INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Replicação Preditiva para Memória Transaccional por Software Distribuída

João Carlos Moreira Fernandes

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Doutor Pedro Manuel Moreira Vaz Antunes de Sousa
Orientador:	Professor Doutor Professor Luís Eduardo Teixeira Rodrigues
Vogais:	Doutor Hervé Miguel Cordeiro Paulino

Outubro 2011

Agradecimentos

Começo por agradecer ao meu orientador, o Professor Luís Rodrigues, pela oportunidade, confiança, força, vontade e gosto em ensinar.

Agradeço também a todos os membros do Grupo de Sistemas Distribuídos que muito me ajudaram na elaboração deste trabalho, em especial ao Paolo Romano por toda a atenção que me despendeu.

Passando para a família, quero agradecer aos meus pais, Clarisse e Carlos, pelo amor incondicional; à minha irmã, Carla, por ser tão chata, bem como ao meu cunhado, Nuno, e à minha sobrinha, Carlota; aos meus padrinhos de baptismo, Mavildia e João por serem segundos pais; aos meus tios e primos, cujos nomes me fariam, felizmente, necessitar de mais páginas para agradecer todo o amor que me dão; ao meu padrinho de crisma, Gonçalo, pela constante presença e amizade; aos pais da minha namorada, Rosa e Fernando, que são já também um bocadinho meus pais. E claro, obrigado Catarina por todo o amor, apoio e carinho.

E não posso esquecer os amigos. Obrigado *Wokini*: Rita, Ana, Vanessa, Irina, Luís, Ricardo, e "respectivos". Obrigado também Joana, Sónia, Pedro Ruivo, Pedro Ribeiro, Xavier, Ivo, Nazar, João e Ivan por todos os momentos em que me deram forças para continuar. Obrigado também aos meus catequizandos, pela vossa amizade e confiança.

E como os últimos são sempre os primeiros, obrigado Pai pelo dom da vida.

Este trabalho foi parcialmente suportado pela FCT através do projecto "ARISTOS" (PTDC/EIA-EIA/102496/2008), pelo financiamento multianual do INESC-ID com fundos do programa PIDDAC e pela União Europeia, através do programa "Cloud-TM" (257784).

Lisboa, Outubro de 2011

João Carlos Moreira Fernandes

Ouvistes que foi dito: Olho por olho, e dente por dente. Eu, porém, vos digo que não resistais ao mal; mas, se qualquer te bater na face direita, oferece-lhe também a outra; E, ao que quiser pleitear contigo, e tirar-te a túnica, larga-lhe também a capa; E, se qualquer te obrigar a caminhar uma milha, vai com ele duas. Dá a quem te pedir, e não te desvies daquele que quiser que lhe emprestes. Ouvistes que foi dito: Amarás o teu próximo, e odiarás o teu inimigo. Eu, porém, vos digo: Amai a vossos inimigos, bendizei os que vos maldizem, fazei bem aos que vos odeiam, e orai pelos que vos maltratam e vos perseguem; para que sejais filhos do vosso Pai que está nos céus; Porque faz que o seu sol se levante sobre maus e bons, e a chuva desça sobre justos e injustos. Pois, se amardes os que vos amam, que galardão tereis? Não fazem os publicanos também o mesmo? E, se saudardes unicamente os vossos irmãos, que fazeis de mais? Não fazem os publicanos também assim? Sede vós pois perfeitos, como é perfeito o vosso Pai que está nos céus.

– Mateus 5, 38-48

Aos meus pais, irmã, sobrinha e namorada,
Clarisse, Carlos, Carla, Carlota e Catarina.

Resumo

Este trabalho descreve e avalia o SPECULA, um sistema de memória transaccional distribuída e replicada, baseado num esquema de certificação. O objectivo do sistema é mitigar os efeitos negativos da latência da rede, através da execução optimista de código. As transacções são executadas num único nó, de forma não coordenada, sendo utilizado o resultado da validação local das mesmas como predição do resultado da validação final. Os resultados das transacções (i.e., as modificações ao estado transaccional) são tornado visíveis de forma optimista e novas transacções são iniciadas com base em estado preditivo. Este processo repete-se de forma encadeada, podendo resultar numa sequência de transacções preditivas. Caso uma predição se confirme, o resultado da transacção é tornado definitivo; caso contrário, a sequência preditiva é cancelada e o estado do sistema aquando da predição é reposto. Este comportamento optimista é totalmente transparente para a aplicação e para o programador. O sistema foi avaliado com recurso a bancadas de teste padrão, nomeadamente a *Bank Benchmark* e a *STMBench7*, as quais exercitam cenários bastante variados.

Abstract

This work describes and evaluates SPECULA, a distributed and replicated software transactional memory system based on a certification scheme. This system tackles the negative effects of network latency, through the optimistic execution of code. Transactions are executed on a single node, in an uncoordinated fashion, and the result of their local validation is used as a prediction of the result of the final validation. The results of speculatively executed transactions (i.e., the modifications to the transactional state) are made visible to future transactions in an optimistic fashion. This speculative process repeats itself, creating a chain of speculatively executed transactions. If the final validation of a speculatively committed transaction allows it to commit system-wide, its result is made definitive; otherwise, a cascading abort takes place and the system restarts its execution in the state that preceded the mis-speculation. This speculative behavior is fully transparent for both the application and the programmer. The proposed design was evaluated using well known benchmarks that exercise very different scenarios, namely, Bank Benchmark and STMBench7.

Palavras Chave
Keywords

Palavras Chave

Memória Transaccional em Software Distribuída

Controlo de Concorrência Multi-versão

Replicação

Execução Preditiva

Keywords

Distributed Software Transactional Memory

Multiversion Concurrency Control

Replication

Speculative Execution

Table of Contents

List of Figures

List of Tables

Acronyms

AB Atomic Broadcast

DBMS Database Management System

DRSTM Distributed and Replicated Software Transactional Memory

DSTM Distributed Software Transactional Memory

GCS Group Communication Service

GSO Global Serialization Order

JVM Java Virtual Machine

MVCC Multiversion Concurrency Control

STM Software Transactional Memory

1 Introduction

It's mental static! I told you, if you HAVE to think, think in German!

– Asuka Langley Soryu, Neon Genesis Evangelion (TV Series)

The mainstream adoption of multiprocessor systems has brought parallel programming to the center of the main stage. For a very long time, application developers relied on the *free lunch* offered by the increase in single processor performance, and were able to avoid the *pains* of parallel programming. That time has come to an end.

Software Transactional Memory (STM) is an abstraction that eases the life of programmers dealing with concurrent data access. When using a STM, the programmer is freed from having to explicitly deal with concurrency control mechanisms. Instead, he only has to identify the sequences of operations that need to access shared data in an atomic fashion (i.e., transactions). Given that the use of low-level concurrency control mechanisms such as locks and semaphores is known to be extremely error prone (Cachopo 2007), the use of STM has the potential to increase the code's reliability, and to shorten the software development cycles.

Although STM was first proposed for cache-coherent shared memory architectures, the need to increase the scalability and fault-tolerance of STM-based systems has motivated the development of distributed and replicated STM implementations.

This work addresses the problem of building efficient Distributed Software Transactional Memory (DSTM) implementations that are fully replicated, with emphasis on mechanisms that explore the notion of possibly useful (speculative) computation to tackle the negative impact of network latency on the throughput of these systems.

1.1 Motivation

One of the common building blocks of Distributed and Replicated Software Transactional Memory (DRSTM) implementations is certification-based replication (Pedone, Guerraoui, & Schiper 2003), a coordination scheme originally designed for multi-master database environments. Implementations of this technique typically rely on a totally-ordered messaging protocol to build a Global Serialization Order (GSO) by which transactions are committed. These messaging protocols solve the problem of serializing conflicting transactions, and enable nodes to exchange control data with some delivery guarantees. Certification-based replication is an optimistic technique, as it allows transactions to be executed locally, without any kind of inter-node coordination.

The time needed to commit a transaction in a DRSTM is very long when compared with the average execution time of a memory transaction (Palmieri, Quaglia, Romano, & Carvalho 2010). Memory transactions are, therefore, costly to replicate. This problem is not so significant on DBMSs because database transactions are typically much longer than memory transactions, as they frequently require access to high-latency I/O[1] devices like hard-drives, and are subject to pre-execution stages such as parsing and query optimization.

Normally, when a thread is committing a transaction in a DRSTM, it stays blocked during the whole coordination phase. Moreover, concurrent transactions keep reading possibly stalled data. Literature around this problem tries to tackle it with two fundamentally different approaches: i) reducing the overhead of the coordination phase (Couceiro, Romano, Carvalho, & Rodrigues 2009), or ii) overlapping the coordination phase with possibly useful computation (Palmieri, Quaglia, & Romano 2010; Carvalho, Romano, & Rodrigues 2011). The former approach will always be limited by the network latency, while the latter can incur in long cascading abort procedures.

This work proposes a solution that follows the approach of overlapping the coordination phase with speculative computation, in order to minimize the negative impact of network latency on DRSTMs.

[1] Input/Output

1.2 Contributions

This work addresses the problem of optimizing the performance of DRSTMs. More precisely, the thesis analyzes, implements and evaluates techniques to maintain the consistency of replicated transactional data. As a result, the thesis makes the following contribution:

- a novel system named SPECULA, that enables the safe speculative execution of both transactional and non-transactional computation. SPECULA extends its operation to the underlying STM, by making transactions commit in a optimistic fashion. This allows threads to continue executing code while slow inter-node coordination takes place in background. Since more code is executed in the same time interval, the system's performance increases;
- the identification of workload scenarios that favor the use of SPECULA.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- a prototype implementation of SPECULA for the Java Virtual Machine (JVM), that integrates all the components required to enable the safe speculative execution of code, which includes i) instrumenting applications at the Java bytecode level, ii) adding speculative extensions to the underlying STM, and iii) extending certification-based replication in a way that ensures the system's correctness in the presence of speculative state;
- An experimental evaluation of the prototype implementation based on synthetic benchmarks.

1.4 Research History

This work was performed in the context of the ARISTOS[2] project (Autonomic ReplicatIon of Software Transactional memorieS).

[2]http://aristos.gsd.inesc-id.pt

During my work, I benefited from the fruitful collaboration with the remaining members of the GSD team working on ARISTOS, namely Paolo Romano, Luís Rodrigues and Pedro Ruivo. I would also like to acknowledge the productive talks I had with members of the ESW group, namely Ivo Anjo.

Parts of the work reported in this dissertation have been published in (Fernandes, Carvalho, Romano, & Rodrigues 2011).

1.5 Outline

The remainder of this dissertation is organized as follows:

- *Related Work.* Chapter 2 presents a review of relevant work related to this dissertation.
- *SPECULA.* Chapter 3 introduces SPECULA, the solution to the problem that we propose to solve. It starts by providing an in-depth motivation for this work and sums up the pros and cons of existent DRSTMs, which is followed by an overview of the solution we propose; it then moves on to the assumptions that were made about the execution environment, and the properties that the system guarantees; it then describes, in detail, how the system works; finally, it presents specific implementation details of our prototype, showing how problems were solved and limitations of the executing environment were overcome.
- *Evaluation.* Chapter 4 presents the results of the experimental evaluation study.
- *Conclusions and Future Work.* Finally, Chapter 5 summarizes the work described in this dissertation, the results achieved, and what are its main contributions, ending with the proposal of possible directions to further improve the SPECULA system.

2 Related Work

*Uhhh... it's not *our* fault...*

– Ritsuko Akagi, Neon Genesis Evangelion (TV Series)

Fully understanding the problem addressed in this thesis requires knowing the fundamentals of various areas in distributed systems. The following sections introduce important research work related to this dissertation.

The chapter is organized as follows. Section 2.1 presents an overview of group communication systems and primitives. Section 2.2 presents fundamental concepts regarding transactional systems. Section 2.3 describes replication techniques and indentifies their strenghts and weakenesses. Section 2.4 contains an insighful discription of software transactional memory systems, including design choices and desired properties. Finally, Section 2.5 presents a set of selected distributed software transactional memories.

2.1 Group Communication

Group Communication (Chockler, Keidar, & Vitenberg 2001; Powell 1996) is a powerful paradigm for performing multipoint-to-multipoint communication by organizing processes in groups. A system that implements this paradigm is commonly called a Group Communication Service (GCS), and offers membership and reliable broadcast services with different message delivery guarantees. GCSs allows programmers to concentrate on what to communicate rather than on how to communicate. Our focus will be on view-synchronous services (Birman & Joseph 1987).

View-synchronous GCSs model the system as a group of n processes $\prod = \{p_1, ..., p_n\}$. The group-membership service exports both the `join(`S`)` and `leave(`S`)` primitives (where S is set of processes such that $S \subset P$). It outputs a sequence of group membership sets called

views. Every view $V \subseteq P$ is delivered through the `v-change`(vid, V) primitive, where $vid \in \mathbb{N}$ denotes a monotonically increasing view identifier. When this occurs, we say that the process installs the new view V.

Apart from handling explicit `join` and `leave` requests, a membership service also plays the role of a failure detector: it detects and excludes crashed processes from the group membership, leaving just the *stable* components of the system, i.e., the set of processes that are correct and that can reliably communicate with each-other. A stable component is defined as a set of processes that is eventually permanently connected.

Reliable broadcast services can provide various types of ordering disciplines, such as FIFO[1], causal and total order. FIFO ordering guarantees that messages sent by a node are delivered by the order they were sent at every process that receives them. Causal ordering extends FIFO, and guarantees that if message m' is causally related to message m (according to Lamport's *happened-before* relation (Lamport 1978)), then m is delivered before m' on every process that receives both of them. Total ordering guarantees that all messages are delivered by the same order, in every receiver.

2.1.1 Atomic Broadcast

Atomic Broadcast (AB) is a reliable broadcast messaging protocol that ensures that all delivered messages are totally-ordered. It exports the two following primitives:

- `AB-broadcast`(m) broadcasts message m to all the nodes in the current view.
- `AB-deliver`(m) delivers message m to the application, according to the GSO.

There are two types of AB: regular and uniform. Uniform AB ensures the following properties:

- *Validity:* if a correct process AB-broadcasts message m, then it eventually AB-delivers m.
- *Uniform Agreement:* if a process AB-delivers message m, then all correct processes eventually AB-deliver m.

