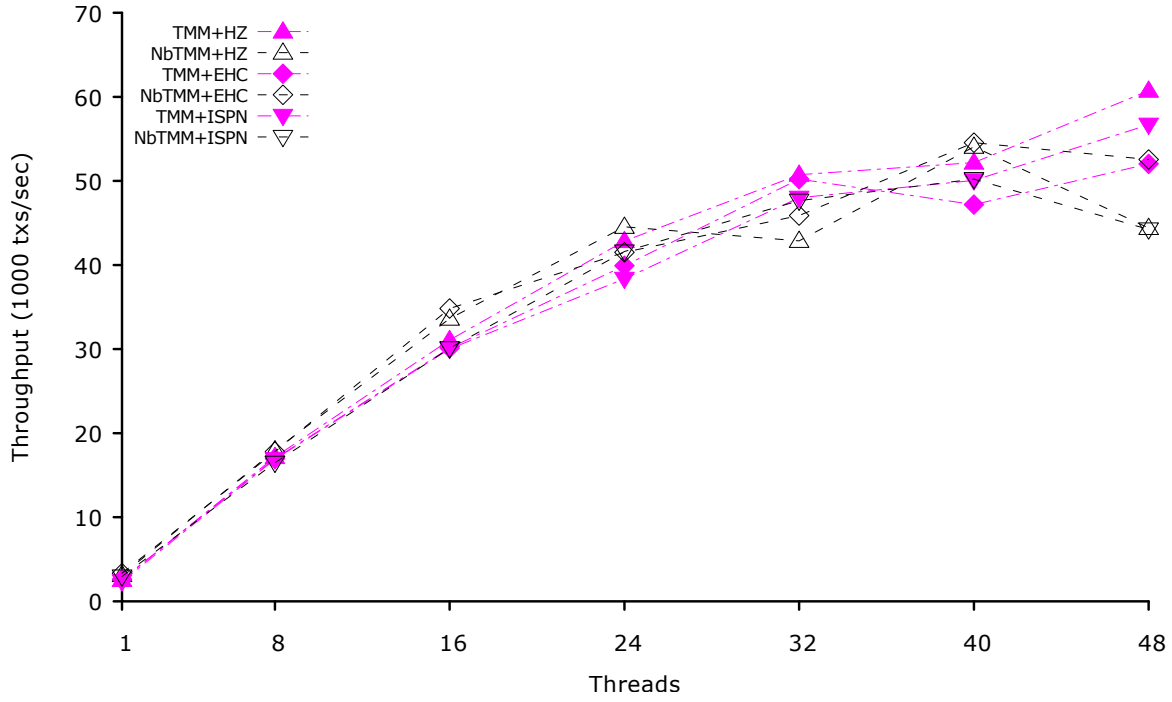

Figure 5.2: TMM versus NbTMM: Total throughput for the single-node executions using 1% write transactions.

other approaches, its purpose is exactly to demonstrate the benefits of NbTMM's adaptive caching, which always stores the node's working set locally, regardless of the data location in the TDR.

5.4.1 TMM versus NbTMM

In Figure 5.2 to Figure 5.4 on pages 174–175, I present the results for the single-node executions. The results obtained in the cluster executions appear in Figure 5.5 to Figure 5.7 on pages 176–177. As before, each percentage of write transactions is grouped in a separate chart. All executions of the NbTMM-based implementations include the deployment of the Sequencer node, even for the single-node configurations, with commit requests being pushed to the Sequencer for publication.

In a single node, the results for NbTMM are, overall, much identical to their counterpart results for TMM. In fact, for 1% write transactions, there are no noteworthy differences whatsoever. This sustains the belief that, for scenarios with very low commit contention, the lock-based approach is still a viable solution. For the 5% write transactions configuration, there starts to appear a slight improvement on NbTMM, in particular for higher thread counts

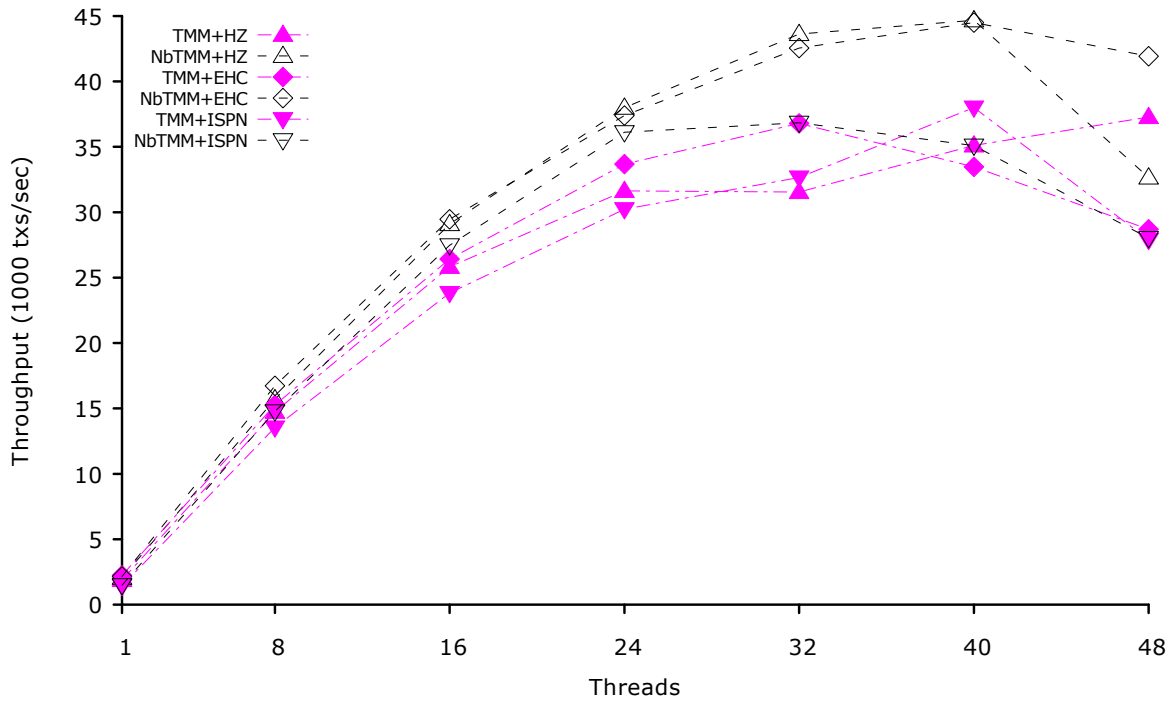

Figure 5.3: TMM versus NbTMM: Total throughput for the single-node executions using 5% write transactions.

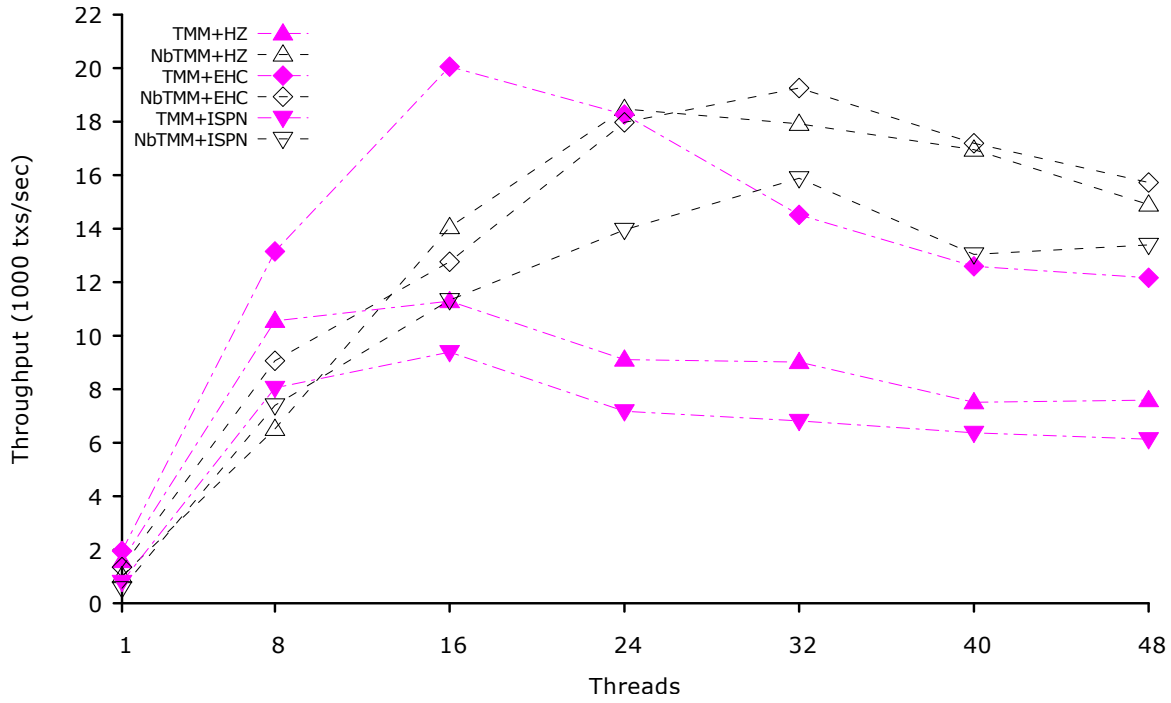

Figure 5.4: TMM versus NbTMM: Total throughput for the single-node executions using 20% write transactions.

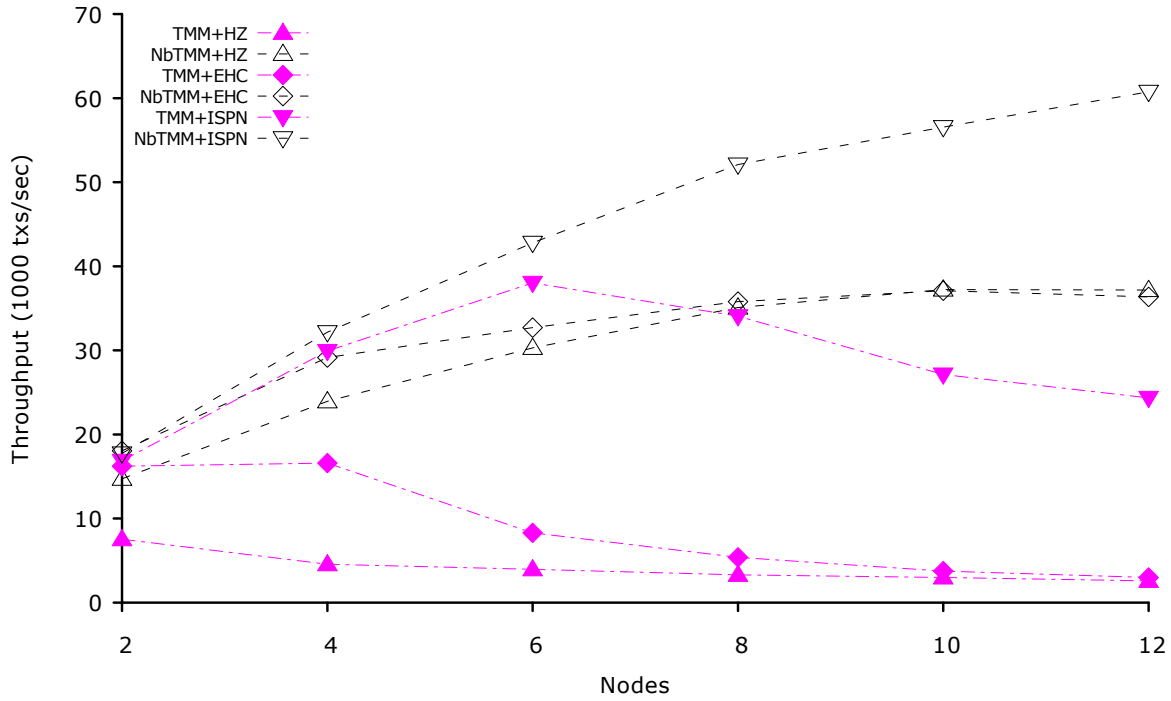

Figure 5.5: TMM versus NbTMM: Total throughput for the clustered executions using 1% write transactions.

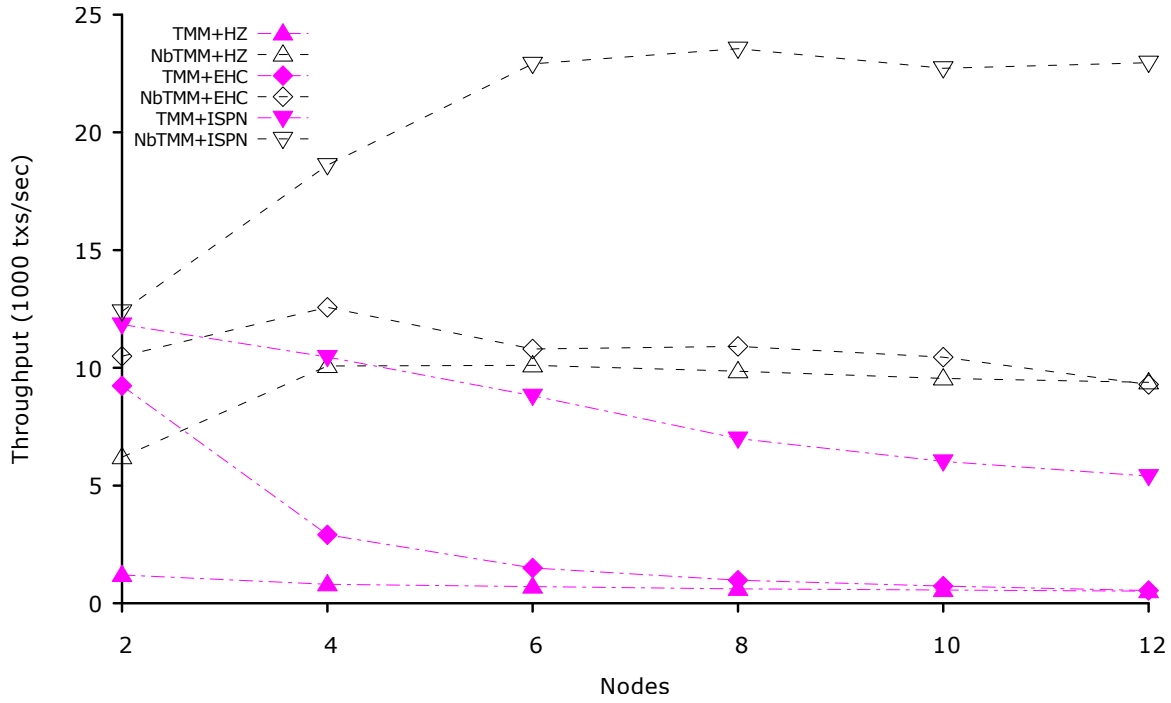

Figure 5.6: TMM versus NbTMM: Total throughput for the clustered executions using 5% write transactions.

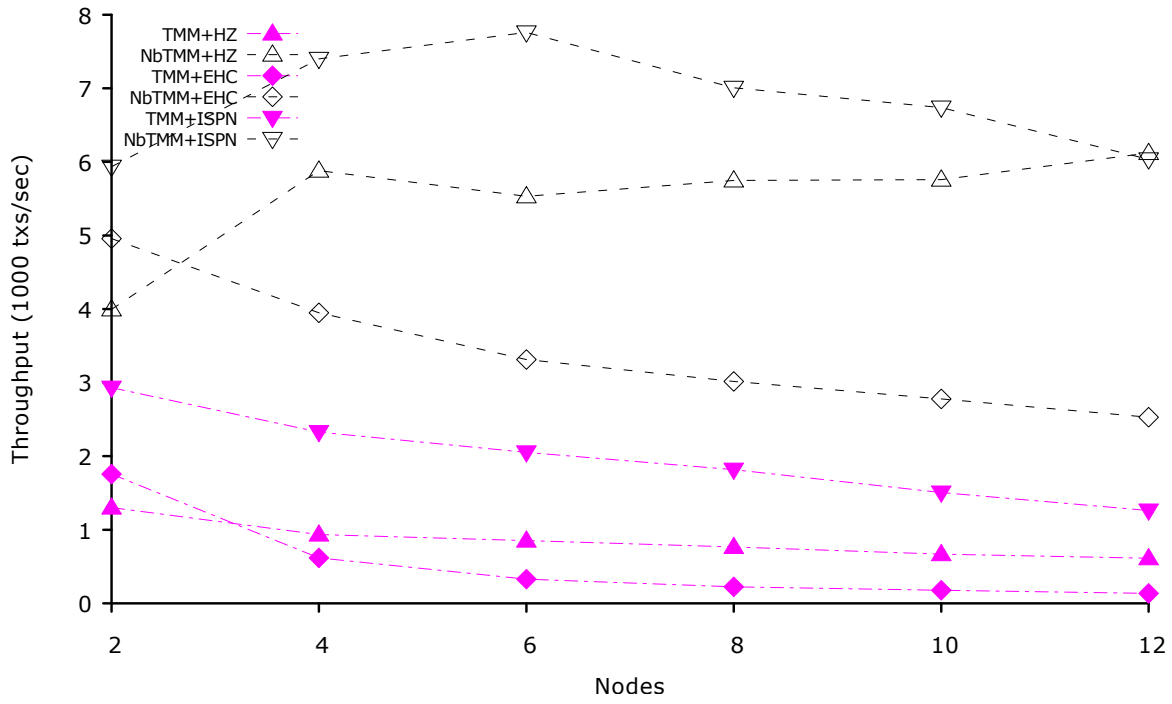

Figure 5.7: TMM versus NbTMM: Total throughput for the clustered executions using 20% write transactions.

(above 24). At 20% write transactions, the improvements in NbTMM are easily distinguishable from TMM: Interestingly, for low thread counts (up to 8), TMM performs better, and it even achieves the best throughput among all implementations, at 16 threads, when using TMM+EHC. Generally, for low thread counts in a single node, TMM appears identical to or, in some cases, better than NbTMM. This is likely due to the higher algorithmic overheads of the nonblocking algorithm, when compared to the much simpler lock-based algorithm that is faster to execute in scenarios with minimal, or nonexistent, lock contention. As thread counts increase, NbTMM eventually overtakes TMM's throughput, due to its higher scalability. Also noteworthy is that the nonblocking implementation appears somewhat unable to continuously improve the application's throughput for very high thread counts. This is, partly, an effect of the load on the test system. The machine has 48 physical cores: On the one hand, already at 40 threads, the total number of threads executing, exceeds the machine's parallel processing ability, because there are other infrastructural threads competing for the resources, such as the GC's. On the other hand, the load put on these infrastructural threads on a single machine is much higher than if it was distributed by several machines, because the cost of the GC grows more than linearly with the load. For example, the memory allocation and collection rates on a single machine running 40 threads will place a much

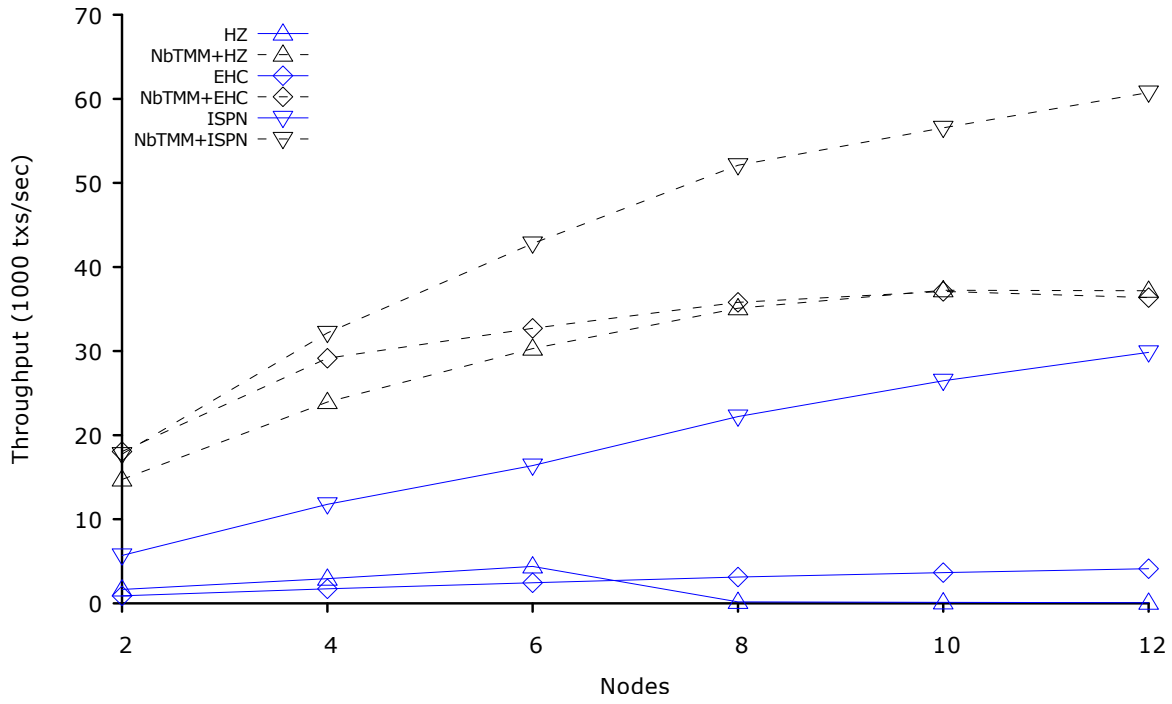

Figure 5.8: NbTMM versus IMDG: Total throughput for the clustered executions using 1% write transactions.

higher load on the GC, than if the same number of threads was running distributed through 10 JVMs running 4 threads each.