[1] First-In, First-Out

- *Uniform Integrity:* for any message m, every process AB-delivers m at most once, and only if m was previously AB-broadcast by its sender.
- *Uniform Total Order:* if processes p and q both AB-deliver messages m and m', then p AB-delivers m before m' only if q AB-delivers m before m'.

Validity and agreement properties are liveness properties, i.e., they ensure that something *good* eventually happens, while integrity and total order properties are safety properties, i.e., they ensure that nothing *bad* happens.

Regular AB has the same validity and integrity properties as Uniform AB, but has the following non-uniform properties:

- *(Regular) Agreement:* if a correct process AB-delivers a message m, then all correct processes eventually AB-deliver m.
- *(Regular) Total Order:* if two correct processes p and q both AB-deliver messages m and m', then p AB-delivers m before m' if and only if q AB-delivers m before m'.

Uniform properties make life easier for application developers, as they apply to both correct and faulty processes. However, enforcing uniformity has often an associated cost. Therefore, it is important to consider if uniformity is strictly necessary. Non-uniform properties can lead to inconsistencies at the application level if they are not considered properly, and so applications subject to them should be prepared to take the necessary corrective actions during failure recovery.

2.1.2 Optimistic Atomic Broadcast

Optimistic Atomic Broadcast (OAB) messaging protocol exports all the primitives of AB plus one:

- `OAB-deliver(`m`)` delivers message m without any ordering guarantees.

OAB is a very important building block for high-performance distributed systems. The `OAB-deliver` primitive enables applications to overlap computation with communication: messages are OAB-delivered as soon as they arrive from the network and, when their final order is

known, they are AB-delivered. Between these two events, applications can perform optimistic, possibly useful computation. Moreover, on LAN environments the optimistic delivery order typically matches the GSO, a property that is known as spontaneous total order (Kemme, Pedone, Alonso, Schiper, & Wiesmann 2003) .

OAB features all the properties of AB plus one:

- *Optimistic Order:* if a process p AB-delivers message m, then p has previously OAB-delivered m.

2.2 Concepts of Transactional Systems

DBMSs are a key component in many information systems. Like many other applications, DBMSs feature replication to provide increased fault-tolerance and scalability. It is necessary to introduce a number of concepts related with transactional systems, such as DMBSs and STMs, before we proceed any further.

- *Read-set:* the set of data items from which a transaction has read.
- *Write-set:* the set of data items to which a transaction has written.
- *Data-set:* the union of the read-set and write-set of a transaction.
- *Commit:* the transaction terminated successfully, therefore its write-set was made visible.
- *Abort:* the transaction did not commit; its write-set is not made visible or its changes are undone.
- *Conflict:* two concurrent transactions conflict if their data-sets overlap.

During its life, a transaction is always in one of the well-defined states of the state machine depicted in Figure 2.1: *executing*, *committing*, *committed* and *aborted*. Both the *executing* and the *committing* states are transitory, while the *aborted* and *committed* states are final. Their description is the following:

- *Executing:* the transaction is performing operations.

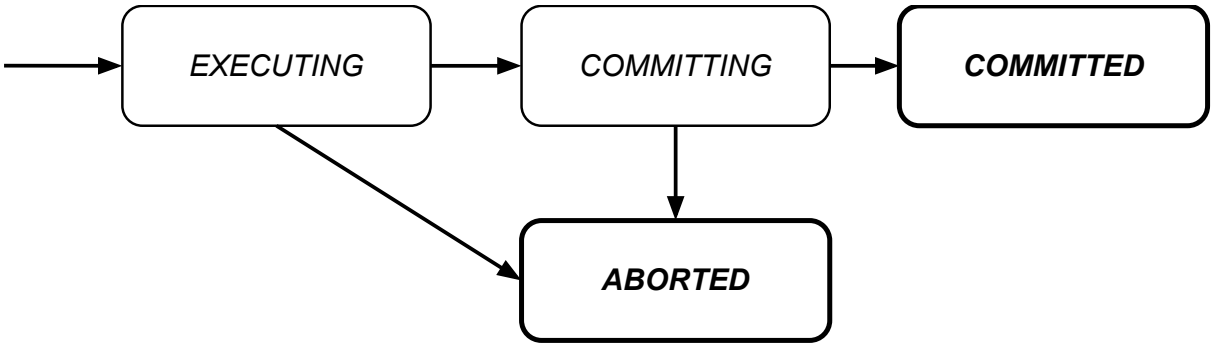


Figure 2.1: The life of a transaction.

- *Committing:* the client has requested the commit of the transaction, and the the global validation procedure is taking place.
- *Committed:* the transaction was committed.
- *Aborted:* the transaction was aborted.

A transaction in either the *executing* or *committing* state is said to be active.

Regarding properties, database transactions satisfy atomicity, isolation, consistency and durability. These are commonly referred to as ACID properties (Härder & Reuter 1983). Their description is the following:

- *Atomicity* ensures that modifications must follow an "all or nothing" rule, i.e., either all the modifications made by a committed transaction are made visible or none is.
- *Consistency* ensures that each transaction changes the database from one consistent state to another consistent state.
- *Isolation* ensures that individual memory updates within a memory transaction are hidden from concurrent transactions.
- *Durability* ensures that once a transaction is committed, its updates will survive any subsequent malfunctions.

Memory transactions satisfy only atomicity, isolation and consistency properties, being the consistency property different in databases and STMs. In STM, consistency generally encompasses the notion of serializability. Sometimes its meaning is extended to include the preserva-

tion of integrity constraints, such as the ones captured by the property of opacity (Guerraoui & Kapalka 2008b), which is discussed in more detail ahead, in Section 2.4.3 of this chapter.

It is also possible to argue that memory transactions satisfy the durability property, if we do not make a strict association between durability and persistent/stable storage.

2.3 Replication Techniques

2.3.1 Foundations

Replication enables the construction of highly available, fault-tolerant systems. There are two main families of replication techniques: passive and active.

Passive replication, also known as master-slave replication or primary-backup, is characterized by the existence of a replica, known as the master, that processes all updates requests and transfers state updates to the remaining replicas, which are known as the slaves or backups. In most cases, slaves can process read-only transactions in a totally uncoordinated fashion. Passive replication often assumes the fail-stop model. When the master fails, one of the slaves is elected to replace it, becoming the new master. One of the limitations of this technique is that the master can quickly become the system's bottleneck, since it is the only one that can process update requests. However, most real-world applications are subject to read-dominated workloads, which are balanceable with this technique.

Active replication is characterized by having all replicas processing the same sequence of requests, by the exact same order. To ensure the consistency of the replicated data, active replication requires all operations to be deterministic, otherwise the state of each replica could diverge. Requests are processed according to the GSO, which is normally defined by the AB protocol used to disseminate requests. One of the problems of this technique is that AB is a relatively slow communication primitive, as it requires that consensus is reached among all nodes. Moreover, since all replicas have to execute all update requests, the ability to process them does not increase. On the contrary, it does in fact decrease, as the cost of group communication increases with the number of peers. However, as happens with passive replication, read-only requests can be processed in parallel at different replicas.

Other replication schemes combine aspects of the two previous techniques. An important

example is certification-based replication, a scheme that is commonly employed on transactional systems. It is described in Section 2.3.2.

2.3.2 Certification-based Replication

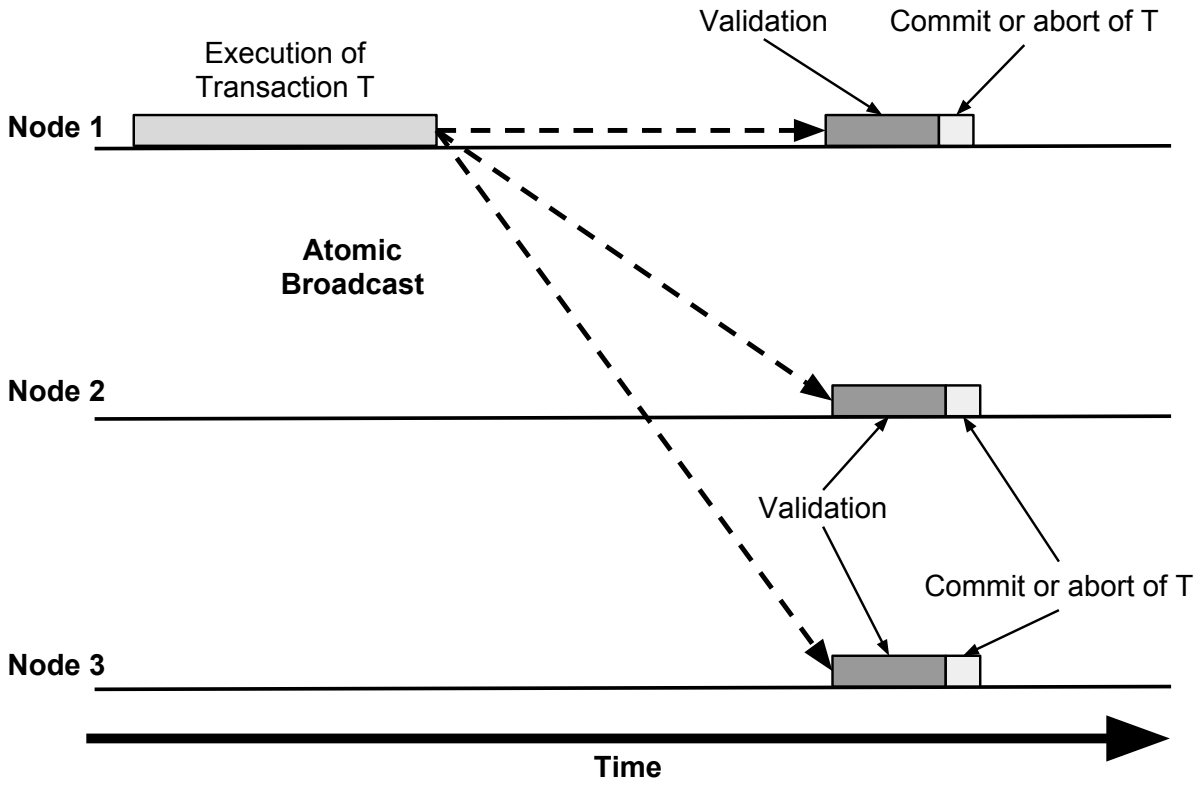


Figure 2.2: Non-voting certification-based replication.

Certification-based replication was first introduced in (Pedone, Guerraoui, & Schiper 2003) by Pedone et al., as a scheme designed to synchronize a cluster of database servers in a multi-master environment. It is based on the state machine approach (Schneider 1990).

In certification-based replication, a single node executes a transaction and, upon its commit request, its data-set is AB-broadcast to all replicas, commencing the certification phase. The certification of a transaction consists in checking for write–read data access conflicts, in order to guarantee that its commit does not violate the one-copy serializability (Bernstein, Hadzilacos, & Goodman 1987) correctness property. A transaction is aborted if its commit would lead the database into an inconsistent state (i.e., one that is non-serializable). This step is executed by all replicas, in a deterministic fashion. Certification-based replication reduces the cost of

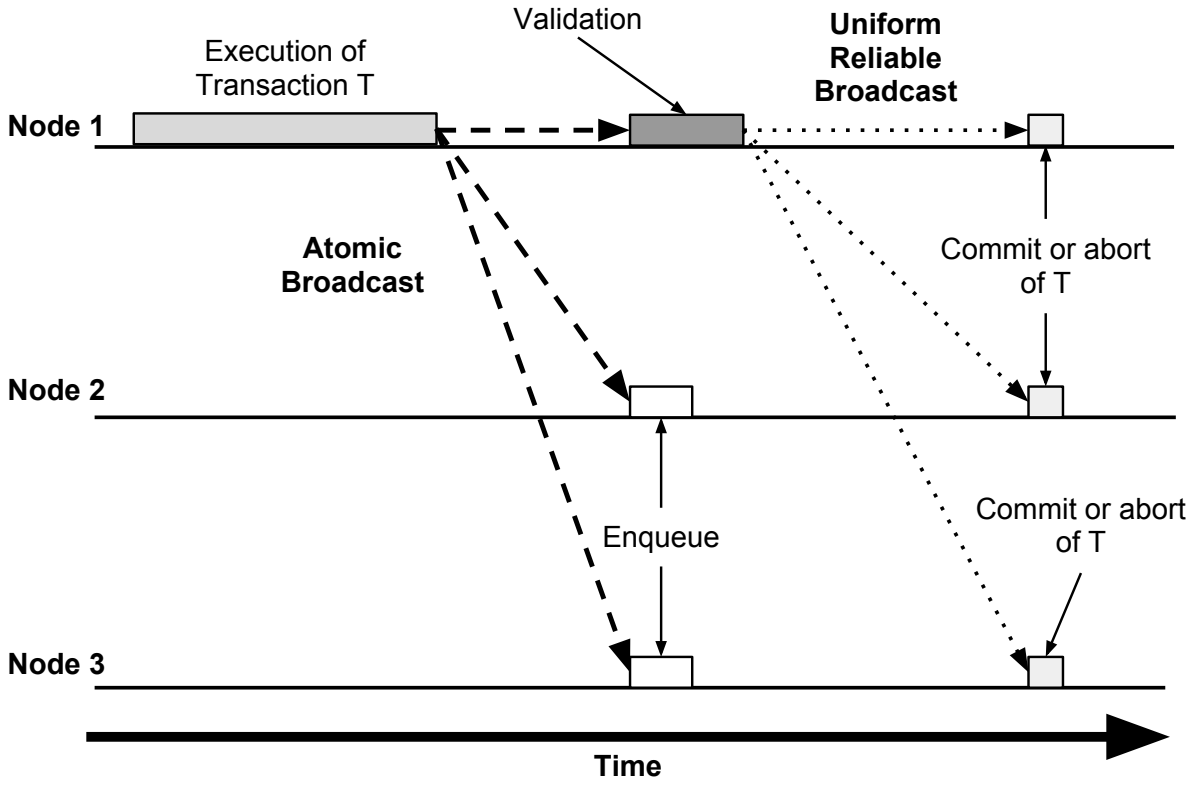


Figure 2.3: Voting certification-based replication.

inter-node coordination, by imposing a single cluster-wide interaction per commit request. It also simplifies recovery, as the intermediate writes of a transaction are never propagated, which means that there is no need to implement a distributed rollback scheme.