When looking at the clustered executions—the main driver for designing NbTMM—it is clear that the nonblocking approach is able to improve the application's performance even further. Already at merely 1% write transactions, the speedup with the NbTMM-based implementations, using 12 nodes, ranges from approximately 2.5 times (ISPN) up to 12 times (EHC). Mostly, the performance of TMM-based implementations starts low and then further decreases as more nodes are added to the cluster, whereas the performance of NbTMM-based implementations starts close or above TMM's, is able to increase for some nodes, and then tends to stabilize when unable to improve after a given number of nodes, which varies, depending on the workload: At 20% write transactions, the absolute performance improvements are notorious, but the scalability is compromised, with even the NbTMM+EHC implementation tending towards a performance drop; as the percentage of write transactions is reduced, the scalability increases—something that does not occur, even for the 1% workload in TMM.

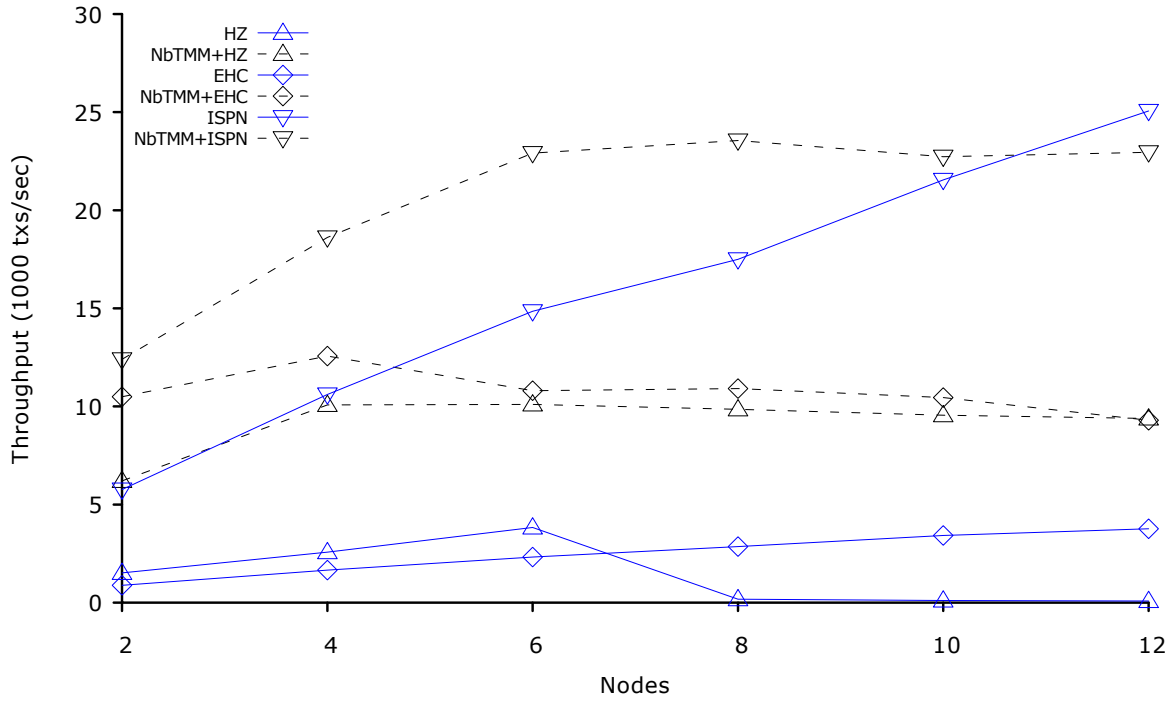

Figure 5.9: NbTMM versus IMDG: Total throughput for the clustered executions using 5% write transactions.

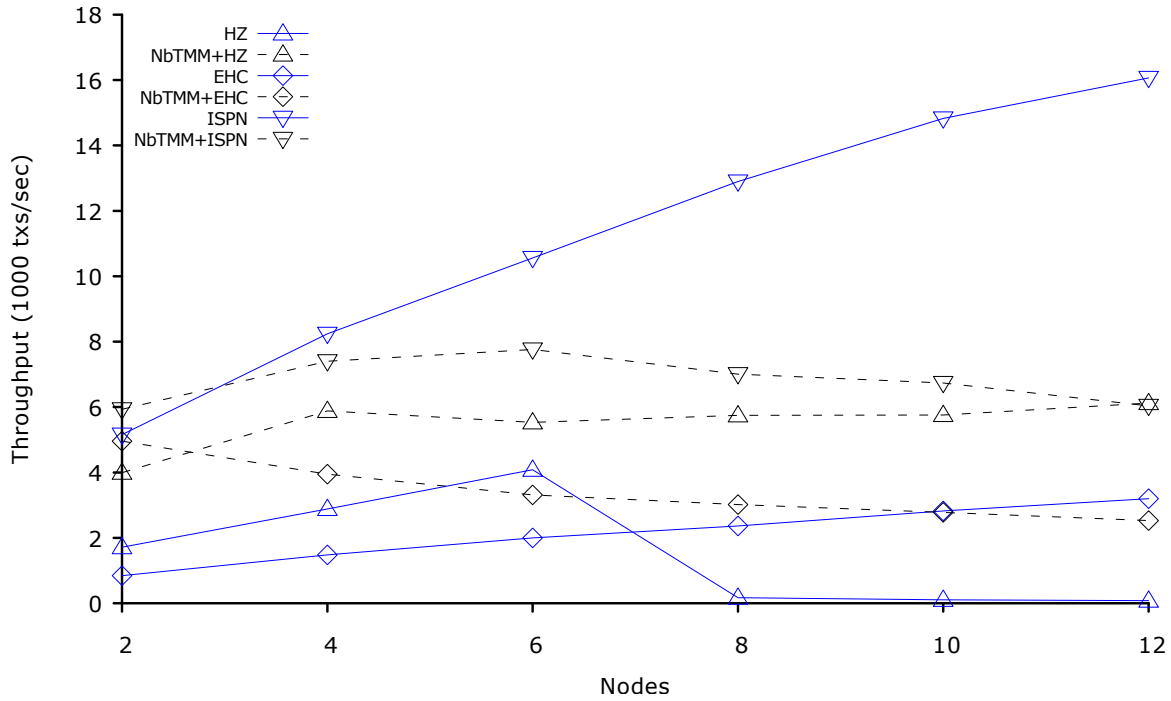

Figure 5.10: NbTMM versus IMDG: Total throughput for the clustered executions using 20% write transactions.

5.4.2 NbTMM versus Standard IMDG

The results from Section 3.8.2 on page 81 show how TMM significantly improves the performance over the standard approaches, for the single-node configurations. In the previous section, the comparison for the same settings shows approximate performance results between TMM and NbTMM in a single node. Together, these data clearly show that NbTMM is also much better in a single-node deployment than the standard approaches, so that repeating the single-node comparison, now between NbTMM and other IMDGs, seems of little additional value.

So, in this section, I shall present the comparison between NbTMM and the other IMDGs, for the clustered executions, which still displays mixed results, even though overall NbTMM tended to be much better than TMM in such configurations. Figure 5.8 to Figure 5.10 on pages 178–179 display the comparison for each workload. Notice that there are no new values presented in these charts: They simply result from the juxtaposition of data from the clustered executions of NbTMM, shown in the previous section, with the clustered executions of IMDGs, shown in Section 3.8.2 on page 81, so that the observations made in those sections, pertaining these configurations, apply equally here.

In the comparison from Chapter 3, TMM-based implementations were not able to improve on their respective IMDG standard implementations, apart from a very reduced set of cases in the workload with 1% write percentage—mainly, ISPN up to 10 nodes. When considering NbTMM, however, there are very different results: It consistently performs better than EHC and HZ executions, even up to 20% write transactions, which is a considerable improvement. In particular, for the 1% write transactions workload, all NbTMM implementations outperform the others. For 5% NbTMM starts much better than ISPN, but they end up matched, and for the 20% case, NbTMM is not able to improve over ISPN at all.

Although, overall the performance results have improved a lot from TMM, and are even better than those of the IMDGs in several cases, it is noticeable that the tendency appears to be for the derivative function of each NbTMM plot to start positive, but tending towards zero: Initially throughput increases as more nodes are added, but the increase in performance brought by adding new nodes is each time less, to the point where, above a certain threshold (which varies with the workload and the underlying TDR), adding more nodes does little or nothing to improve performance. At worst, in some cases total performance even starts to drop a little. This is undesired and, at least in theory, unexpected: Taking into the account that the algorithms in NbTMM are nonblocking, the intuitive idea would be that,

even though latency might increase, thereby causing a reduction in throughput per thread, performance should be able to continue to scale with the number of threads. The fact that this does not occur, leads to the intuition that some of the external components, on which the solution is laid—the TDR and NbTMM's communication infrastructure based on ZMQ—could be the source of some contention. ZMQ was chosen as the underlying communication infrastructure, because it has shown very low latency and high throughput in previous benchmarks (cf. Section 5.3.1 on page 154), which turned my attention to the TDR as more likely to be the source of contention. Another factor pointing towards this educated guess, is that overall application performance is affected when changing the TDR.

5.4.3 Profiling the Commit Algorithm

With the purpose of identifying where the potential bottleneck may reside, I instrumented the commit algorithm with probes to measure the total commit time, as well as the time that transactions spend executing each of the commit steps, such as validation, persist the write set, broadcast a commit request, help to commit a transaction, and persist the mapping of a version to a commit identifier. Most of the commit times that were measured, displayed little variations across different workload configurations, except for the times related to communications with the TDR. Figure 5.11 to Figure 5.12 on page 182 display the average total commit time per transaction—shown as the height of each bar in the plots—for each of the write workloads in the clustered configurations. Every bar comprises two segments, each representing:

1. The time spent making calls to the TDR, which includes only the time used in the following two operations:

   (a) Persisting the write set of the transaction: Operation `persistWriteSet()` invoked in line 9 of Listing 5.5 on page 156.

   (b) Persisting the mapping with the commit identifier to the TDR: Operation `persistTxVersionToCommitId(int, UUID)` invoked in line 17 of Listing 5.13 on page 169.

2. The remainder of time spent in the commit of a transaction. It is relevant to mention that this includes the time spent in calls to the communication infrastructure.