The above described scheme is also known as non-voting certification-based replication, and is depicted in Figure 2.2. A small variation of it only AB-broadcasts the write-set of transactions. Upon receiving a write-set, only the node that executed the transaction validates it, as it is the only one that knows the transaction's read-set. That node then broadcasts the validation result, informing all remaining nodes if they should either commit or abort the transaction. This scheme is known as voting certification-based replication, and is depicted in Figure 2.3.

2.4 Software Transactional Memory

Concurrent applications make use of multiple threads of control to execute multiple sequences of operations in parallel. Concurrent access to shared data by these threads can cause inconsistencies that may lead the applications into incorrect states. For this reason, access to

shared data has to be subject to some sort of concurrency control. Classic concurrency control mechanisms, such as locks, can ensure mutually exclusive access on data. Unfortunately, locks are hard to use in a correct manner, as a single misplaced or missing lock can easily create a severe problem in the application. Indeed, locks comprise so much complexity that, even when placed in a way that ensures mutual exclusion, they can still lead to undesired effects, such as deadlock, priority inversion, poor performance in face of preemption and page faults, lock convoying or incorrect behaviors (e.g., infinite loops). In order to avoid these problems, the programmer needs to have a deep knowledge of low-level properties of the execution environment, and a complete insight as to how concurrent execution flows in the application.

STM support the transaction abstraction as an high-level concurrency control mechanism. Transactions release the programmer from explicitly dealing with the low-level details of concurrency control. When using an STM, the programmer only has to identify the sequences of operations that need to access shared data in an atomic fashion. STM allows programmers to express what should be executed atomically, rather than requiring them to specify how to achieve such atomicity. This translates into what is argued to be the main advantage of STM: composability (Harris, Marlow, Jones, & Herlihy 2005). Unlike locks, STM enables software composition, i.e., correctly implemented concurrent abstractions can be composed together to form larger abstractions.

2.4.1 Types of Transactional Memory

Currently, there are several proposals for hardware transactional memory(HTM) (Ananian, Asanovic, Kuszmaul, Leiserson, & Lie 2005; Hammond, Wong, Chen, Carlstrom, Davis, Hertzberg, Prabhu, Wijaya, Kozyrakis, & Olukotun 2004; Herlihy & Moss 1993; Tomic, Perfumo, Kulkarni, Armejach, Cristal, Unsal, Harris, & Valero 2009; Bobba, Goyal, Hill, Swift, & Wood 2008), purely software based implementations, i.e. software transactional memory (Shavit & Touitou 1995; Saha, Adl-Tabatabai, Hudson, Minh, & Hertzberg 2006; Cachopo & Silva 2006; Harris & Fraser 2003; Dice, Shalev, & Shavit 2006), and hybrids that combine both hardware and software components (Tabba, Moir, Goodman, Hay, & Wang 2009; Damron, Fedorova, Lev, Luchangco, Moir, & Nussbaum 2006; Kumar, Chu, Hughes, Kundu, & Nguyen 2006).

Regardless of the layer where the implementation is made, most transactional memory implementations support both unbounded and dynamic transactions (Dice & Shavit 2007).

Supporting unbounded transactions (as opposed to bounded transactions (Dice & Shavit 2007)) means that there is no limit to how many items a transaction can read or modify. Supporting dynamic transactions (as opposed to static transactions (Dice & Shavit 2007)) means that there is no need to know the transaction's data-set *a-priori*, since it is determined at runtime.

Some HTMs, such as the one proposed by Herlihy et al. in (Herlihy & Moss 1993), only support bounded, static transactions. This forces the programmer to be aware of the limitations of the transactional engine, and thus to write code in a way that circumvents them, a fact that contradicts the transactional memory abstraction goal of simplifying the programming of concurrent applications. Providing large scale transactions in hardware tends to introduce large degrees of complexity into the design. Because of this, unbounded and dynamic HTMs, like the one proposed by Ananian et al. in (Ananian, Asanovic, Kuszmaul, Leiserson, & Lie 2005), are unlikely to be adopted by mainstream commercial processors in the near future.

STMs have fewer limitations when compared to HTMs. Since STMs make no exotic hardware support assumptions, they can be implemented on commodity hardware, a factor that increases their usability. Furthermore, software is more flexible and easier to evolve than hardware. However, one of the most troublesome drawbacks of STMs is performance. Although the performance of STMs has improved over the years, they are still significantly slower than traditional lock-based and HTM solutions. The design and implementation of STMs is subject to many choices, and each one carries advantages and shortcomings (Ennals 2006; Dice & Shavit 2007; Marathe, Iii, & Scott 2004; Marathe, III, & Scott 2005; Cachopo & Silva 2006).

Hybrid solutions aim at combining the best of both approaches. The key idea is to use hardware support to boost performance and fallback to software when facing hardware limitations (or when hardware support is simply not available). As shown by Dice et al. in (Dice & Shavit 2007), hardware support for read-set validation, as opposed to full blown HTM, may deliver significant performance benefits.

2.4.2 Design Choices and Classification

The design of STMs involves choices. These choices have a significant impact on their properties and behavior. STMs are commonly classified using the following properties:

- *Granularity.* The view of the transactional memory and the unit of conflict detection. An

STM can be *word-based*, *page-based*, *field-based* or other. The smaller the unit, the smaller the probability of a conflict due to false sharing, i.e., conflicts caused by concurrent writes to different parts of the same unit. However, smaller units require the maintenance of more metadata, resulting in a higher memory overhead.

- *Atomicity semantics.*[2] Can either be *strong* or *weak*. An STM that features strong atomicity offers the guarantee that all access to transactional state is made inside a transactional context. STMs featuring weak atomicity do not offer this guarantee.

- *Update strategy.* Can either be *direct* or *deferred*. STMs that use a direct strategy perform writes in-place, and, therefore, have to revert modifications on rollback, while STMs that use a deferred strategy, i.e., perform writes in an private write-set, have to write-back values to memory at commit time.

- *Concurrency control.* Can either be *optimistic* or *pessimistic*. An STM that features a pessimistic concurrency control scheme detects and resolves conflicts at the time they occur. STMs that features an optimistic concurrency control scheme postpone conflict detection and resolution, typically to commit time.

- *Progress.* There are three non-blocking guarantees of progress: wait-freedom, lock-freedom and obstruction-freedom. Wait-freedom is the strongest of the three, and states that all threads contending over a set of items make forward progress in a number of finite steps, i.e., forward progress is guaranteed. Lock-freedom is a weaker guarantee than wait-freedom, and ensures that a system as a whole moves forward regardless of anything, but forward progress for each individual thread is not guaranteed, i.e., individual threads can starve. Obstruction-freedom is the weakest of three guarantees, and ensures that a thread makes forward progress only if it does not encounter data contention, i.e, two threads can prevent each other's progress and lead to a livelock. All three non-blocking progress guarantees provide a guarantee of termination-safety, i.e., a terminated thread does not prevent system-wide forward progress. Lock-based synchronization can also ensure forward progress. In (Guerraoui & Kapalka 2009), Guerraoui et al. propose strong progressiveness as the *de facto* progress property for lock-based STMs. Strong progressiveness ensures that non-conflicting transactions are guaranteed to commit, and that at least one transaction

[2] Also commonly referred to as isolation semantics.

among conflicting transactions is guaranteed to commit.

- *Conflict detection.* Can either be *eager* or *lazy*. An eager conflict detection scheme detects a conflict when a transaction declares its intent of accessing an item. Lazy conflict detection happens typically just once, before the commit of a transaction. Eager conflict detection may abort transaction that could otherwise commit, while late conflict detection discards more computation. There are many ways to perform conflict detection, being the two primary ones validation and invalidation. Validation checks if the read-set of an active transaction overlaps with the write-sets of concurrent but already committed transactions. Validation should happen either by value or by version number (the latter avoids the *ABA* problem). In case the validation procedure detects a conflict (also referred to as failed validation) then the committing transaction has to be aborted, since the transaction with which it overlaps is already committed. Invalidation checks if the write-set of an active transaction overlaps with the read-sets of other active transactions. Since all transactions involved are active, the STM can choose what transactions it should abort. Another relevant property to conflict detection is disjoint access parallelism (Guerraoui & Kapalka 2008a), which states that operations on disconnected data should not interfere, i.e., the data-sets of two concurrent transactions tx_a and tx_b intersect if they both access to the same base object and at least one of the transactions has wrote in it.

- *Conflict resolution.* Resolving conflicts between concurrent transactions requires the abort of one or more transactions. A contention manager typically implements one or several contention policies in order to decide which transaction(s) to abort. An example of a contention policy can be to always abort the newest transaction(s).

- *Nesting.* There are three types of nesting: closed, open and flattened. The definitions of closed and open nesting are very complex. In practice, the abort of a closed nested transaction does not cause any side-effect to its parent, whereas the abort of a flattened nested transaction forces its parent to abort. An open nested transaction operates at a different abstraction level from its parent. The commit of an open nested transaction makes its write-set immediately public, which does not happen with closed or flattened nested transactions.

2.4.3 Opacity

In (Guerraoui & Kapalka 2008b), Guerraoui et al. state that properties such as linearizability (Herlihy & Wing 1990) and serializability (Papadimitriou 1979) are not sufficient for transactional memories to ensure program correctness, while opacity (Guerraoui & Kapalka 2008b) is. Opacity can be viewed as an extension of the classical database serializability property, with the additional requisite that even non-committed transactions are prevented from accessing inconsistent states (most DBMSs only guarantee that committed transactions did not see inconsistent states). It is an important correctness property because memory transactions may be coded in a wide range of programming languages and might not be executed on a sandboxed environment. The lack of this property can lead to exceptions or incorrect behaviors on otherwise correct code (e.g., infinite loops).

Listing 2.1: An opacity violation example.

```
1  // Invariant: (x + y >= 0) && (x + y < ARRAY_SIZE)
2  // Initially: x == y == 0
3
4  // Thread 1
5  void doA() {
6    // tx_a
7    Transaction.begin();
8    a = x; // a = 0
9                                                    // Thread 2
10                                                   void doB() {
11                                                     // tx_b
12                                                     Transaction.begin();
13                                                     x = 1;
14                                                     y = -1;
15                                                     Transaction.commit();
16                                                   }
17   b = y; // b = -1
18   arr[a + b] = 42; // a + b == -1 --> ERROR!
19   Transaction.commit();
20 }
```

Let us analyze the example depicted in Listing 2.1. We consider that there is a program invariant where $(x + y \geq 0 \wedge x + y < ARRAY\_SIZE)$. As shown, transaction tx_a performs the operation $arr[a + b] = 1$ where arr is an array of size $ARRAY\_SIZE$. If while tx_a is making local copies of x and y, a concurrent transaction tx_b changes the values of both x and y to respectively 1 and -1 and commits, then if later tx_a sees that changes, the assignment to arr either fails and emits an exception (if on a supervised environment) or corrupts the system's

memory. Notice that no damage is made to the shared state, as the invariant holds true. Notice also that if tx_b commits, then tx_a does not, as it conflicts with tx_b.

2.5 Distributed Software Transactional Memory

Much of the work around STM targets shared memory, cache-coherent architectures. The construction of efficient STMs for non-shared memory and non-cache-coherent architectures is a relatively new and unexplored topic.

In this section, we present a selection of relevant systems, including replicated and non-replicated systems.

2.5.1 Distributed Software Transactional Memory

The Distributed Software Transactional Memory (DiSTM) (Kotselidis, Ansari, Jarvis, Luján, Kirkham, & Watson 2008) is a system created to facilitate the prototyping of transactional memory coherence protocols. It is built around two main components: an extended version of the DSTM2 transactional engine (Herlihy, Luchangco, & Moir 2006), and a remote communication system based on the ProActive framework (Laurent Baduel, Francoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, & Romain Quilici 2006). The system relies on the existence of a central master node, which is responsible for coordinating the cluster. Each node maintains a cache that contains a subset of the shared transactional data. A distributed transactional memory coherence protocol is responsible for ensuring the strict consistency of each node's cache.

Being based on DSTM2, DiSTM executes all transactions speculatively. When a transaction updates an object, instead of directly modifying the actual object, a cloned version of the object is generated, used, and kept private until the transaction commits. The commit phase is preceded by a validation phase, where conflicts are detected and resolved. A transaction that passes the validation phase with success can be safely committed.

The key concept of the ProActive framework is the notion of active object. Each active object has its own execution context and can be distributed over the network, supporting both

mobility and remote method invocation. Nodes communicate among themselves through calls to active objects.

DiSTM does not exploit multiversioning. Therefore, it does not guarantee that read-only transactions are abort-free. Moreover, the lack of fault-tolerance guarantees represents additional complexity for programmers that develop applications which demand them.

All the three featured distributed transactional memory coherence protocols proposed face scalability issues. The Transactional Coherence and Consistency (TCC) protocol imposes two broadcasts along the critical path of the commit phase, which translates in long commit procedures. The serialization lease protocol minimizes the number of broadcasts needed by the TCC protocol, but forces transactions to acquire a unique system-wide lease from the master node, making it the bottleneck for acquiring and releasing the lease. Moreover, transactions stay blocked in the commit phase until they acquire the lease, and the master node may attempt to assign the lease to the commit request of a transaction that has aborted due to a conflict with the last committing transaction. The multiple leases protocol effectively reduces the time that a transaction takes to acquire a lease, but since every transaction has to be validated on the master node, it again becomes the bottleneck. Is also important to consider that in face of conflict intensive workloads, the benefits of the multiple leases scheme diminishes, because fewer transactions are allowed to commit concurrently.

2.5.2 Distributed Multiversioning

Distributed Multiversioning (DMV) (Manassiev, Mihailescu, & Amza 2006) is a page-level distributed concurrency control algorithm that exploits the presence, and easy maintenance, of different versions of the transactional data-set across nodes. Like local multiversioning schemes, DMV allows read-only transactions to execute in parallel with conflicting update transactions. This is achieved by ensuring that read-only transactions always access a consistent snapshot. However, each node maintains a single copy of each transactional item. Therefore, the system delays applying (local or remote) updates to the transactional items, in order to maximize the probability of not having to invalidate the snapshot of any active transaction and thus forcing them to abort. For this reason, updates are only applied when it is strictly necessary.

The authors propose two cluster configurations: i) a completely decentralized, update-

anywhere configuration, with no scheduler support, and ii) a centralized, master-update configuration with scheduler support.

This solution presents a good cost-benefit relation. Unlike classic local multiversioning, DMV does not impose the overhead of having to maintain multiple copies of the same item. However, DMV does not ensures that read-only transactions are abort-free, as their success is dependent on the timing of concurrent access to data by conflicting transactions. The pessimistic approach used in the update-anywhere protocol for the serialization enforcement (distributed mutual exclusion) can seriously hamper the system's performance, considering that only one node commits at any time, and that consequently, all other nodes can be blocked, waiting for the commit token. In the master-update scheme, the master node can become the performance bottleneck, as it is responsible for the execution of all update transactions.

2.5.3 Cluster-STM

Cluster-STM (Jr., Adve, & Chamberlain 2008) is a distributed (non-replicated) STM that was designed to achieve high performance on large-scale non-cache-coherent distributed systems, such as commodity clusters. Performance boosts come from an efficient distribution of both data and computation across the nodes of the cluster. An efficient distribution should require minimum inter-node communication, by taking advantage of fast intra-node communication. The system requires the assignment of a home node for each data item (a scalar or an array element), which maintains an authoritative version of the item and its metadata. Home nodes are also responsible for synchronizing the access of conflicting remote transactions to the items they shelter. The distribution of both data and computation is not transparently managed by Cluster-STM. It has to be managed independently, either by another system, or manually, by the programmer.

The system does not support the dynamic creation of new execution contexts, thus it is bootstrapped with a fixed number of tasks, and no task is created or destroyed during the life of the program. Tasks can request the execution of transactional code on other tasks, in order to avoid slow network data transfers, and through this benefit from data locality. Cluster-STM does not feature any replication scheme, nor does it provide a coherent cache of transactional remote items. Programmers in need of such features have to implement their own schemes, at the application-level.

2.5.4 Distributed Dependable Software Transactional Memory

The Distributed Dependable Software Transactional Memory (D^2STM) (Couceiro, Romano, Carvalho, & Rodrigues 2009) is built on top of the JVSTM transactional engine, and the Appia (Miranda, Pinto, & Rodrigues 2001) GCS, in which the replication manager relies to coordinate the cluster. All the nodes in the cluster maintain a full copy of the transactional data. D^2STM provides a conventional STM interface that transparently ensures non-blocking and strong correctness guarantees (i.e., one-copy serializability) even in the presence of failures. By using JVSTM, D^2STM inherits opacity and strong atomicity guarantees.

The novel replica synchronization scheme employed is called Bloom Filter Certification (BFC). It is a non-voting certification scheme that exploits a Bloom filter based encoding (Bloom 1970) of the transactions' read-set, in order to reduce the overhead of the coordination phase. A Bloom filter is a probabilistic data structure with a tunable size that is used to test whether an element is member of a set. Queries to a Bloom filter can result in false positives (i.e., a query for a certain item may indicate that it is a member of the set when in fact it is not), but never in false negatives. The more elements that are added to a set, the larger the probability of occurring false positives. Conversely, the greater the size of the Bloom filter, the lower the probability of occurring false positives for the same number of elements.

By providing full replication of the transactional set, D^2STM presents itself as a suitable platform for the development of systems with strong consistency and high availability requisites. Bloom filter based encoding effectively helps to reduce the amount of data exchanged among replicas. This is a determinant factor for the performance of the AB service and leads to a significant reduction of the overhead associated with the transaction certification phase. Providing abort and wait-free read-only transactions is a very important feature, since most realistic transaction-processing workloads are read-dominated.

2.5.5 Aggressively Optimistic

AGGRessively Optimistic (AGGRO) (Palmieri, Quaglia, & Romano 2010) is an active replication protocol that aims at maximizing the overlap between communication and computation, through an optimistic concurrency control scheme.

The key idea underlying AGGRO is the propagation of the write-sets of yet uncommitted

transactions to their following transactions, according to a serialization order that is compliant with the message delivery order defined by an OAB service, through which the nodes of the system exchange control data. This optimistic scheme takes advantage of the spontaneous total order property, allowing the processing of any transaction while its GSO is still unknown. When the optimistic serialization order and the GSO are not equivalent, the system triggers a (cascading) abort event for all the transactions that have directly or indirectly read from the write-set of an aborted transaction.

By being a pure active replication protocol, AGGRO creates a great amount of redundant computation, and by serializing transactions against an optimistic message delivery order, AGGRO's performance should decrease significantly when it is facing conflict-intensive workloads, in a network environment with a high percentage of out-of-order message deliveries. In this scenario, it is almost impossible to overlap communication with computation, so the overhead created by the necessary execution of cascading rollbacks does not pay-off.

2.5.6 Speculative Certification

Speculative Certification (SCert) (Carvalho, Romano, & Rodrigues 2011) is a certification-based replication protocol designed specifically for DRSTMs. It exploits optimistic deliveries made by an OAB service to propagate, in a speculative fashion, the write-sets of transactions, before their GSO is established. This scheme lowers the chances of accessing a stale snapshot, thus minimizing the abort rate of transactions. It also features early conflict detection, thus reducing the amount of computation and/or waiting time of transactions doomed to abort.

The optimistic nature of certification-based replication leads to very high abort rates when in the presence of conflict intensive workloads. SCert relies on the spontaneous total order property found in LAN environments. This property establishes an upper bound on the number of transactions that may have to abort due to mis-speculation. A transaction that reads from a snapshot created by the speculative commit of another transaction can itself commit speculatively, creating a chain of speculatively committed transactions. Therefore, in case of mis-speculation, the system has to trigger a (cascading) abort event for all the transactions that have directly or indirectly read from the write-set of an aborted transaction.

The downside of SCert is that, like in most speculative scheme, cascading aborts are ex-

pensive to perform. When mis-speculation occurs, the SCert algorithm inhibits the creation of new transactions while the transactional state is being patched. This can seriously hamper local concurrency when in massively multi-threaded environments.

Increases in the abort rate are typically followed by a significant performance degradation. The effectiveness of SCert clearly depends on the probability that the optimistic message delivery order matches the final order.

Another subtle problem of SCert is that it requires changes to already committed snapshots. This may lead to temporary inconsistencies, as these changes may not be immediately visible to concurrent threads. Programmers developing implementations of SCert must be fully aware of the memory model of the target execution platform.

Summary

This chapter has introduced core concepts about group communication, replication techniques and transactional systems. The construction of efficient DSTMs is a relatively new topic, with a limited amount of literature around, although there is plenty of literature about the replication of systems that are similar to STMs, such as databases and distributed shared memories. We analyzed both the limitations and strengths of current state-of-the-art solutions.

Next chapter introduces SPECULA, a system that can speculatively execute transactional and non-transactional code in a safe manner, by being able to rollback all changes made to memory if a mis-speculation is detected. It integrates with certification-based replication to enable the speculative commit of memory transactions, while their GSO is being defined in background.

3 SPECULA

I think... I am the third.

– Rei Ayanami, Neon Genesis Evangelion (TV Series)

This chapter describes SPECULA, a system that can speculatively execute transactional and non-transactional code in a safe manner, by being able to rollback all changes made to memory if a mis-speculation is detected. It is integrated with certification-based replication to enable the speculative commit of memory transactions, while their GSO is being defined in background.

The chapter is organized as follows. Sections 3.1 and 3.2 present, respectively, an in depth description of the problem we identified and of how we intend to solve it. Section 3.3 identifies work solution in the design space. Section 3.4 describes the target environment in which SPECULA operates. Section 3.5 describes the architectural design of our system, and Section 3.6 describes the set of properties guaranteed by it. Section 3.7 focuses on the system operation, providing a detailed overview of its internals. Section 3.8 discusses the strengths and weaknesses of SPECULA. Finally, Section 3.9 addresses details of the prototype developed for the JVM platform.

3.1 The Problem

The main motivations behind the creation of DRSTMs are scalability and reliability. Regarding scalability, a commodity cluster is much cheaper than a supercomputer, but it is also typically much more difficult to scale systems horizontally than vertically, due to the cost of inter-node communication. Regarding reliability, high-availability requisites are very common in real world applications. However, high-availability must be achieved at the minimum possible cost, thus replication protocols should be simultaneously effective and efficient.

Most replicated systems communicate through some sort of computer network. Due to distance and medium propagation speed, computer networks feature much higher latency and lower bandwidth than the bus of a computer. This means that if a computer stops executing application's code to communicate with another node, it is wasting the possibility to execute millions of instructions, as it is just sending or receiving data. Therefore, programmers struggle to minimize the effect of the communication latency, either by reducing the number of communication steps or the amount of data to exchange, or by overlapping communication with useful computation. In this context, the cost of the communication can be measured by the number of instructions that could be executed while the process is waiting for the communication exchange to terminate.

Certification-based replication schemes have shown to offer good performance on multi-master database environments. In those settings, there are several sources of delay. First, transaction processing is subject to pre-execution stages like parsing and query optimization. Second, database transactions are required to access stable storage synchronously. These effects dilute the costs induced by communication. It is worth noticing that in certification-based replication most communication costs are associated with the execution of an atomic broadcast primitive, that can take two or more communication steps and requires the exchange of multiple messages (Coulouris, Dollimore, & Kindberg 2002).

In a DRSTM, many of the costs above are not present. This amplifies the relative cost of communication. Therefore, naive ports of DBMSs protocols to the DRSTMs may offer poor performance (Palmieri, Quaglia, Romano, & Carvalho 2010). Figure 3.1 depicts the transaction execution times observed in two benchmarks, STMBench7 and TPC-W, being the former for STMs and the later for DBMSs. As we can see, almost 80% of all memory transactions executed in the STMBench7 benchmark took less than 1 *ms* to finish, while a similar percentage of database transactions executed in the TPC-W benchmark needed almost 10 *ms*.

There are several approaches to mitigate the performance loss due to costs involved in inter-replica coordinations:

- One approach consists in reducing the amount information exchanged in the messages, to speed up the message exchange. This can be achieved using some encoding techniques, such as Bloom filters. This is the approach used by D^2STM, and experimental results show that it is an effective solution (Couceiro, Romano, Carvalho, & Rodrigues 2009).

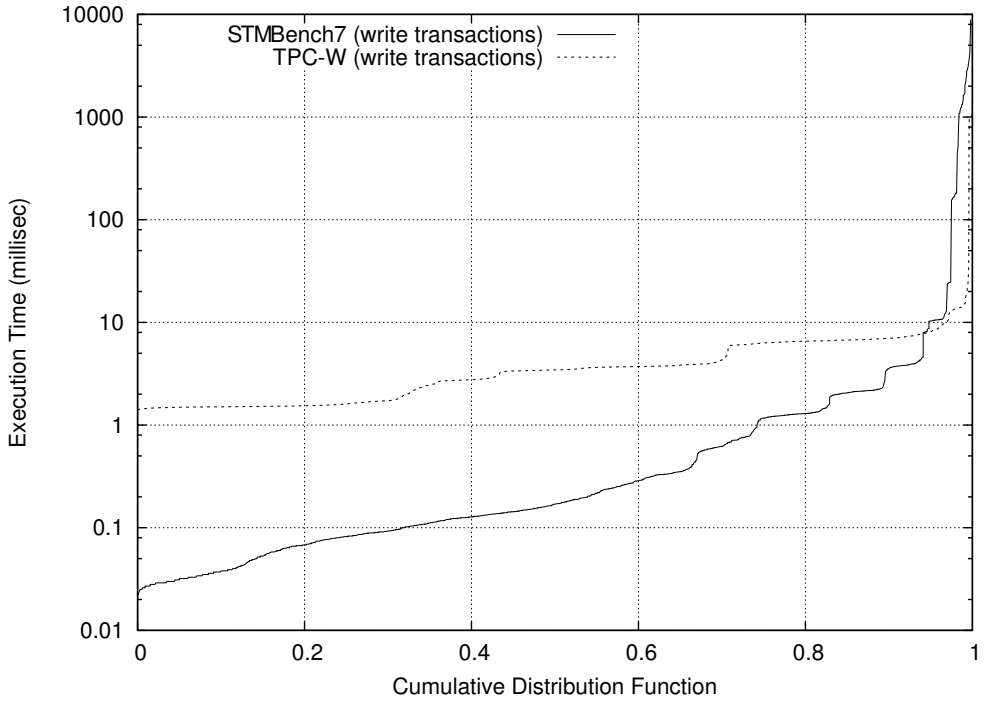


Figure 3.1: Execution time of database and memory transactions (Romano, Carvalho, & Rodrigues 2008).

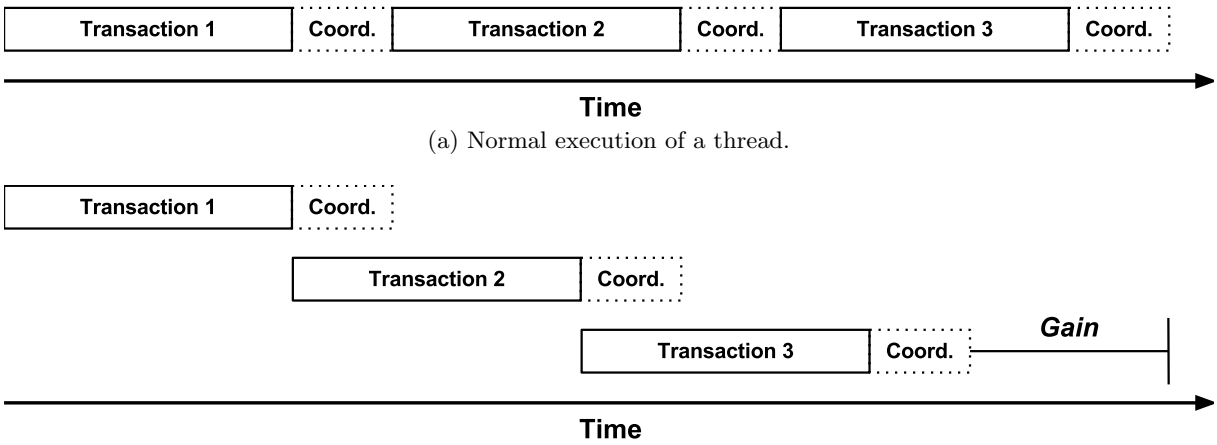
- Another approach would be to start a speculative commit when the transaction begins, in an attempt to run most of the coordination in parallel with the computation of the transaction. This approach has several limitations. First it requires the read and write set of the transaction to be accurately estimated, which is generally hard (unless the system is restricted to use static transaction). Another disadvantage is that coordination is still much slower than computation, so the process would be required to wait in any case. The experimental results in (Palmieri, Quaglia, Romano, & Carvalho 2010) show communication taking around 6 to 26 times more time than execution when considering an AB-delivery time of 2 *ms*. To put these results in a different perspective, conducting the same tests over the same network environment, but with 10 times faster machines, will result in a speedup varying between around 1.03 and 1.14, which truly reflects the amount of computational power that is wasted by standard solutions.