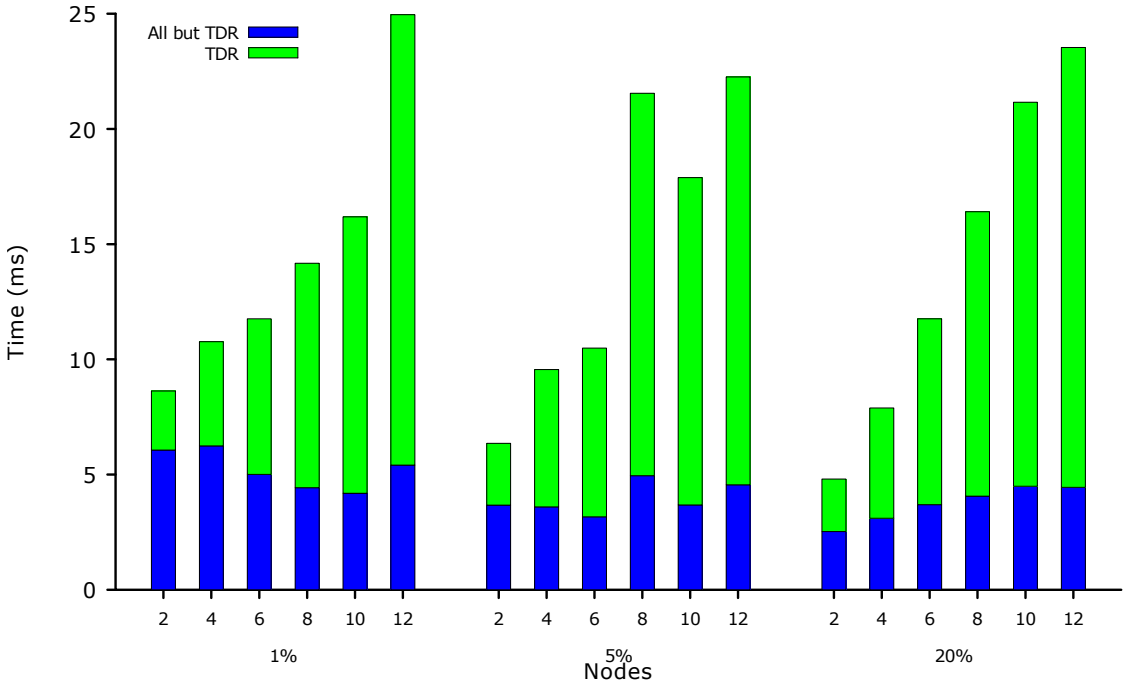

Figure 5.11: Commit times for each workload in NbTMM+ISPN. Each bar represents the average total commit time per transaction. The time spent in calls to the TDR—the upper segment in each bar—increases with the number of nodes, regardless of the workload, wheres the rest of the time remains almost unchanged.

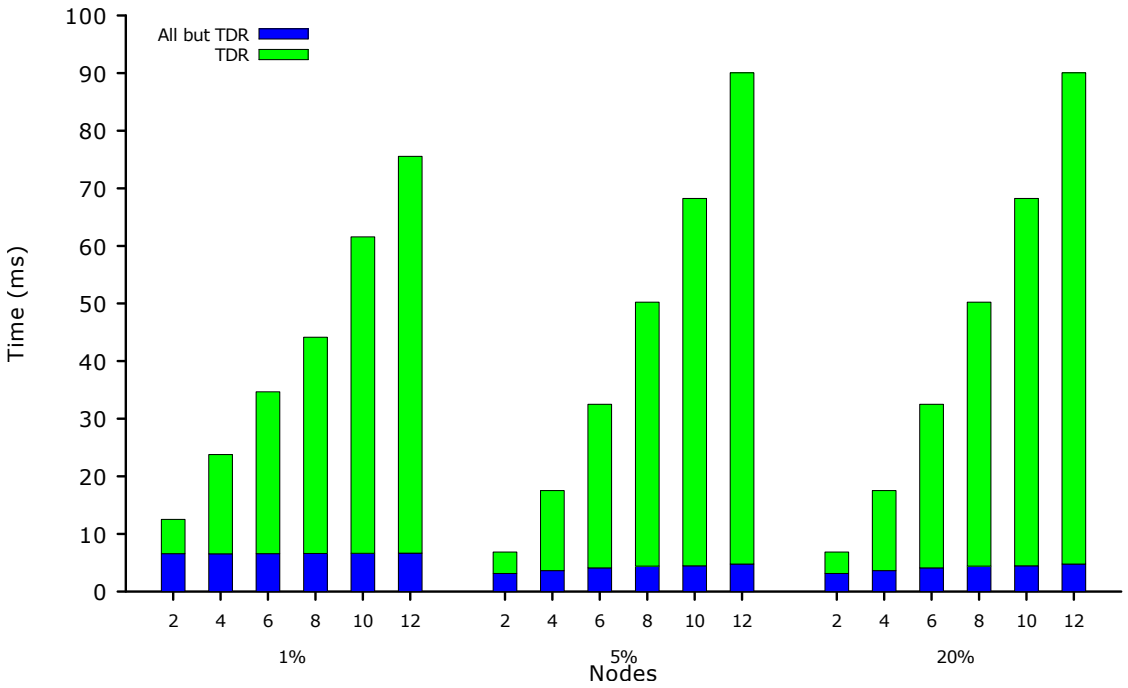

Figure 5.12: Commit times for each workload in NbTMM+EHC. Commit times display the same pattern as those for NbTMM+ISPN.

Figure 5.11 on page 182 shows the results obtained for all the clustered workload configurations using the implementation NbTMM+ISPN. It is clearly visible that the time taken by all operations excluding the time spent in calls to the TDR is approximately the same for all workloads, with minor variations, whereas the time spent interacting with the TDR greatly increases as the number of nodes increases, regardless of the workload. I speculate that this is likely due to the need to synchronize the state of the nodes when data is written, which is imposed by NbTMM to ensure that data is correctly available on any other node after the underlying write returns. Often, IMDGs are built with a bias towards data distribution, which means that the number of nodes to keep in synchronization for each datum update is usually limited to a number smaller than the size of the entire cluster. NbTMM's design stresses the synchronization protocols of the TDR, especially under full replication. Also, the fact that performance of the calls to the TDR does not seem to depend on the workload, is consistent with the performance of the ISPN-only implementation, which displays little performance variations, when changing the write percentage (cf. the previous plots for ISPN-based throughput in the clustered configurations).

The performance of EHC and HZ are overall similar, and both are much worse than ISPN. In Figure 5.12 on page 182 I repeat the same measurements, now taken with for NbTMM+EHC. These confirm the same observations as before. Interestingly, here the time spent in calls to the TDR is much higher (about four times more), but the remainder of the commit algorithm takes approximately the same time as it did with NbTMM+ISPN (around 5ms on average).[44] Again, this is consistent with the idea that the overall performance of the commit algorithm is mostly affected by the concrete TDR. These results also help to explain the better performance of the ISPN implementation over the others.

5.4.4 The Mock IMDG

Considering the previous analysis, a question that may still linger is whether the time spent in calls to the TDR is the sole factor limiting NbTMM's overall scalability. To answer such question, I developed another RadarGun plug-in extension to simulate an IMDG that does not involve any synchronization costs between its nodes. In practice, this is a mock implementation that does not send any updates to the other nodes. This can be done without affecting the semantics of the application under test, because it leverages on the fact that the workloads that I use, as mentioned in Section 3.8.1 on page 79, access disjoint data: Each

[44]The small variations among different workloads, which were observed in NbTMM+ISPN, are less visible in NbTMM+EHC, because the vertical axis spans a greater interval in the latter case.

thread in each node accesses random keys within a predefined set that does not intersect with any other thread's key set.

Thus, the implementation of RadarGun's `CacheWrapper` by the Mock IMDG could simply become an empty operation. I need, however, to introduce delays on the `CacheWrapper`'s operations, to simulate the delays in reading and writing entries, lest the simulated IMDG display an unbeatable performance in terms of its measured throughput. Taking a random set of results from ISPN in a single node, I computed the average time taken by the read and write operations, and then introduced those delays as active waiting cycles in the implementation of the Mock. Each `get` introduces a delay of 2.8 microseconds and each `put` introduces a delay of 67 microseconds. Also, there is an approximately constant delay of 64 microseconds per transaction, which I added to the operation `commitTransaction()`. The implementations of the remaining operations is empty during the benchmarking stage.

To properly execute RadarGun's stages, I had to introduce additional complexity to the Mock implementation. Such was necessary because, immediately after starting each node of the cluster, RadarGun's master node instructs all other nodes to validate their presence in the cluster, which entails having each node write to a predefined key, and all nodes confirming that they can read the keys written by all other nodes. So, in fact, the implementation of the Mock must behave as a real TDR during the initial bootstrap phase. My solution was to adopt the ISPN's `CacheWrapper` implementation and modify it so that during a fixed number of operations it uses the real Infinispan IMDG, and then it switches to the mock mode with the simulated operations, described above. This provides enough time for the nodes to correctly start, and eventually, during the warm-up phase, the operations transparently stop making calls to Infinispan's data grid. By the time the real benchmarking phase begins, the underlying IMDG is no longer being used.

Figure 5.13 on page 185 presents the results comparing the best performance possible for the MOCK implementation—the read-only scenario, shown as MOCK-0—with the execution of the NbTMM+MOCK implementation for the workloads ranging from read-only up to 20% write transactions. Notice that whereas in the MOCK there is no network communication involved whatsoever during the benchmark stage, in NbTMM there is still the communication of commit requests to all nodes, broadcast via the Sequencer. In spite of that, the NbTMM implementation scales consistently and is able to provide much higher throughputs than the baseline MOCK-0, even for 20% write transactions. The absolute throughputs decrease as the write percentage increases, mostly for two reasons: (1) There is an increase in the network communications to broadcast commit requests, which occurs increasingly often, and (2) the

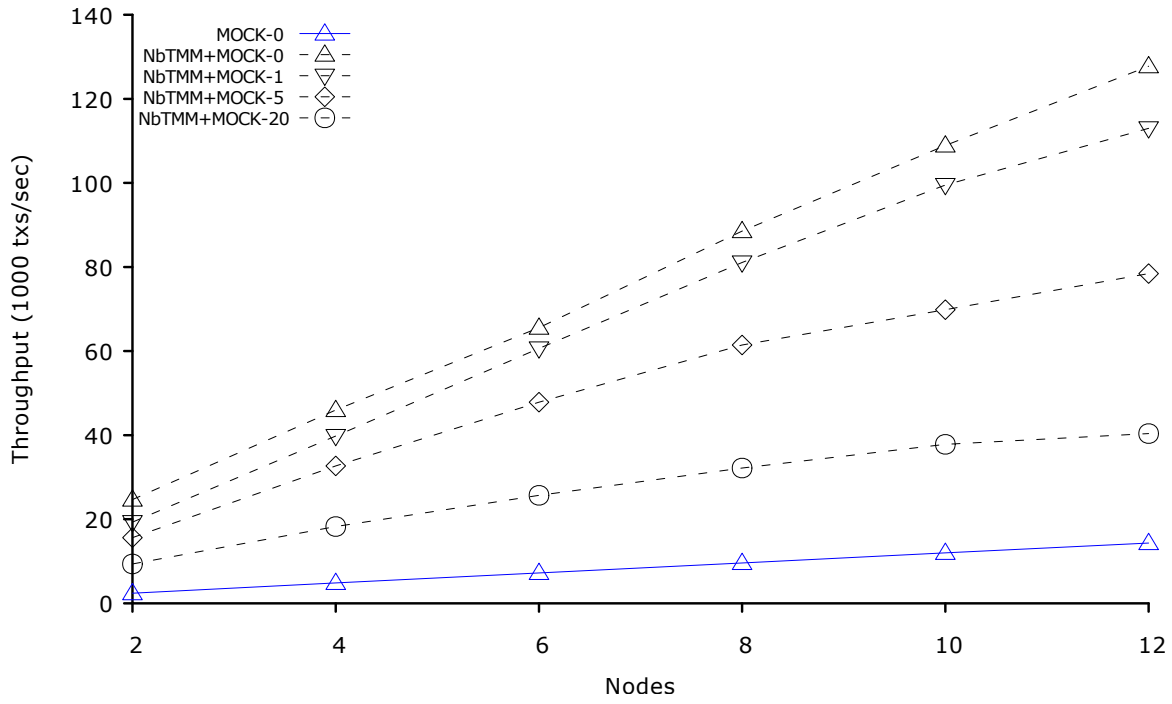

Figure 5.13: NbTMM versus MOCK: Total throughput for the clustered executions with multiple workloads.

queue of commit requests to process in each node also increases, while maintaining the same number of application threads per node (four), which requires more helping effort per transaction.