- A third alternative consists in using speculation in the coordination protocol to start the certification of the transaction earlier, and proceed with the (speculative) execution of other transactions before the final outcome of the coordination is known. On LAN networks, experimental results show that the OAB-delivery order matches the AB-deliver

order around 85% of the time (Kemme, Pedone, Alonso, Schiper, & Wiesmann 2003). This property establishes an upper bound on the amount of mis-speculations that can occur. SCert (Carvalho, Romano, & Rodrigues 2011) uses this approach.

In this chapter we propose a novel approach, that builds on the strategies listed above but aims at exploring in higher degree the idea of executing transactions speculatively.

3.2 The Solution



(a) Normal execution of a thread.

(b) Execution of a thread in SPECULA.

Figure 3.2: Comparison between SPECULA and the normal execution of a thread.

As we have already stated, certification-based replication is an optimistic scheme. A transaction tx is locally executed assuming that there will be no conflicts with other transactions, and that, therefore, tx will commit. Only when the transaction terminates, will coordination with other nodes be established. On systems using standard certification-based replication schemes, while tx is being certified, the thread that is committing it stays blocked. The system that we propose here, named SPECULA, avoids blocking by letting the thread run based on a speculative snapshot of the system state (this snapshot may be invalidated when coordination is completed).

SPECULA integrates with a certification-based replication protocol and extends its default level of optimism. Figure 3.2b depicts the optimistic process, and we can compare it to a classic execution depicted in Figure 3.2a. In SPECULA, if transaction tx is locally valid, it is *speculatively committed* before the results of its global validation are known. This allows the

committing thread to unblock and execute code while slow inter-node communication is taking place in background. New transactions can be executed in the same thread while tx is being certified. This creates a flow dependency among transactions. Both those transactions and concurrent ones gain access to the snapshot created upon the speculative commit of tx, so they can also depend on previous speculations, by reading speculatively committed data.

The trade-off of being more optimistic is having to undo more modifications when mis-speculations occur, i.e., when the global validation of a speculatively committed transactions fails. SPECULA has the ability to undo changes made to both the transactional and the non-transactional data of the application. Moreover, it resumes execution where the wrong speculation was initially performed.

Ideally, all speculation should be transparent for both the programmer and the application. It is also crucial that the system guarantees the correct execution of the application, which means that the application should present the exact same behavior as if it was running over a non-speculative environment. As it will become clear later in the text, SPECULA achieves these goals by controlling both the memory and the execution flow of the application.

3.3 Design Space

Now that we have given an overview of the SPECULA approach, it is possible to compare it with the related schemes that we have listed previously. SPECULA fits into the set of distributed and fully replicated software transactional memory systems that rely on certification-based replication. It exploits the possibility of safely executing speculative computation to overlap computation with slow inter-node computation, hence effectively minimizing the negative impact of network latency. A comparison with other systems is depicted in Table 3.1.

3.4 System Model

We consider a system composed by a set of processes $\prod = \{p_1, ..., p_n\}$ that communicate through a message passing interface. We assume that a majority of processes is correct and that the remaining minority may fail according to the fail-stop (crash) model. Furthermore, we assume that the system is asynchronous but augmented with an unreliable failure detector, so

	Replication Scheme		Optimization Techniques
	Active	**Certification-based**	
D²STM		X (Non Voting)	Bloom filter read-set encoding
AGGRO	X		Speculative value propagation + Speculative execution (fully transactional)
SCERT		X (Non Voting)	Speculative value propagation
SPECULA		X (Voting and Non Voting)	Speculative value propagation + Speculative execution

Table 3.1: SPECULA in the design space.

that a primary partition view synchronous GCS (Birman & Joseph 1987) can be implemented. We assume the use of a GCS ensuring the following properties on the delivered views:

- *Self-inclusion:* if process p delivers view v_i, then p belongs to v_i.
- *Strong view-synchrony:* messages are delivered in the same view in which they were sent.
- *Primary component view:* the sequences of views delivered are totally ordered and for any two consecutive views v_i, v_{i+1} there always exists a v_i-correct process p that belongs to both views.
- *Non-Triviality:* when a process fails or it is partitioned from the primary view, it will be eventually excluded from the primary component view.
- *Accuracy:* a correct process is eventually included in every view delivered by the GCS.

The GCS has to provide provide an Uniform Atomic Broadcast communication service and an Uniform Reliable Broadcast (URB) communication service. Uniform AB was formally described in Section 2.1.1. The Uniform AB service also guarantees FIFO ordering on the AB-delivered messages. FIFO ordering is formalized as following:

- *FIFO Order:* if a process p FIFO-broadcasts message m before message m', then any correct process FIFO-delivers m before m'.

The URB service exports communication primitives `URB-broadcast(`m`)` and `URB-deliver(`m`)`, analogous to those provided by the Uniform AB service. It features all the guarantees of the Uniform AB service, with the exception of the Uniform Total Order property.

Regarding transactions, we consider that they are dynamic and snapshot deterministic, i.e., their read-set is not known *a-priori*, and their execution over a given snapshot always produces the same write-set, i.e., writes depend solely on reads.

3.5 System Architecture

The architecture of each SPECULA replica is depicted in Figure 3.3, in which the components that are either new or that we have modified are highlighted in bold. These new and modified components add the following functionality to the system:

- The transactional engine was modified to support the notion of *speculative state*. SPECULA propagates the write-sets of speculatively executed transactions to their following transactions, while ensuring the correctness properties described in Section 3.6.
- The SPECULA approach requires that, in case of mis-speculation, it is possible to rollback the state of the application. This means undoing writes made to both transactional and non-transactional shared state, as well as rolling back the execution flow of threads that executed the aborted transactions, all in a way that preserves the system's correctness. To do this, we use a modified JVM that has support for capturing and resuming continuations (Haynes, Friedman, & Wand 1984), and developed *Class Loader* that makes modifications to classes prior to loading them. The STM was also enhanced with the ability to rollback changes made to the transactional state.
- The replication protocol was modified to be aware of the existence of speculative transactions, as their presence requires the execution of preventive measures, in order to preserve the correctness of the system.

It is possible to rely on a set of different technologies to implement all the functionalities described above. For our prototype, we decided to use the following software components: the

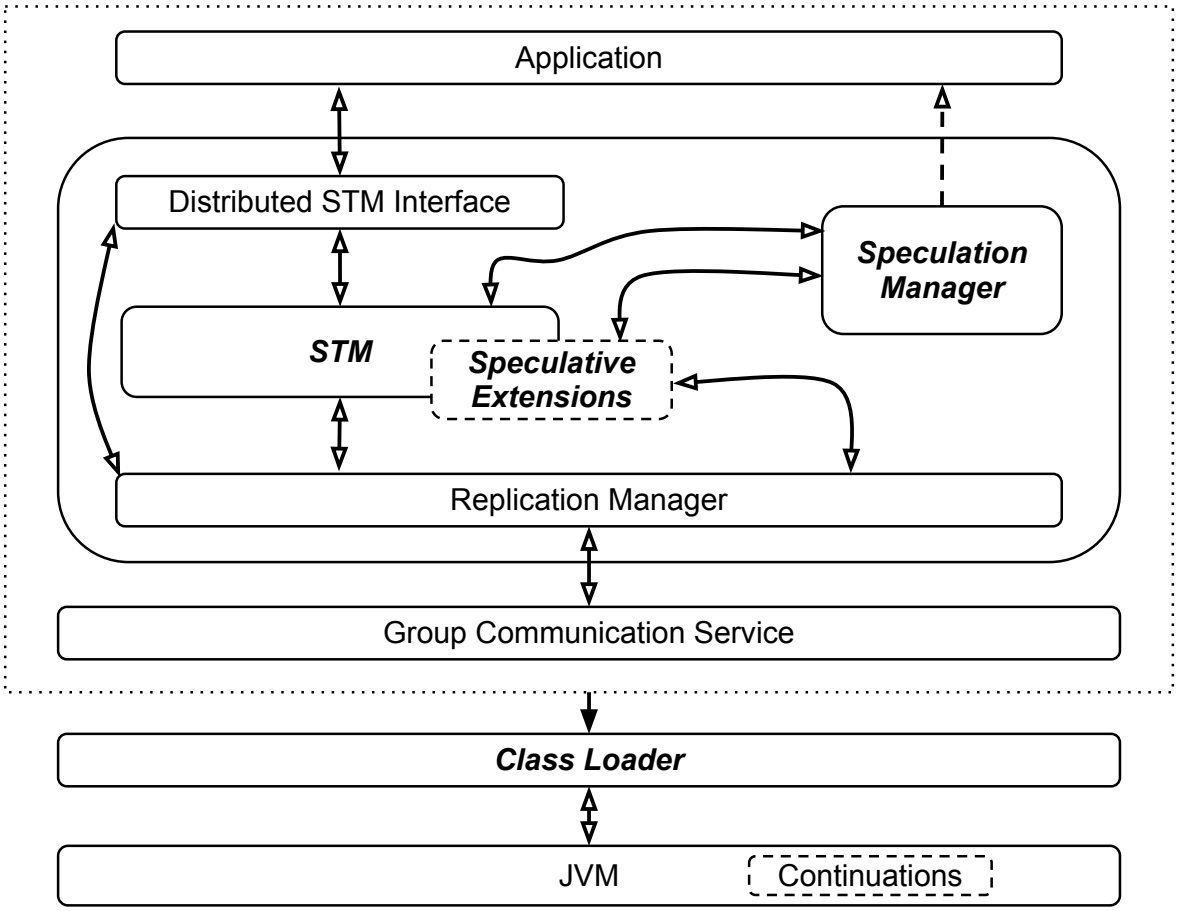


Figure 3.3: The system's architecture.

JVM is a modified version[1] of the OpenJDK[2] that offers support for capturing continuations; the STM is the JVSTM, as it features Multiversion Concurrency Control (MVCC), which makes it easier to add support for speculative state; the GCS is the Appia toolkit (Miranda, Pinto, & Rodrigues 2001).

The *Class Loader* uses the ASM bytecode manipulation framework (Bruneton, Lenglet, & Coupaye 2002) to perform modifications to the application. These modifications intend to facilitate the life of the programmer, by injecting code that will automatically add support to each thread for capturing continuations and create all the data structures that the Speculative Manager (SM) requires. In turn, the SM will control the state of the application. This is represented in Figure 3.3, by the connector between the SM and the Application.

3.6 System Properties

SPECULA was designed to hold most of the properties of its underlying system. The Replication Manager ensures one-copy serializability (1SR). According to what Bernstein et al. wrote in (Bernstein, Hadzilacos, & Goodman 1987), 1SR determines that the interleaved execution of clients' transactions must be equivalent to a serial execution of those transactions on a single replica. 1SR is more than a consistency model, and is the most common correctness criterion among transactional replication protocols.

SPECULA also maintains properties of the underlying STM, such as opacity and strong progressiveness. For the latter we should only consider transactions that do not depend on any speculation.

However, SPECULA does not guarantee that read-only transactions are abort-free, thus it is not multiversion permissive (Perelman, Fan, & Keidar 2010). This limitation derives from the fact that read-only transactions can depend on speculations.

[1] `http://wikis.sun.com/display/mlvm/StackContinuations`
[2] `http://openjdk.java.net`

3.7 Operation

In this section, we describe how SPECULA works. Although our prototype was implemented over a specific set of software components, the algorithms presented here are technology agnostic. Nevertheless, we assume that the underlying transactional engine features MVCC.

To simplify the description, we omit information regarding the concurrency control mechanisms used locally at each replica. Also, as the pseudo-code serves only as a programmatic description of the general idea and of how to implement it, we omit the declaration of some straightforward procedures to increase its readability.

We shall start with the disambiguation of some concepts. A transaction is *speculatively committed* if, upon its commit request, it successfully passes on its local validation. The write-set of a speculatively committed transaction is applied locally, making it visible for future local transactions. The thread that executed the transaction proceeds normally, as the commit procedure returns. The system is, therefore, optimistic about the outcome of the transaction's global validation. The write-set of a speculatively committed transaction becomes available as *speculative state*.

We call *speculative transaction* to a transaction that depends on any speculation, i.e., one that has read speculative state or was executed over the coordination phase of a previously speculatively committed transaction.

A speculation ends up being either right or wrong. When the result of the global validation of a transaction is known, the node that executed the transactions has to act accordingly. If the speculation reveals itself correct, the transaction is *finally committed*, which leads to the system-wide apply of its write-set as *final state*. Otherwise, the system has to hide the transaction's write-set, which was made (locally) public during its speculative commit, and abort all transactions that depend directly or indirectly on the wrong speculation.

During its life, a transaction is always in one of the states depicted in Figure 2.1. In SPECULA, when on state *committing*, a transaction is speculatively committed, while when on state *committed*, it is finally committed.

3.7.1 Speculative Execution Support

In order to support the safe execution of both transactional and non-transactional code over speculative state, it is required to keep track of some data. Therefore, per thread, we maintain the following state:

- A FIFO queue containing all transactions that are in state *committing*.
- The oldest speculatively committed transaction that has been aborted since the last synchronization (described ahead).

As we are considering the use of an STM featuring MVCC, read-only transactions constitute a special case. MVCC schemes typically offer abort-free read-only transactions, as all transactions read from a consistent snapshot (i.e., one that is serializable). In SPECULA, a transaction may read speculative values, hence, it may have to abort due to a mis-speculation. As a result, read-only transactions have to be validated. This poses a significant but inevitable computational overhead.

To simplify the description of the algorithms here presented, we assume that all transactions are globally validated according to the order by which their commit requests were issued, i.e., if the commit request for transaction tx_a is issued before the commit request for transaction tx_b, then tx_a is globally validated before tx_b, and, therefore, if both transactions (final) commit, tx_a commits before tx_b. Notice that systems featuring MVCC typically do not offer this guarantee, because read-only transactions do not need to be validated. In Section 3.9 we describe how this detail is handled in our prototype.

Algorithm 3.1 Thread event handlers.

```
 1: Map<Thread,ThreadContext> contexts ← ∅

 2: upon event start(Thread thread) do
 3:     var ThreadContext tc ← createThreadContext()
 4:     put(contexts, thread, tc)
 5:     tc.oldestAbortedTx ← null
 6:     tc.committingTxs ← ∅
 7: end event handler

 8: upon event end(Thread thread) do
 9:     var ThreadContext tc ← get(contexts, thread)
10:     sync(tc)
11: end event handler
```

Algorithm 3.2 `ThreadContext` data structure.

```
1: data structure ThreadContext
2:     SpeculativeTransaction oldestAbortedTx
3:     FIFOQueue<SpeculativeTransaction> committingTxs
4: end data structure
```

Algorithm 3.1 describes the actions performed upon the start and the end of a thread. Upon its start, a thread t gets associated with a `ThreadContext`, and vice-versa, in a one-to-one relationship between both. The `ThreadContext` data structure is described in Algorithm 3.2. It holds information regarding the thread to which it is associated. On field $oldestAbortedTx$, it holds a reference to the oldest speculatively committed transaction that has been aborted since the last synchronization, and on field $committingTxs$, it holds a reference to a queue containing all speculatively committed transactions that are still waiting for their final commit.

When a thread terminates, a synchronization is forced. The algorithm for synchronizing a thread is described in Algorithm 3.3, by function `sync`. It consists in waiting until all transactions present in the $committingTxs$ queue are either final committed, or one of them is aborted. In the first case, the thread proceeds normally. Otherwise, in the second case, the thread is rolled-back, i.e., all the modifications made to shared state performed on the thread since the speculative commit of the aborted transaction are undone, and the thread resumes its execution on the commit request of the oldest aborted transaction, from where the application is informed of the inability to commit that transaction. After a synchronization, a thread resumes its execution with the $committingTxs$ queue in its `ThreadContext` empty.

Thread termination is not the only event that requires a synchronization point to be forced. All non-transactional operations, like for instance, the output of data to the screen, have to be preceded by a synchronization point. For our Java prototype, we carefully analyzed the set of non-transactional operations present on the JVM, and we describe them in detail on Section 3.9.1. Each thread should also make periodic calls to the `mustSync` function and synchronize accordingly. Function `mustSync` checks if there is any speculatively committed transaction in the $committingTxs$ queue that was aborted. If that is the case, the thread should synchronize as soon as possible, as it is doomed to be rolled-back.

Continuations are essential for our speculative execution support. A continuation reifies the program's control state. A common analogy used to explain the concept is that continuations are the dynamic version of the GOTO statement, although much more powerful. From the

Algorithm 3.3 Speculative execution support.

```
1: Integer MAX_COMMITTING_TXS ← getProperty("MAX_COMMITTING_TXS")
2: Integer MIN_COMMITTING_TXS ← getProperty("MIN_COMMITTING_TXS")
3: Integer specLimit ← MIN_COMMITTING_TXS

4: void sync(ThreadContext tc)
5:     var undoLogs ← ∅ // LIFO queue
6:     var abortAllRemainingTxs ← false
7:     var resumePoint ← null
8:     for all SpeculativeTransaction tx ∈ tc.committingTxs do
9:         if abortAllRemainingTxs then
10:            abort(tx)
11:            push(undoLogs, tx.undoLog)
12:        else
13:            wait while tx.state = State.COMMITTING
14:            if tx.state = State.ABORTED then
15:                abortAllRemainingTxs ← true
16:                resumePoint ← tx.resumePoint
17:                push(undoLogs, tx.undoLog)
18:            end if
19:        end if
20:    end for
21:    tc.committingTxs ← ∅
22:    tc.oldestAbortedTx ← null
23:    if abortAllRemainingTxs then
24:        restoreNonTransactionalSharedState(undoLogs)
25:        resumeContinuation(resumePoint)
26:    end if
27: end function

28: boolean mustSync(ThreadContext tc)
29:    if tc.oldestAbortedTx ≠ null then
30:        return true
31:    end if
32:    return false
33: end function

34: void gotAborted(ThreadContext tc, SpeculativeTransaction tx)
    // assumes that transactions are always aborted
    // from the oldest to the newest
35:    if tc.oldestAbortedTx = null then
36:        tc.oldestAbortedTx ← tx
37:        specLimit ← max(MIN_COMMITTING_TXS, specLimit / 2)
38:    end if
39: end function

40: void gotCommitted(ThreadContext tc, SpeculativeTransaction tx)
41:    remove(tc.committing, tx)
42:    specLimit ← min(MAX_COMMITTING_TXS, specLimit + 1)
43: end function
```

perspective of our work, they can be seen as a snapshot of a thread, which saves the content of all its local variables and the program counter. Continuations allow us to, in case of mis-speculation, resume the execution at the precise moment where it should be resumed.

3.7.2 Modifications to the STM

Existing STMs have not been designed to take speculative state into account. We propose a scheme that enables STMs to provide transactions with speculative values, while ensuring the whole system's correctness, namely the opacity property described in Section 2.4.3.

Algorithm 3.4 describes the data structures required. We use the concept of versioned boxes introduced by Cachopo et al. in (Cachopo & Silva 2006). Let us assume that `Box`, `Body` and `Transaction` are data structures already available in the underlying STM. A `Box` represents a transactional variable, while a `Body` represents a value that was once written to a box by a committed transaction, i.e., a `Body` represents a version in MVCC terms. `SpeculativeBox` extends `Box` by holding references to both speculative and final bodies, on fields *specBody* and $finalBody$, respectively. A `SpeculativeBody` represents a body created during a speculative commit, and extends `Body` by holding a reference, on field tx, to the transaction that wrote it. A `FinalBody` represents a body created during a final commit, and extends `Body` by holding a reference, on field *specBody*, to its equivalent `SpeculativeBody`, i.e., the speculative body that was created and placed on the same box as him, with the exact same value, upon the speculative commit of a now finally committed transaction. `SpeculativeTransaction` extends `Transaction` by holding, on field *state*, the state of the transaction (*executing*, *committing*, *committed* or *aborted*); on field $specSnapshotID$, the oldest snapshot created by a speculatively committed transaction that is accessible to the transaction; on field *resumePoint*, a reference to the continuation that will be resumed in case of mis-speculation; on field *threadContext*, a reference to the context of thread where the transaction executes; finally, on field *undoLog*, a reference to the undo log that will be applied to non-transactional shared state in case of mis-speculation.

In standard MVCC, a transaction begins by obtaining the identifier of the newest snapshot. During its life, a transaction always reads from that snapshot. Access to speculative transactional data follows a similar approach. In SPECULA, $specSnapshotID$ and $snapshotID$ delimit what we call the *speculative transactional window*. As executing transactions get speculatively

Algorithm 3.4 `SpeculativeTransaction`, `SpeculativeBox`, `SpeculativeBody` and `FinalBody` data structures.

```
 1: data structure Transaction
 2:     Integer snapshotID
 3:     Set<Pair<Box,Body>> readSet
 4:     Set<Pair<Box,Object>> writeSet
 5: end data structure

 6: data structure SpeculativeTransaction extends Transaction
 7:     State state
 8:     Integer specSnapshotID
 9:     Continuation resumePoint
10:     ThreadContext threadContext
11:     Map<Address,Object> undoLog
12: end data structure

13: data structure Box
14:     Body body
15: end data structure

16: data structure SpeculativeBox extends Box
    // inherited field body is not used
17:     SpecBody specBody
18:     FinalBody finalBody
19: end data structure

20: data structure Body
21:     Integer version
22:     Object value
23:     Body next
24: end data structure

25: data structure SpeculativeBody extends Body
26:     SpeculativeTransaction tx
27: end data structure

28: data structure FinalBody extends Body
29:     SpecBody specBody
30: end data structure
```

committed, and speculatively committed transactions are either final committed or aborted, the window *moves*. A speculative commit adds new speculative state, so the right limit of the window moves forward, which increases its size. In turn, an abort or a final commit makes speculative state eventually disposable, and therefore, the left limit of the window moves forward, which reduces its size. Notice, however, that when a transactions begins, it obtains a fixed view of the transactional window. This scheme is fully exempt from reorderings or removal of bodies from boxes, at the cost of having to apply write-sets twice: once upon the speculative commit of a transaction and again, upon its final commit.

Algorithm 3.5 Transaction initialization.

```
1: Integer newestSnapshotID ← 0
2: Integer oldestSpecSnapshotID ← 0
3: FIFOQueue<SpeculativeTransaction> allCommittingTxs ← ∅

4: void start(SpeculativeTransaction tx, Thread thread)
5:     var ThreadContext tc ← get(contexts, thread)
6:     tx.state ← State.EXECUTING
7:     tx.snapshotID ← newestSnapshotID
8:     tx.specSnapshotID ← oldestSpecSnapshotID
9:     tx.resumePoint ← null
10:    tx.threadContext ← tc
11:    tx.undoLog ← ∅
12: end function
```

The Begin of a Transaction Algorithm 3.5 depicts the initialization procedure of a transaction. Upon its begin, a transaction tx acquires not only the identifier of the newest snapshot available, but also the left limit of its speculative transactional window, respectively $snapshotID$ and $specSnapshotID$.

Read Algorithm 3.6 depicts the read procedure. A transaction tx starts by consulting its internal write-set. If there is value available there, it is returned to the application. Otherwise, the read procedure starts searching through the state inside the target box. The search starts by the speculative bodies, as they represent a possible future. If it exists, the matching body is the one with the highest version that is both lower or equal to $tx.snapshotID$ and greater or equal to $tx.specSnapshotID$. If no body is found within the speculative transactional window of the transaction, the search continues through the final bodies, from where the first body with version lower or equal to $tx.snapshotID$ is returned to the application.

If a speculative body is a match but belongs to the write-set of a transaction that was

Algorithm 3.6 Read procedure.

```
1: Object read(Box box, SpeculativeTransaction tx)
2:     var body ← readFromWriteSet(tx, box)
3:     if body ≠ null then
4:         return body
5:     end if
   // look into the speculative values
6:     body ← box.specBody
7:     while body ≠ null do
8:         if body.version < tx.specSnapshotID then
   // reached the end of the transaction's speculative window
9:             break
10:        end if
11:        if body.version ≤ tx.snapshotID then
12:            if body.tx.state = State.ABORTED then
13:                throw MisspeculationException
14:            end if
15:            put(tx.readSet, box, body)
16:            return body.value
17:        end if
18:        body ← body.next
19:    end while
   // look into the non-speculative values
20:    body ← box.finalBody
21:    while true do
22:        if body.version ≤ tx.snapshotID then
23:            put(tx.readSet, box, body)
24:            return body.value
25:        end if
26:        body ← body.next
27:    end while
28: end function
```

aborted, the application is informed of the inability of the transaction to read from the target box. This behavior prevents transactions from accessing an inconsistent snapshot, which avoids violating opacity.

Let us examine the examples depicted in Figures 3.4 and 3.5. In both examples, transaction tx starts with $tx.snapshotID$ equal to 8 and $tx.specSnapshotID$ equal to 5. tx is a read-only transaction, therefore, its internal write-set is empty. In Figure 3.4, tx is reading from box b_a. The read procedure finds a match on body version 6, as 6 is within tx's speculative window ($6 \geq tx.specSnapshotID \wedge 6 \leq tx.snapshotID$). Therefore, the value inside body version 6 is returned to the application.

In Figure 3.5, tx is reading from box b_b. The read procedure finds no matching body within the speculative versions inside b_b, as both versions 9 and 4 are outside tx's speculative window ($9 > tx.snapshotID \wedge 4 < tx.specSnapshotID$). However, the final body version 8 is a match,

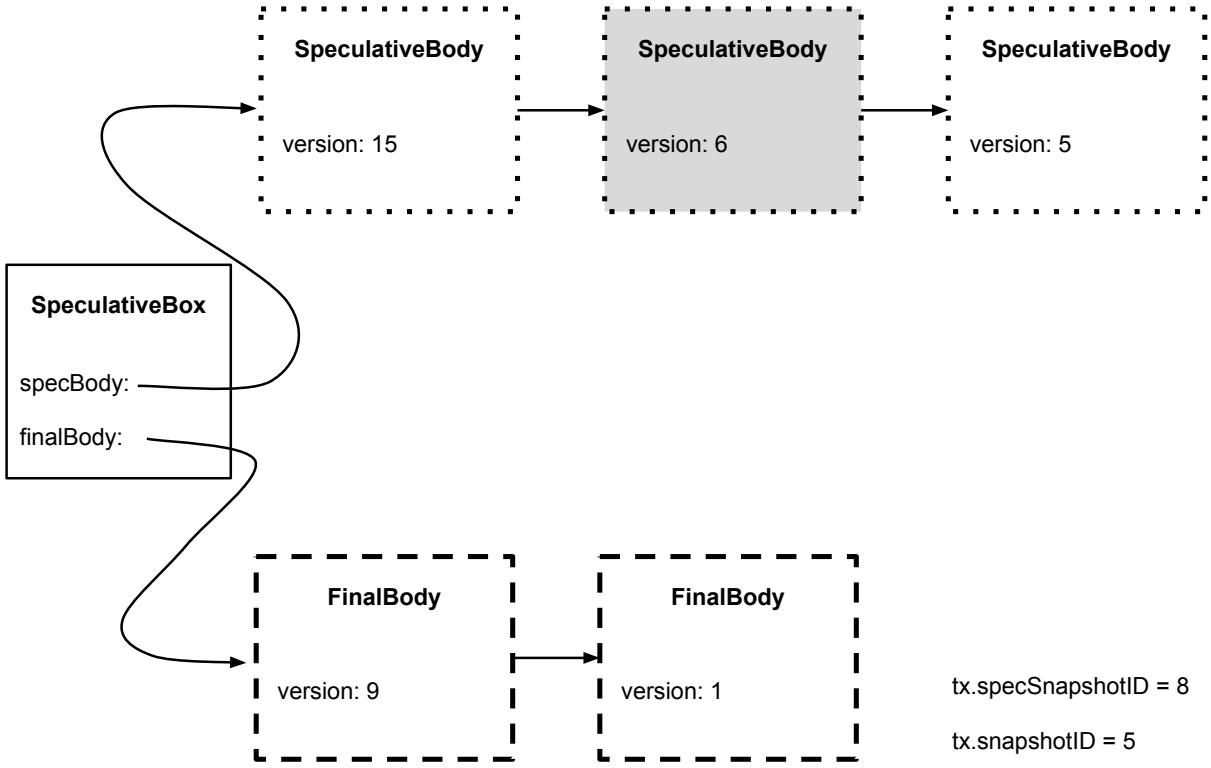


Figure 3.4: Reading example number 1.

as $8 \leq tx.snapshotID$. Therefore, the value inside final body version 8 is returned to the application.