These results clearly show that the nonblocking algorithms proposed in NbTMM scale and consistently improve the application's throughput, absent the limitations imposed by the calls to the TDR's operations.

Given that the performance of the calls to the TDR is the source of such limitations, yet another question arises: How does it affect TMM? Recall that the experimental results from TMM in Chapter 3 show that, while the contention on the lock is low, it performs well—which is mostly visible in a single node. Under these circumstances, one might question whether TMM would be able to outperform NbTMM, assuming the zero-cost to use the TDR. In Figure 5.14 on page 186, I present the comparison between TMM and NbTMM using the MOCK TDR. Even without the cost of the calls to the TDR, the network communications involved while the lock is acquired take long enough so that contention still occurs: This is visible in that, again, NbTMM is consistently much better than TMM for all write workloads. As expected, they perform identically in the read-only scenario. Then, as the rate of write

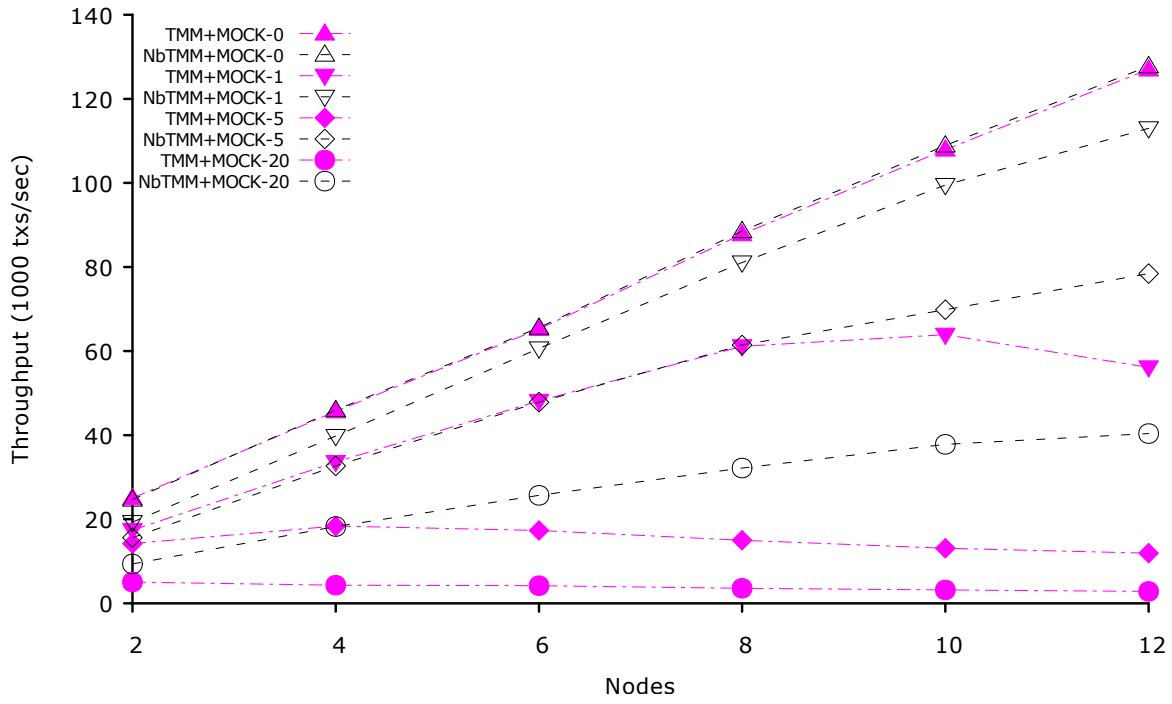

Figure 5.14: TMM versus NbTMM using MOCK. NbTMM is consistently better, which is more noticeable as the percentage of writes increases.

transactions increases, so does the performance gap between the two designs.

5.4.5 FutureGrid Cloud Platform

All results presented up to this point have been taken in a single machine with 48 cores. When deploying multiple nodes, these were communicating through the kernel's loopback device. To assess whether this has any relevance on the results obtained, I also repeat some of the tests in a cloud computing environment provided by the FutureGrid project.[45]

FutureGrid is built from a geographically distributed set of heterogeneous computing systems. The tests were executed in the India cluster, hosted at the Indiana University, USA. This cluster uses the OpenStack cloud computing software to provide an *Infrastructure as a Service* (IaaS) platform. It provides several *Virtual Machine* (VM) configurations to use. The experiments with RadarGun executed on a VM-based environment, deploying each application node in a separate VM. Each VM is configured with 8GB of main memory, 4 Intel X5550 CPUs, and all nodes are connected using an InfiniBand network. For some reason that has eluded me, I was not able to correctly configure Hazelcast in FutureGrid, as the members of

[45] https://portal.futuregrid.org/

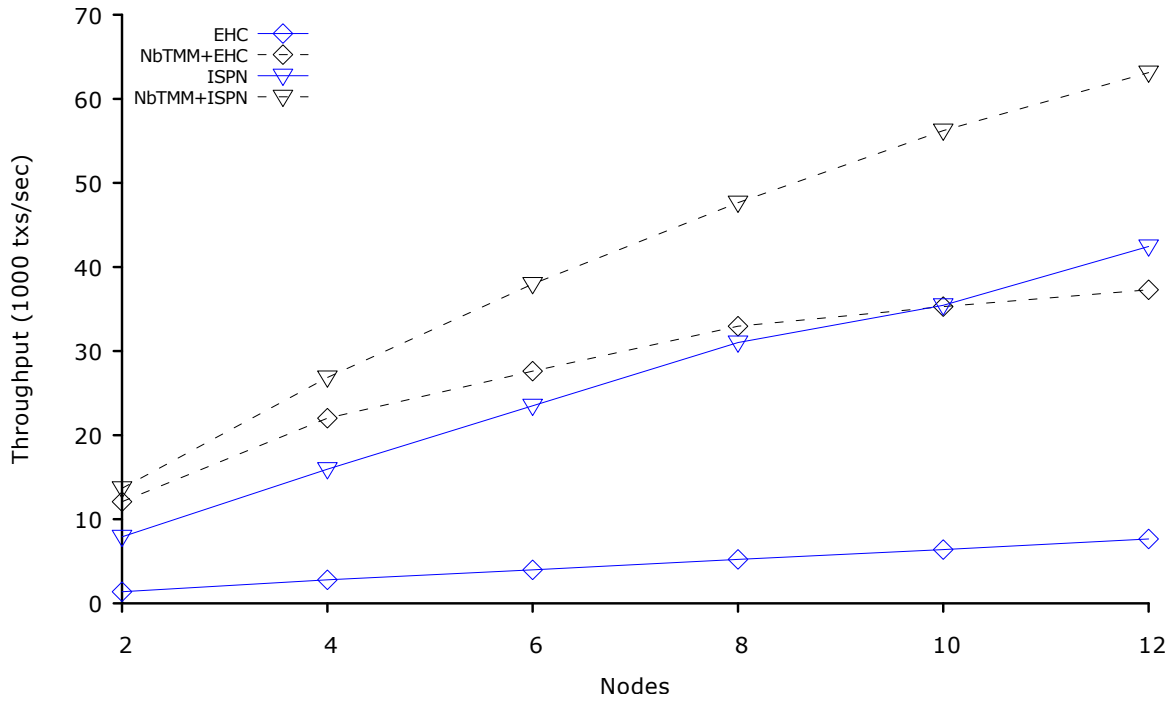

Figure 5.15: NbTMM versus IMDG in FutureGrid: Total throughput for the clustered executions using 1% write transactions.

the cluster were never able to find each other. For this reason, the following results include only the plots for ISPN and EHC.

In Figure 5.15 to Figure 5.17 on pages 187–188, I present the results obtained when running the three write workloads in FutureGrid, respectively from 1% up to 20%. These results are extremely similar to those presented previously in Figure 5.8 to Figure 5.10 on pages 178–179 for both the IMDGs and their corresponding NbTMM-based implementations. If anything, I would comment that there is a slight performance improvement favoring the IMDG-based implementations, but overall the results are identical, showing that the test environment, based on a single machine, did not skew the results.

5.4.6 Using Data Distribution

When I presented the base configuration of all experiments, in Section 3.8.1 on page 79, I justified that using full data replication was more appropriate than distribution, to compare my approach with the others, because the workloads were read-dominated and the caching mechanism from my design operates based on accessed data and not on their location in the TDR. The expectation is that, when using data distribution in the TDR, the performance

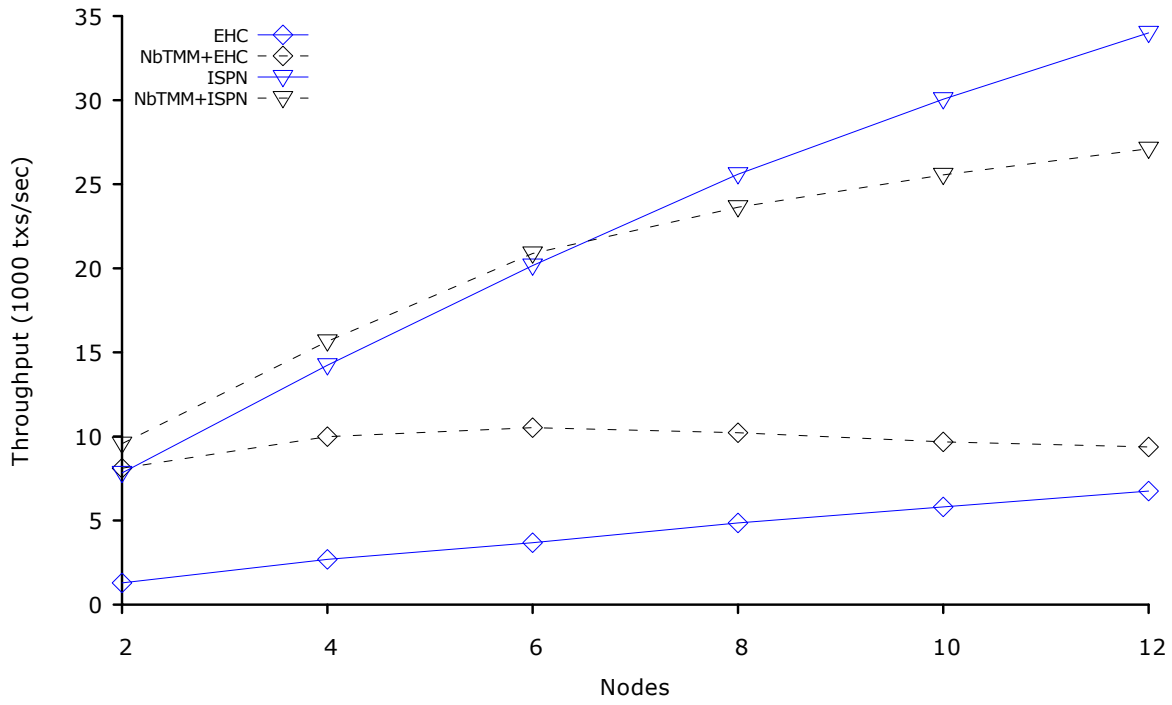

Figure 5.16: NbTMM versus IMDG in FutureGrid: Total throughput for the clustered executions using 5% write transactions.

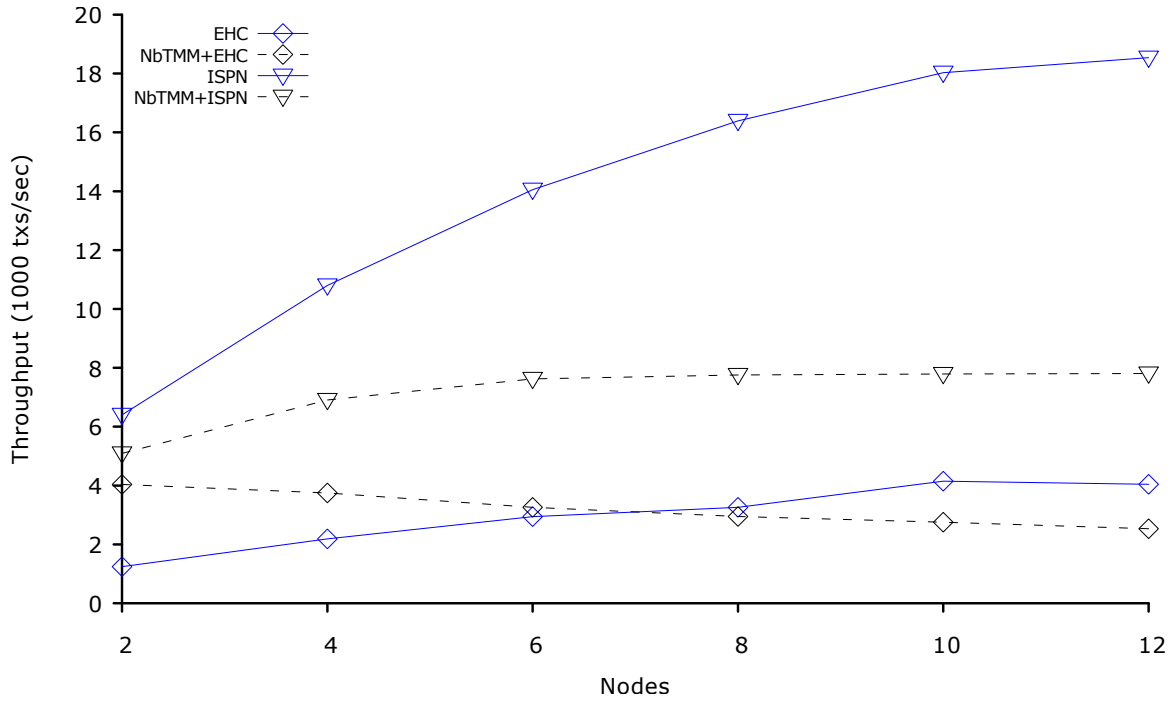

Figure 5.17: NbTMM versus IMDG in FutureGrid: Total throughput for the clustered executions using 20% write transactions.

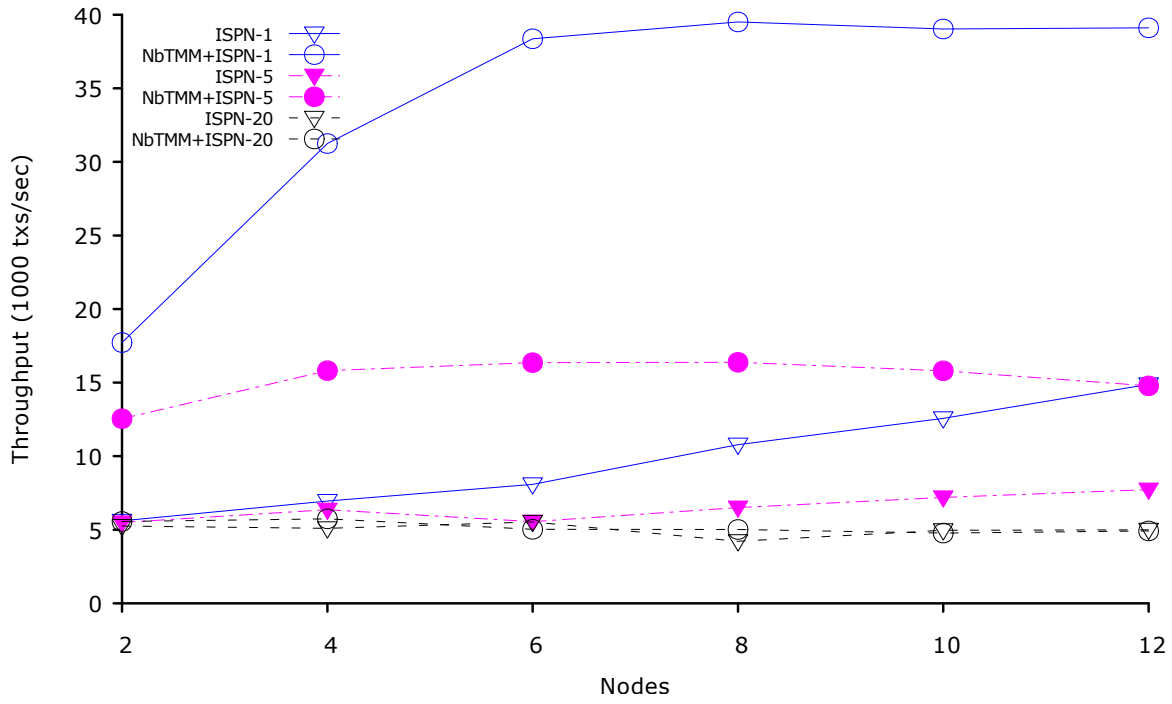

Figure 5.18: NbTMM versus ISPN in distributed mode. Only two nodes store each datum.

improvements from using NbTMM for read-dominated workloads should be even higher. Nevertheless, there is an important trade off to consider: When running in distributed mode, the IMDG makes fewer calls to synchronize its state, whereas NbTMM's commit protocol must still send commit requests to all members of the cluster. This could negatively affect the expected performance gains.

In this section I pick the best-performing IMDG among the ones tested—ISPN—and configure it to work in distributed mode, using two nodes for replication of each datum. Additionally, ISPN uses an L1 cache that stores locally the keys read from remote nodes.

Figure 5.18 presents the performance results obtained for each of the write workloads, using both ISPN and NbTMM+ISPN. As expected, for these workloads the performance of ISPN using distribution is worse than what was obtained under full replication. This worse performance affects NbTMM, which also performs worse than before when it was using a TDR with full replication. Nevertheless, the relative improvements brought about by using NbTMM are actually higher that before. Considering the relative performance improvements of using NbTMM+ISPN over ISPN, then Figure 5.19 on page 190 shows the relative increase in performance between using distribution and replication. In all cases, all ratios are above 1,

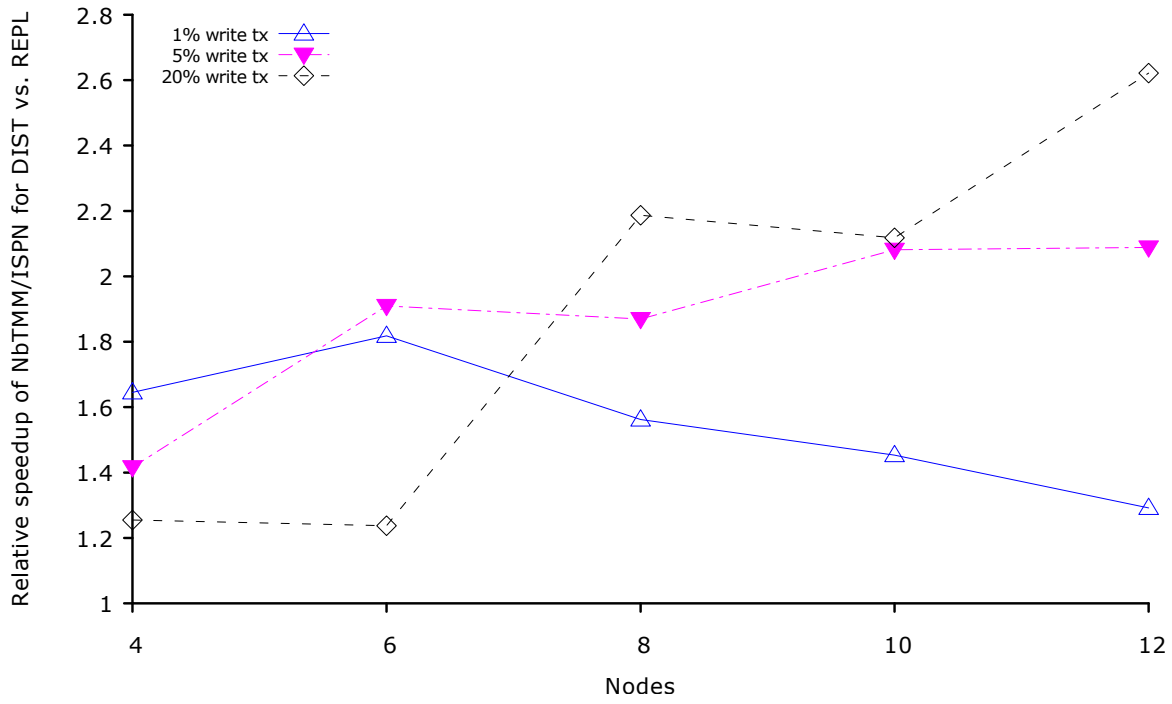

Figure 5.19: Relative performance improvements obtained with NbTMM when the TDR uses data distribution instead of full replication.

meaning that the improvements of using NbTMM over a distributed configuration are higher than the improvements obtained when using NbTMM over a replicated configuration.