Validation In standard MVCC, a transaction cannot commit if it conflicts (read–write conflict) with a concurrent but already committed transaction. This is the only reason why a transaction cannot commit.

In SPECULA, there are two additional reasons that may force a transaction to abort:

- If a transaction has read from the write-set of a speculatively committed transaction that was aborted – we call this a *speculative data dependency*;
- If a transaction was executed on a thread that has to be rolled-back and resumed at a point prior to the transaction's starting point – we call this a *speculative flow dependency.*

It is possible to avoid aborting transactions just due to a speculative flow dependency. By tracing the execution flow of a rolled-back thread during its reexecution, it is possible to detect

Algorithm 3.7 Validation procedures.

```
 1: boolean validate(SpeculativeTransaction tx)
 2:     if mustSync(tx.threadContext) ∨ tx.state = State.ABORTED then
 3:         return false
 4:     end if
 5:     if isReadWriteTransaction(tx) then
 6:         return validateReadWrite(tx)
 7:     else
 8:         return validateReadOnly(tx)
 9:     end if
10: end function

11: boolean validateReadWrite(SpeculativeTransaction tx)
12:     for all Pair<Box,Body> entry ∈ tx.readSet do
13:         var readBody ← entry.second // the body that was read
14:         var NSBody ← entry.first.specBody // the newest speculative body
15:         var NFBody ← entry.first.nonSpecBody // the newest final body
16:         if tx.state = State.EXECUTING then
17:             if NSBody ≠ null ∧ NSBody.version ≥ oldestSpecSnapshotID then
18:                 if readBody instanceof FinalBody then
19:                     return false
20:                 else if readBody instanceof SpecBody then
21:                     if NSBody ≠ readBody ∨ NSBody.tx.state = State.ABORTED ∨ (NSBody.tx.state =
    State.COMMITTED ∧ NFBody.specBody ≠ NSBody) then
22:                         return false
23:                     end if
24:                 end if
25:             else if NFBody ≠ readBody ∧ NFBody.specBody ≠ readBody then
26:                 return false
27:             end if
28:         else if tx.state = State.COMMITTING then
29:             if NFBody ≠ readBody ∧ NFBody.specBody ≠ readBody then
30:                 return false
31:             end if
32:         end if
33:     end for
34:     return true
35: end function

36: boolean validateReadOnly(SpeculativeTransaction tx)
37:     for all Pair<Box, Body> entry ∈ tx.readSet do
38:         var readBody ← entry.second
39:         if readBody instanceof SpecBody ∧ readBody.tx.state = State.ABORTED then
40:             return false
41:         end if
42:     end for
43:     return true
44: end function
```

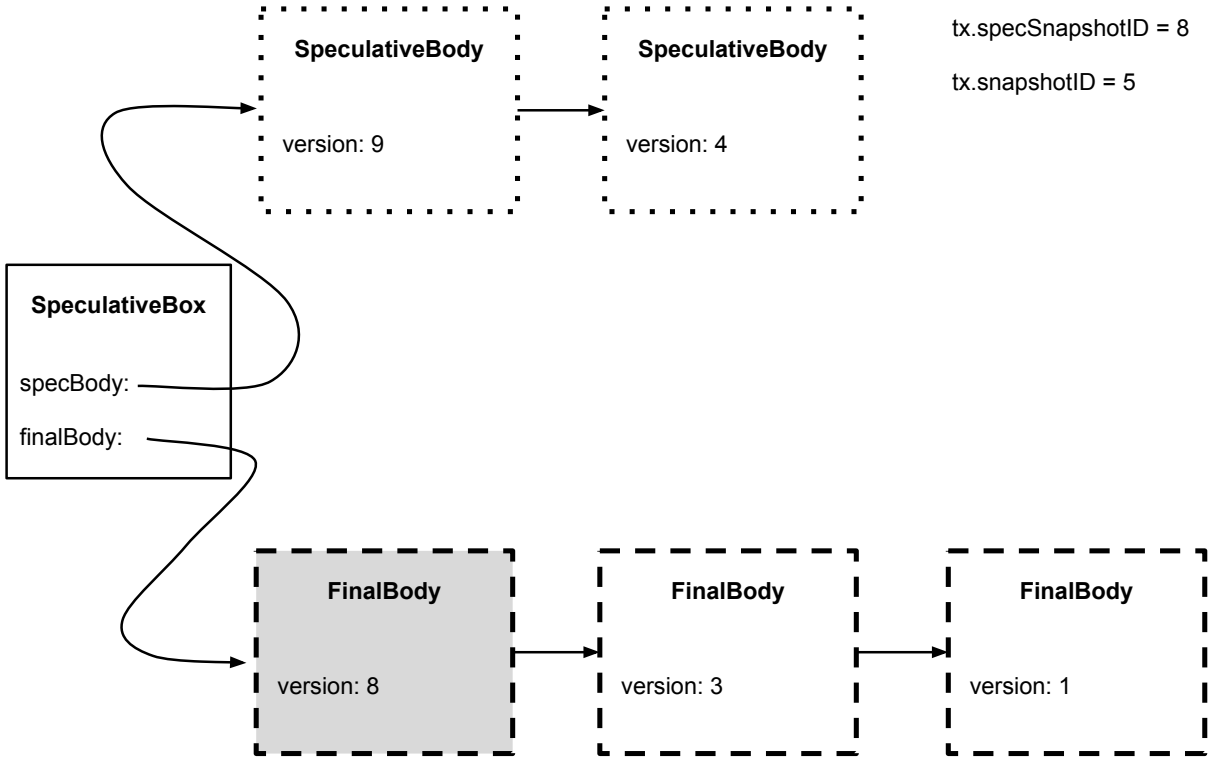


Figure 3.5: Reading example number 2.

if its flow is being repeated, thus leading to the execution of the same memory transactions. This is, however, outside of the scope of this work.

All transactions are validated to ensure that their commit preserves the system's correctness. Validation is commonly made in two steps: first *local*, then *global*. Local validation is typically not mandatory. It filters transactions that will certainly fail on global validation, avoiding announcing them on the network. Performing local validation is normally a trade-off between computation and communication. However, in SPECULA, local validation is mandatory, as it is responsible for ensuring that speculative commits preserve the system's correctness. Global validation is always mandatory, as it is responsible for ensuring that a given transaction can be committed system-wide, as final state.

Algorithm 3.7 depicts the validation procedures. Both local and global validations fail if i) the transaction was already aborted in either a previous synchronization or cascading abort procedure (described ahead), or if ii) function `mustSync` returns **`true`**, which reveals a speculative flow dependency.

An update transaction tx_a fails to pass on its local validation if i) it has read from the

Algorithm 3.8 Commit procedures.

```
 1: void commit(SpeculativeTransaction tx)
 2:     if isReadWriteTransaction(tx) then
 3:         if validate(tx) then
 4:             wait while size(tx.tc.committingTxs) > tx.tc.specLimit
 5:             AB-broadcast(tx)
 6:             tx.state ← State.COMMITTING
 7:             newestSnapshotID ← tx.snapshotID ← newestSnapshotID + 1
 8:             specApplyWriteSet(tx)
 9:             enqueue(allCommittingTxs, tx)
10:         else
11:             throw CommitException
12:         end if
13:     end if
14:     tx.resumePoint ← captureContinuation()
15:     if tx.state = State.ABORTED then
    // this is only reached if the thread was rolled-back
16:         throw CommitException
17:     end if
18:     enqueue(tx.threadContext.committingTxs, tx)
19: end function

20: void finalCommit(SpeculativeTransaction tx)
21:     tx.state = State.COMMITTED
22:     if isWriteTransaction(tx) then
23:         newestSnapshotID ← tx.snapshotID ← newestSnapshotID + 1
24:         finalApplyWriteSet(tx)
25:         gotCommitted(tx.context, tx)
26:         moveSpecWindow()
27:     end if
28: end function
```

write-set of a transaction in state *aborted*, or if ii) it has read from speculative state that is equivalent to final state, but that is not the newest available, or if iii) it has not read from the latest speculative state available, or if iv) there is no speculative state available and tx_a has not read from the latest final state available, or the equivalent speculative state. Transaction tx_a fails to pass on its global validation if it has not read from the latest final state available, or the equivalent speculative state.

A read-only transaction tx_b fails to pass on its (final and only) validation if it has read from the write-set of speculatively committed transaction that ended up being aborted.

Commit Algorithm 3.8 depicts the commit procedures. Any locally valid update or read-only transaction tx pass to state *committing* when its commit request is issued by the user. If tx is an update transaction, its write-set is applied to memory as speculative state, and it is then enqueued in the $allCommittingTxs$ queue, which is used to adjust the limits of the

speculative transactional window. Otherwise, if tx is a read-only transaction, it is exempt from the two previously described actions, but both read-only and locally valid update transaction are associated with a continuation that will be resumed, if necessary, in case of mis-speculation. Finally, tx is enqueued in the *committingTxs* queue of its `ThreadContext`.

Final commits occur according to the GSO. If tx is a read-only transaction, it simply passes to state *committed*. If otherwise, tx is an update transaction, its write-set is applied to the transactional memory as final state, its associated `ThreadContext` is notified of the final commit, and it tries to move the speculative transactional window, in order to hide its speculatively committed write-set. The field *specBody* of each `FinalBody` created during the final commit of a local transaction points to its equivalent `SpeculativeBody`, so that the validation procedure can check for valid speculations. In turn, the write-sets of remote transactions are always directly applied as final state.

Algorithm 3.9 Abort procedure.

```
 1: void abort(SpeculativeTransaction tx)
 2:     tx.state = State.ABORTED
 3:     moveSpecWindow()
 4:     if tx.state = State.EXECUTING then
 5:         standardAbort(tx)
 6:     else if tx.state = State.COMMITTING then
 7:         if isReadWriteTransaction(tx) then
 8:             gotAborted(tx.threadContext, tx)
 9:         end if
10:     end if
11: end function
```

Abort Algorithm 3.9 depicts the abort procedure. All aborting transactions pass to state *aborted* and try to move the speculative transactional window. If the aborting transaction tx is a read-only transaction, nothing more has to be done. Otherwise, if tx is an update transaction, it is subject to a different set of actions. If tx is in state *executing*, the abort procedure can be delegated to the underlying STM. Otherwise, ff tx is in state *committing*, it notifies its `ThreadContext` of the abort, as it may be the oldest speculatively committed transaction to be aborted.

Notice that there is no need to clean the entire speculative state with a cascading abort procedure, upon the abort of a speculatively committed transaction.

3.7.3 Moving the Speculative Transactional Window

Algorithm 3.10 Moving the limits of the speculative transactional window.

```
1: void moveSpecWindow()
2:     for all SpeculativeTransaction tx ∈ allCommittingTxs do
3:         if tx.state ≠ State.COMMITTING then
4:             oldestSpecSnapshotID ← tx.snapshotID + 1
5:             dequeue(allCommittingTxs)
6:         else
7:             return
8:         end if
9:     end for
10: end function
```

When a transaction is speculatively committed, it widens the speculative transactional window, as the identifier of the newest available snapshot increases, and it corresponds to the right limit of the window. Upon the commit of a remote transaction, the number of the newest available snapshot also increases, but since no speculative values are added to memory, the increase in the size of the window sorts no effect.

Moving the left limit of the window is a more complex procedure. Algorithm 3.10 depicts it. When a speculatively committed transaction is finally committed or aborted, there is an attempt to move the left limit of the window. Although final commits follow the GSO, aborts do not. Therefore, the oldest transaction in the $allCommittingTxs$ queue may still be waiting for its global validation, while a newer speculatively committed transaction is already aborted. An attempt to move the left limit of the window advances it until a transaction in state *committing* is found. Therefore, while the oldest transaction in the $allCommittingTxs$ queue is in state *committing*, the left limit of the window stays still, and, in turn, if for instance the oldest five transactions in the $allCommittingTxs$ have already left state *committing*, the window is moved five times in a row.

3.7.4 Integration with Replication Protocols

It is possible to use SPECULA with both voting and non-voting certification-based replication protocols.

Algorithm 3.11 Integration with a voting certification-based replication protocol.

```
1: upon event URB-deliver(SpeculativeTransaction tx, boolean validationResult) do
2:     if isLocal(tx) then
   // tx was executed locally
3:         if validationResult then
4:             finalCommit(tx)
5:         else
6:             abort(tx)
7:         end if
8:     else
   // tx was executed remotely
9:         if validationResult then
10:            cleanSpeculativeState(tx)
11:            moveSpecWindow()
12:            newestSnapshotID ← tx.snapshotID ← newestSnapshotID + 1
13:            finalApplyWriteSet(tx)
14:        else
15:            // do nothing, ignore tx
16:        end if
17:    end if
18: end event handler

19: void cleanSpeculativeState(SpeculativeTransaction remoteTx)
20:    for all SpeculativeTransaction localTx ∈ allCommittingTxs do
21:        if shouldHaveReadFrom(localTx, remoteTx) ≠ ∅ then
22:            cascadingAbort(localTx)
23:        end if
24:    end for
25: end function
```

3.7.4.1 Voting

When a node n broadcasts the commit message for a transaction tx, it is certifying that tx can be committed over n's state, both final and speculative. At that point, the final state is the same on all nodes, but the speculative state is not. Therefore, all other nodes have to execute preventive actions before applying tx's write-set to their memory. If they do not, new (local) transactions may gain access to an inconsistent snapshot. The call to preventive actions is depicted in Algorithm 3.11.