Even tough NbTMM pays a higher price than ISPN to commit, because in addition to waiting for ISPN to commit its own state, it must still contact all nodes to send the commit request, the read operations from NbTMM's cache are much faster and largely compensate for the cost of the writes, because reads occur much often. ISPN's L1 cache allowed it to reduce the number of remote lookups by maintaining a local copy of the data but, nevertheless, its reads are still much slower.

5.5 Summary

The goal of this chapter is to improve the design of TMM for the clustered scenarios. To this effect, I combine the main concepts and principles from Chapter 3 and use the lock-free JVSTM presented in Chapter 4 to design NbTMM, a nonblocking alternative to TMM.

The structure of the presentation in this chapter is similar to that of Chapter 3, in which I first overview the main aspects of the solution, and then present the algorithmic details of

each element. Some parts of the solution proposed in Chapter 3 have been reused, others extended or completely redesigned to use nonblocking algorithms in all elements of NbTMM.

After presenting NbTMM, I conduct several experiments to assess the performance of this new design. I start by repeating the experiments from Section 3.8 on page 77 with NbTMM. The single-node executions show that performance is approximately identical to that of TMM, which is understandable given that TMM's lock-based solution already proved usable in a single node. The executions in clustered scenarios provide mixed results: In spite of clearly improving over TMM, as well as over a few IMDGs, the most striking aspect was that scalability did not match the theoretical expectations.

The previous observations led to a more detailed analysis of the commit algorithm. After profiling its execution it became apparent that the duration of the calls to the TDR increases as the number of nodes increases, which limits NbTMM's scalability.

Tests performed with a simulation of a TDR that does not incur into communication costs reveal that when these costs are removed application performance is able to scale with the number of nodes, confirming the limitations imposed by calls to the TDR during the commit of write transactions.

The evaluation concludes with two additional results: (1) The confirmation of the previous results using a cloud computing platform, and (2) a proof-of-concept demonstration that the potential for improving application performance with NbTMM is higher when the underlying TDR uses data distribution instead of full replication.

The results also show that NbTMM is not a complete replacement for TMM. Instead it is a complement, and each can be used whenever it provides the best performance: Even though NbTMM is overall better for improving the performance of clustered applications, TMM's simpler design still allows it to outperform NbTMM, mostly in single-node configurations.

6 Conclusions

In this final chapter, I summarize the main results and contributions of this dissertation, and I present some ideas for future research that stem from this work's current state of development.

6.1 Main Contributions

The work in this dissertation stems largely from the perception that, often, developers need to overcome existing limitations in the semantics of the transactional systems, by tangling into application logic the necessary verifications to ensure correct executions of concurrent transactions. Additionally, the current multitiered layout of enterprise applications increases the difficulty to develop strongly consistent transactional systems that are capable of offering acceptable performance, in part because complex application logic executes in a component that is separate (even by a different tier) from the component that ultimately ensures transactional semantics.

The primary goal of this dissertation is to improve the consistency guarantees that the transactional system provides to the programmer, ideally without sacrificing performance. There are some factors that have been key to enable such work:

- Recent technological advancements in hardware have brought affordable many-core systems with large amounts of main memory. These allow application servers to process many concurrent requests and operate locally over large amounts of data.
- Research on STM, focusing on strong consistency for in-memory transactions.
- Nonblocking programming techniques that take advantage of parallel architectures.
- Observation of the data access patterns in enterprise applications, showing predominantly read-dominated workloads (less than 5% write transactions in the use cases

presented in Chapter 2) and very few conflicting operations within concurrent write transactions (less than 2%).

The combination of these factors has enabled the development of this work's main contribution: The design of a middleware that provides strong transactional consistency while, at the same time, being able to improve the performance of the typical read-dominated workloads observed in enterprise applications.

In Chapter 3, I presented such design together with its complete implementation. The results obtained from the experimental evaluation have revealed striking improvements to the performance of read operations, and even notorious results for workloads with write percentages as high as 20%.

These good results, however, apply only to single-node deployments. The usefulness of this design in a clustered environment is exclusively for the sake of providing strong consistency. While, by itself, it may be a sufficient reason to use this design in a very small cluster, greatly increasing the number of nodes is of little use, because this design lacks scalability, given that it is based on the JVSTM's lock-based approach to the commit phase.

Following these first results, I have concentrated my attention on rethinking the initial design, with the purpose of improving it for the clustered environment, while maintaining its fundamental design principle—to embed transaction control into the application using a strongly consistent STM. To this effect, I undertook two main tasks: First, I redesigned the lock-based JVSTM into an efficient lock-free STM. This contribution resulted in a completely usable standalone STM implementation, which has been used already to pursue other research directions, e.g. [Diegues and Cachopo, 2013; Carvalho and Cachopo, 2013]. To the best of my knowledge, this is the first entirely lock-free implementation of a multi-version STM. Second, I took the lock-free JVSTM as a starting point, and proposed a new middleware design, this time using a nonblocking approach, driven by the lock-free JVSTM. The experimental evaluation of the new implementation supported the following conclusions:

- The single-node executions were generally similar in performance to that of the initial design. Nevertheless, in scenarios that were less strenuous for the commit algorithm, the simpler lock-based solution still provided for the best application performance.
- The clustered executions showed improved performance, even though its scalability was still somewhat affected.

- Further profiling of the nonblocking commit algorithm and experimentation with a simulation of the underlying TDR revealed that this new design effectively allows for performance to scale when increasing the number of nodes, albeit its performance is limited by the cost of the calls to the TDR.

- If available memory supports it, and taking into account that operations are read dominated, a configuration using a replicated TDR allows for better overall application performance. In spite of that, if some constraints impose data distribution in the TDR, then the measurements taken, indicate that this middleware applied to a distributed TDR improves the performance even more than when it is applied to a fully replicated TDR, making it a possible choice in both scenarios.

Finally, a side contribution of my work has been the presentation, in Chapter 2, of digested information about the workload patterns, collected over several years, of two real-world enterprise applications—FénixEDU and •IST—which I believe to be representative of a large class of applications, thus providing motivation for the practical applicability of my work.

6.2 Future Work

Both middleware designs that I describe in this dissertation are fully implemented and operational. However, there is still much room for improving them in different areas. The following are some ideas for possible improvements and future research directions, in no particular order.

Improve the properties of the distributed middleware. In this dissertation, I dedicate my attention to the algorithmic aspects of the middleware aiming to support strongly consistent transactions in enterprise applications, which is based on extending an STM with the required mechanisms to support persistent state and clustered deployment. I assume a reliable underlying communication's network and I do not delve into the complexities of distributed systems with regards to properties such as reliability and availability. For example, in TMM if a node holding the global lock crashes, then the commit of further write transactions will block indefinitely, whereas in NbTMM if a node crashes, the others will be able to continue without it. However, if instead of a full crash, a node is just temporarily disconnected and looses some commit requests it will become out of sync and, as it continues

to operate in the cluster, it will likely affect the correctness of future transactions, by validating transactions in an incorrect state. I consider that all issues related to the distribution aspects of the proposed middleware clearly present relevant research topics, which would contribute further to enhance its qualities.

Design a NbTMM-friendly TDR. As identified in the experimental evaluation of NbTMM, its scalability is affected by how the calls to the TDR perform. However, there are specific characteristics about the operations that NbTMM performs that could be leveraged to optimize the internals of the TDR. For example, writes never conflict. Any regular TDR will be unaware of this fact and most likely incur into unnecessary costs to check for possible conflicts among concurrent transactions. Also, different nodes may attempt to write the same mapping from a transaction's version to its commit identifier, concurrently. On the one hand, this may cause a (semantically false) conflict to be detected in the TDR, leading to a costly restart of the underlying operation. On the other hand, the TDR may simply push the overwrites of the same key to all other nodes, without realizing that the value is the same and, thus, it is unnecessary to resend it. I strongly believe that a customized TDR, optimized for NbTMM's access characteristics, would greatly reduce its usage costs, mainly by being able to perform better in each node, and by reducing the number and payload of its network communications.

Provide middleware-specific object-oriented support. When starting to present the middleware, I defined a `Repository` abstraction that contained a minimum set of operations common to all underlying storage systems, allowing me to abstract from the specificities of each TDR. This abstraction also represented the API used by the application's business logic to access its data, either to read or write. However, domain-rich and complex enterprise applications are often developed using an object-oriented approach, and they use richer concepts for modeling their data structures, such as entities, attributes and relationships. In particular, data navigation may start with a key-based query to obtain some object reference, and then proceed by following object references. To smooth the integration of this middleware in such applications, instead of requiring the development of an ORM-like layer that can convert object navigation into calls to the bare `Repository` interface, it makes sense to enrich this middleware with specific object-oriented support. In fact, the Fénix Framework[46]—which was initially extracted from FénixEDU—is already several steps into this direction, by providing an object-oriented *Domain Modeling Language* (DML) that can be used to generate domain entities and relationships, which in turn provide a transactional shared state that is

[46] `http://fenix-framework.github.io/`

supported using JVSTM's VBoxes, and whose object navigation transparently interacts with the transactional middleware. I have been involved with the recent restructuring of the Fénix Framework, in the pursuit of this goal already. Even though such work has fallen out of the scope of this dissertation, it is a path that I intend to continue to follow.

Develop algorithms for garbage collection in the TDR. The design I proposed keeps all versions in the TDR. Even though this has a practical utility, such as executing transactions that access past versions of the application's data, which could be used, for example to audit an application, there may be usages where it is desirable to delete unaccessible versions from the TDR, much like they are already garbage collected from main memory. How to best design GC algorithms for persistent data is another interesting open research issue.

Optimize the loading of older data from the TDR. The algorithm used by NbTMM to access any version of a datum to load it into main memory requires keeping a map of transaction versions to commit identifiers, and to consult this map before attempting to read a VBoxBody. Especially when accessing older versions of a datum, this mechanism can introduce a significant overhead on the reload operation. I have given some thought to this issue, and even started to design an algorithm for it. However, in the experiments I undertook, this has never presented a performance bottleneck, for which reason it has been left as a future improvement.

Bibliography

Adya, A. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. Ph.D. thesis, Massachusetts Institute of Technology, 1999.

Alomari, M., Cahill, M., Fekete, A., and Rohm, U. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*, pages 576–585. IEEE Computer Society, Cancun, Mexico, 2008.

Ansari, M., Kotselidis, C., Watson, I., Kirkham, C., Luján, M., and Jarvis, K. Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory. In *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing (ICA3PP '08)*, pages 196–207. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69500-4. doi:http://dx.doi.org/10.1007/978-3-540-69501-1_21.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*, pages 1–10. ACM, San Jose, CA, USA, 1995.

Bernstein, A., Lewis, P., and Lu, S. Semantic conditions for correctness at different isolation levels. In *Proceedings of the 16th International Conference on Data Engineering (ICDE '00)*, pages 57–66. San Diego, CA, USA, 2000.

Blasgen, M., Gray, J., Mitoma, M., and Price, T. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, 1979. ISSN 0163-5980. doi:10.1145/850657.850659.

Brantner, M., Florescu, D., Graf, D., Kossmann, D., and Kraska, T. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 251–264. ACM, Vancouver, Canada, 2008.

Brewer, E. A. Towards Robust Distributed Systems (Invited Talk). In *Symposium on Principles of Distributed Computing (PODC)*. 2000.

Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., et al. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (USENIX ATC'13)*. 2013.

Cachopo, J. *Development of Rich Domain Models with Atomic Actions*. Ph.D. thesis, Instituto Superior Técnico/Universidade Técnica de Lisboa, 2007.

Cachopo, J. and Rito-Silva, A. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006. doi:10.1016/j.scico.2006.05.009.

Cahill, M. J., Rohm, U., and Fekete, A. D. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4):20:1–20:42, 2009. doi:10.1145/1620585.1620587.

Carvalho, F. M. and Cachopo, J. STM with transparent API considered harmful. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'11)*. Deakin University, Australia, Melbourne, Australia, 2011.

Carvalho, F. M. and Cachopo, J. Lightweight identification of captured memory for software transactional memory. In *Algorithms and Architectures for Parallel Processing*, pages 15–29. Springer, 2013.

Carvalho, N., Cachopo, J., Rodrigues, L., and Silva, A. R. Versioned transactional shared memory for the FénixEDU web application. In *Proceedings of the 2nd Workshop on Dependable Distributed Data Management (WDDDM '08)*, pages 15–18. ACM, Glasgow, Scotland, 2008. doi:10.1145/1435523.1435526.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264. USENIX Association, Berkeley, CA, USA, 2012. ISBN 978-1-931971-96-6.

Couceiro, M., Romano, P., Carvalho, N., and Rodrigues, L. D2STM: Dependable distributed software transactional memory. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '09)*, pages 307–313. Shanghai, China, 2009.

Dalessandro, L., Spear, M. F., and Scott, M. L. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP '10: Proc. 15th ACM Symp. on Principles and Practice of Parallel Programming*. 2010.

Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-591-5. doi:10.1145/1294261.1294281.

Dice, D., Shalev, O., and Shavit, N. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208. Stockholm, Sweden, 2006.

Dice, D. and Shavit, N. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33. 2007. doi:10.1109/CGO.2007.38.

Diegues, N. M. L. and Cachopo, J. Practical parallel nesting for software transactional memory. In *27th International Symposium on Distributed Computing*. Tel-Aviv University, Springer, Jerusalem, Israel, 2013.

Dongol, B. Formalising Progress Properties of Non-blocking Programs. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ICFEM'06, pages 284–303. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3-540-47460-9, 978-3-540-47460-9. doi:10.1007/11901433_16.

Dragojevic, A., Ni, Y., and Adl-Tabatabai, A.-R. Optimizing transactions for captured memory. In *Proceedings of the 21st annual Symposium on Parallelism in Algorithms and Architectures (SPAA'09)*, SPAA '09, pages 214–222. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-606-9. doi:10.1145/1583991.1584049.

Dworak, A., Sobczak, M., Ehm, F., Sliwinski, W., and Charrue, P. Middleware trends and market leaders 2011. Technical report, CERN, 2011.

Ennals, R. Efficient software transactional memory. Technical Report IRC-TR-05-051, Intel Research Cambridge Tech Report, 2005.

Ennals, R. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel, 2006.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005. doi:10.1145/1071610.1071615.

Fekete, A., O'Neil, E., and O'Neil, P. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14, 2004. doi:10.1145/1031570.1031573.

Felber, P., Fetzer, C., Guerraoui, R., and Harris, T. Transactions are back—but are they the same? *SIGACT News*, 39(1):48–58, 2008a. ISSN 0163-5700. doi:http://doi.acm.org/10.1145/1360443.1360456.

Felber, P., Fetzer, C., and Riegel, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)*, pages 237–246. ACM, New York, NY, USA, 2008b. ISBN 978-1-59593-795-7. doi:http://doi.acm.org/10.1145/1345206.1345241.

Fernandes, S. M. and Cachopo, J. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming (PPoPP '11)*, pages 179–188. ACM, San Antonio, TX, USA, 2011a.

Fernandes, S. M. and Cachopo, J. Strict Serializability is Harmless: a New Architecture for Enterprise Applications. In *Proceedings of the ACM international conference companion on Object Oriented Programming Systems Languages and Applications companion*, SPLASH '11, pages 257–276. ACM, New York, NY, USA, 2011b. ISBN 978-1-4503-0942-4. doi:10.1145/2048147.2048221.

Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Boston, MA, USA, 2002.

Fraser, K. *Practical lock-freedom*. Ph.D. thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

Fraser, K. and Harris, T. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007. doi:10.1145/1233307.1233309.

Gilbert, S. and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

Gray, J. The Transaction Concept: Virtues and Limitations. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

Guerraoui, R. and Kapalka, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 175–184. ACM, Salt Lake City, UT, USA, 2008. ISBN 978-1-59593-795-7. doi:http://doi.acm.org/10.1145/1345206.1345233.

Guerraoui, R., Kapalka, M., and Vitek, J. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*, pages 315–324. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-636-3. doi:http://doi.acm.org/10.1145/1272996.1273029.

Harris, T. and Fraser, K. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'03)*, pages 388–402. ACM Press, 2003. doi:10.1145/949305.949340.

Harris, T., Larus, J. R., and Rajwar, R. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2nd edition, 2010.

Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 48–60. ACM Press, 2005. doi:10.1145/1065944.1065952.

Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: double-ended queues as an example. *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003a. doi:10.1109/ICDCS.2003.1203503.

Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC'03)*, pages 92–101. ACM Press, 2003b. doi:10.1145/872035.872048.

Herlihy, M. and Moss, J. E. B. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual Snternational Symposium on Computer*

Architecture (ISCA'93), pages 289–300. ACM, New York, NY, USA, 1993. ISBN 0-8186-3810-9. doi:http://doi.acm.org/10.1145/165123.165164.

Herlihy, M. and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

Herlihy, M. and Shavit, N. On the nature of progress. In A. Fernàndez Anta, G. Lipari, and M. Roy, editors, *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25872-5. doi:10.1007/978-3-642-25873-2_22.

Herlihy, M. P. and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990. ISSN 0164-0925. doi:http://doi.acm.org/10.1145/78969.78972.

Instituto Superior Técnico. The FenixEdu Project: an Open-Source Academic Information Platform. `https://ciist.ist.utl.pt/projectos/Fenix.pdf`, 2011.

International Standards Organization. *SQL-92, ISO/IEC 9075:1992*, 1992.

Ireland, C., Bowers, D., Newton, M., and Waugh, K. A classification of object-relational impedance mismatch. In *Proceedings of the 1st International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA '09)*, pages 36–43. Cancun, Mexico, 2009.

Jiang, Y. *HBase Administration Cookbook*. PACKT Publishing, 2012.

Johns, M. *Gettting Started with Hazelcast*. PACKT Publishing, 2013.

Jorwekar, S., Fekete, A., Ramamritham, K., and Sudarshan, S. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1263–1274. VLDB Endowment, Vienna, Austria, 2007.

Korland, G., Shavit, N., and Felber, P. Noninvasive concurrency with Java STM. In *MultiProg 2010: Third Workshop on Programmability Issues for Multi-Core Computers*. 2010.

Lakshman, A. and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010. ISSN 0163-5980. doi:10.1145/1773912.1773922.

LSA-STM. LSA-STM project home page. `http://tmware.org/lsastm`, 2013. Last visited in August, 2013.

Manson, J., Pugh, W., and Adve, S. V. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05)*, pages 378–391. ACM, New York, NY, USA, 2005. ISBN 1-58113-830-X. doi:http://doi.acm.org/10.1145/1040305.1040336.

Marathe, V. J. and Moir, M. Toward High Performance Nonblocking Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 227–236. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-795-7. doi:http://doi.acm.org/10.1145/1345206.1345240.

Marathe, V. J., Scherer, W. N., and Scott, M. L. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems*, pages 1–7. ACM Press, 2004. doi:10.1145/1066650.1066660.

Marchioni, F. and Surtani, M. *Infinispan Data Grid Platform*. PACKT Publishing, 2012.

Olson, M. A., Bostic, K., and Seltzer, M. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43. USENIX Association, Berkeley, CA, USA, 1999.

Papadimitriou, C. H. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979. doi:10.1145/322154.322158.

Peluso, S., Ruivo, P., Romano, P., Quaglia, F., and Rodrigues, L. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465. IEEE, 2012.

Perelman, D., Fan, R., and Keidar, I. On maintaining multiple versions in STM. In *PODC '10: Proceedings of the 29th ACM symposium on Principles of Distributed Computing*. 2010.

Perez-Sorrosal, F., Patiño-Martinez, M., Jimenez-Peris, R., and Bettina, K. Consistent and scalable cache replication for multi-tier J2EE applications. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware (Middleware '07)*, pages 328–347. Springer-Verlag, Newport Beach, CA, USA, 2007.

Pritchett, D. BASE: An acid alternative. *Queue*, 6(3):48–55, 2008. doi:10.1145/1394127.1394128.

Riegel, T., Felber, P., and Fetzer, C. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006. ISBN 3-540-44624-9.

Rito, H. and Cachopo, J. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10)*, pages 89–98. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0269-2. doi:10.1145/1852761.1852775.

Romano, P., Carvalho, N., Couceiro, M., Rodrigues, L., and Cachopo, J. Towards the integration of distributed transactional memories in application servers' clusters. In *The Third International Workshop on Advanced Architectures and Algorithms for Internet DElivery and Applications (AAA-IDEA '09)*. ICST, Springer, Las Palmas, Gran Canaria, 2009.

Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM Press, 2006.

Seovic, A., Falco, M., and Peralta, P. *Oracle Coherence 3.5*. PACKT Publishing, 2010.

Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995. doi:10.1145/224964.224987.

Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littlefield, K., Menestrina, D., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., and Apte, H. F1: A distributed sql database that scales. In *VLDB*. 2013.

Spear, M. F., Michael, M. M., and von Praun, C. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*, pages 275–284. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-973-9. doi:http://doi.acm.org/10.1145/1378533.1378583.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1150–1160. VLDB Endowment, Vienna, Austria, 2007.

Vogels, W. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009. doi: 10.1145/1435417.1435432.

Watson, I., Kirkham, C., and Lujan, M. A Study of a Transactional Parallel Routing Algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*, pages 388–398. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2944-5. doi:http://dx.doi.org/10.1109/PACT.2007.11.