Before applying the write-set of a remote transaction, each node performs the cascading abort of all transactions that should have read from the write-set of the remote committing transaction, i.e., a transaction that has read from the newest final (or speculative equivalent) state available.

Let us analyze the problematic scenario exemplified in Listing 3.1. Node 1 has issued the commit confirmation for transaction tx_a. Transaction tx_b was already speculatively committed

Listing 3.1: Example of a problematic scenario.

```
1  // Remember that Transaction.commit() returns after the speculative commit
2
3  // Initially: x.get() == y.get() == z.get() == 100
4
5  // Node 1
6  void doSomething1() {
7    // tx_a
8    Transaction.begin();
9    x.put(x.get() - 50); // x = 50
10   y.put(y.get() + 50); // y = 150
11   Transaction.commit();
12   // tx_a is speculatively committed ----------------------------------------
13
14                                          // Node 2
15                                          void doSomething2() {
16                                            // tx_b
17                                            Transaction.begin();
18                                            y.put(y.get() - 10); // y = 90
19                                            z.put(z.get() + 10); // z = 110
20                                            Transaction.commit();
21
22   // final commit of tx_a -------------------------------------------------
23 }
24                                            // tx_c
25                                            Transaction.begin();
26                                            int a = x.get(); // a = 50
27                                            int b = y.get(); // b = 90
28                                            int c = z.get(); // c = 110
29                                            Transaction.commit();
30
31                                            int sum = a + b + c; // sum = 250
32                                            // when it should be 300
33                                          }
```

at node 2, and it conflicts with tx_a. If tx_b is not aborted before node 2 applies the write-set of tx_a, a transaction tx_c that starts over the new freshly available snapshot will be able to read values written by both tx_a and tx_b until tx_b is aborted. Therefore, this scenario violates opacity: although tx_c will eventually end up being aborted, it was able to read from an inconsistent snapshot, as there is no possible serialization where both tx_a and tx_b commit.

3.7.4.2 Non-voting

Making SPECULA work with a non-voting protocol is slightly more complex than with a voting protocol, as it requires each replica to be aware of speculative dependencies present in other nodes. The typical message used by non-voting protocols to announce transactions on the

Algorithm 3.12 Integration with a non-voting certification-based replication protocol.

```
1: Map<ID,ID> globalToLocalSnapshotID ← ∅
2: Set<ID> badThreadContexts ← ∅
3: Integer finalSnapshotID ← 0

4: upon event AB-deliver(SpeculativeTransaction tx) do
5:    if isLocal(tx) then
   // tx was executed locally
6:       if validate(tx) then
7:          finalCommit(tx)
8:          finalSnapshotID ← finalSnapshotID + 1
9:          put(globalToLocalSnapshotID, tx.snapshotID, finalSnapshotID)
10:      else
11:         abort(tx)
12:      end if
13:   else
   // tx was executed remotely
14:      if contains(badThreadContexts, tx.contextID) ∨ ¬ containsKey(globalToLocalSnapshotID,
   tx.snapshotID) ∨ wasRemotelyCascadingAborted(tx) then
15:         put(badThreadContexts, tx.contextID)
16:         return
17:      end if
18:      if validate(tx, get(globalToLocalSnapshotID, tx.snapshotID)) then
19:         cleanSpeculativeState(tx)
20:         moveSpecWindow()
21:         newestSnapshotID ← tx.snapshotID ← newestSnapshotID + 1
22:         finalApplyWriteSet(tx)
23:         finalSnapshotID ← finalSnapshotID + 1
24:         put(globalToLocalSnapshotID, tx.snapshotID, finalSnapshotID)
25:      else
26:         put(badThreadContexts, tx.contextID)
27:      end if
28:   end if
29: end event handler
```

network contains the read and write-sets of the committing transaction, plus the snapshot timestamp in which it was started. In SPECULA, each node has bodies with different timestamps inside the boxes in its memory. However, although different, these timestamps are equivalent, as they ensure the same total order between versions. Since in non-voting certification-based replication all replicas certify transactions, it is required to identify snapshots uniquely system-wide, so that the announce message tells in a precise way from which snapshot the committing transaction has read.

To make all nodes aware of the speculative flow dependency of the transaction, we give each `ThreadContext` an unique identifier, and change it every time its associated thread is rolled-back. This identifier is placed in the announce message of every transaction. When the global validation of a remote transaction fails, all nodes get to know that they have to abort the following transactions that hold the same identifier (the detection of speculative flow

dependencies).

Speculative data dependencies are recognized when the announce message of a transaction indicates it has read from a snapshot that does not belong to the set of finally committed snapshots.

The last detail are the transactions aborted during cascading abort procedure. Knowing the snapshot accessible by a given transaction is not enough to know if it was aborted in its home node due to a cascading abort. All replicas have to know not only the snapshot accessible by the transaction, but also the newest final committed snapshot accessible by the transaction. With this information, and the transaction's read-set, it is possible to check if the committing transaction has missed a final commit or not.

3.7.5 Speculative Execution Control

High conflict scenarios are not favorable for a speculative scheme like the one we propose. Mis-speculations introduce unwanted load on the system. The overhead created by SPECULA only pays off if a large faction of all speculatively committed transactions are final committed. Notice that in Section 3.1, we stated that experimental results have shown that communication time dominates execution time in DRSTM environments. This means that a thread can execute a high number of transactions in the time required to by a single broadcast.

We decided to implement a scheme that limits the amount of speculations per thread. While the number of transactions in the $committingTxs$ queue of the context of a thread is higher than the value of the *specLimit* variable, the execution of that thread stays blocked. The maximum number of transactions allowed in each queue is dynamically adapted by an algorithm that derives from the *additive increase/multiplicative decrease* algorithm used in TCP (Allman, Paxson, & Stevens 1999). Its key heuristic is that successful speculations should allow more speculations to occur, while, conversely, mis-speculations should decrease the maximum number of speculations allowed. The value of the *specLimit* variable is never smaller than the value of the $MIN\_COMMITING\_TXS$ variable and never greater than the value of the $MAX\_COMMITTING\_TXS$ variable.

This scheme enables the system to better adapt to its surrounding environment.

3.7.6 Restoring Non-transactional State

Algorithm 3.13 Building undo logs for non-transactional shared state.

```
1: upon event write(Thread thread, Address address) do
2:     var threadContext ← get(contexts, thread)
3:     var newestCommittingTx ← tail(threadContext.committingTxs)
4:     if newestCommittingTx = null then
5:         return // there is no speculatively committed transaction
6:     end if
7:     if ¬ containsKey(newestCommittingTx.undoLog, address) then
8:         var currentValue ← read(address)
9:         put(newestCommittingTx.undoLog, address, currentValue)
10:    end if
11: end event handler
```

Few application are fully transactional. Most make use of transactions to access shared state, but perform heavy computations and system calls outside of them. Changes made to non-transactional state have be undone in case of mis-speculation, just like the changes made to transactional state. In order to restore the state of the non-transactional shared memory, at the point where execution is resumed, we propose the construction of undo logs. Since we are using continuations to snapshot the execution flow of a thread, and continuations save the state of the thread's stack, we only need to deal with writes made to non-transactional shared state.

To build an undo log, all writes to non-transactional shared state have be intercepted. Algorithm 3.13 depicts the procedure. When an instruction executed on thread t tries to modify non-transactional shared state, we save the value that is in target address before it is overwritten with the new value. An undo log keeps just one value per memory address: the oldest. Each saved value is kept in the undo log that is associated with the last speculatively committed transaction that was executed on t. If later, a synchronization procedure has to resume the execution of t using a continuation, it applies all undo logs required to restore the state that was available when the continuation that is going to be resumed was captured.

3.7.7 Correctness Arguments

We now provide some informal arguments to show that our algorithm is correct.

All replicas start with the same state and final commit the same transactions, by the same order. Therefore, the final state is always kept coherent among all nodes. This guarantees 1SR, as it is easy to see that, if all replicas go from snapshot s_i to snapshot s_{i+1}, then the interleaved

execution of transactions among all nodes produces a serialization order that matches the one produced by a serial execution, i.e., both executions would end up producing the same snapshot s_n.

The key to guarantee opacity is ensuring that every transaction always reads from the same consistent snapshot. SPECULA's read procedure ensures this by forcing transactions to read from the newest snapshot that was available when they began. Moreover, since no snapshot is ever modified after its creation, and the apply of the write-sets of remote transactions is preceded by a clean-up of the speculative state, temporary inconsistent states never exist. This suffices to guarantee opacity.

Notice that local validation ensures that all speculative commits create potentially serializable snapshots. If a transaction always reads from the same consistent snapshot, its commit produces a new guaranteed consistent snapshot. It is not even possible for a transaction to build an inconsistent write-set.

Finally, as we assume that the program's correctness does not depend on any kind of synchronized access to non-transactional shared state, we are free to undo all modifications made to it.

3.8 Strengths and Weaknesses

SPECULA makes a trade-off between the latency required to commit a transaction and the amount of memory consumed by the middleware. With SPECULA, transactions can (speculatively) commit based on local information, and transaction processing can continue with no further delays. This is achieved at the cost of having to continuously create snapshots of the system. Therefore, SPECULA consumes more memory than other non-speculative transactional memory systems. However, the increase in memory usage occurs for a limited amount of time, and one that is expected to be short, as the final commit of a transaction hides its speculative commit from new transactions, and most memory transactions are themselves very short, as stated in Section 3.1. SPECULA has also the drawback of requiring the write-set of a transaction to be applied twice: once when the transaction commits locally and again, when the transaction is applied system-wide. We expect this cost to be paid off by the overlap of communication with computation.

On the other hand, new transaction are always provided with a consistent snapshot of the system, and are shielded from the concurrent execution of other transactions. Thus SPECULA preserves the programming model that makes software transactional memory appealing. Furthermore, in face of low-conflict workloads, cascading aborts due to speculation rarely occur, enabling SPECULA to effectively hide network delays.

3.9 Java Prototype Implementation

We developed a Java prototype as a proof-of-concept. The prototype makes use of the components described in Section 3.5. This section describes how some minor low-level details were implemented and some of the issues found along the development were solved.

3.9.1 Dealing with Non-transactional Operations

The set of non-transactional operations contains all methods that send data out of, or into the JVM. These methods carry the **`native`** flag at the bytecode level, as they have to be implemented in native code, in order to communicate with the underlying operating system. In fact, a native method may not execute any non-transactional operation, but it is very hard to know that, as it requires complex binary inspection. Therefore, we inject a synchronization point before any call to a native method.

Another problem we had to deal with is that the Java Compatibility Kit requires certified JVMs to enforce loading all classes in the `java.*` package with the bootstrap class loader, thus not allowing us to modify those classes in runtime. A way to overcome this limitation is by providing modified versions of those classes to the JVM in a custom bootstrap class path. However, Oracle's JVM license forbids any kind of modification to the JVM's core classes, which should also be true for most proprietary JVMs. Due to this, we have decided to use a different approach: package filtering. We inject a synchronization point before any call of a method that belongs to, or was inherited from a class that that is part of the `java.*` package.

3.9.2 Bytecode Manipulation

In order to ease the life of the programmer, we modify the application's code in runtime, so that SPECULA can work in a fully transparent way.

Speculative Execution Support To be able to transparently offer speculative execution support to all threads of the application, we modify all classes containing a concrete implementation of a method with the signature **public void** `run()`. An example is provided. Listing 3.2 depicts the original code of class `Example`, and Listing 3.3 depicts the modified code. Although the modifications are made at the bytecode level, we feel that it is both easier to explain and understand them at the source level. We rename the original method `run()` to `specula$run`, and preserve the **final** modifier if it is present. If the method belongs to an object of a class that does not implement the `java.lang.Runnable` interface, than it certainly is not bootstrapping a thread, so running the original `run()` method will suffice. Otherwise, we check if we are already running on a thread with speculative support, and act accordingly. The `specula$bootstrapping` field is used to flag if the method `run()` of the object was the bootstrapping point of the executing thread or not. If it was, a synchronization has to be enforced before the method returns.

It does not suffice to modify only classes implementing the `java.lang.Runnable` interface, because they can inherit a **public void** `run()` method, which turns them into valid non-abstract classes.

Listing 3.2: Original class `Example`.

```
1 class Example {
2
3   public void run() {
4     for (int i = 0; i < 100; i++);
5   }
6
7 }
```

Intercepting Writes to Non-transactional Shared State Saving non-transactional shared state requires that we intercept the following opcodes: **PUTSTATIC**, **PUTFIELD**, **AASTORE**, **IASTORE**, **LASTORE**, **FASTORE**, **BASTORE** and **CASTORE**. The first two opcodes

Listing 3.3: Modified class **Example**.

```
1  class Example {
2
3    private boolean specula$bootstrapped = false;
4
5    public void run() {
6      if (! (this instanceof java.lang.Runnable)) {
7        specula$run();
8        return;
9      }
10     final ThreadContext tc = ThreadContext.getCurrent();
11     if (tc == null) {
12       specula$boostrapped = true;
13       ThreadContext.makeNew();
14       Continuation.runWithContinuationSupport(this);
15     } else {
16       specula$run();
17       if (specula$bootstrapped) {
18         tc.syncThreadContext();
19         specula$bootstrapped = false;
20       }
21     }
22   }
23
24   private void specula$run() {
25     // the original run() method
26     for (int i = 0; i < 100; i++);
27   }
28
29 }
```

modify respectively, static and non-static fields, while the remaining modify arrays. We then use Java's reflection capabilities, more precisely classes `java.lang.reflect.Field` and `java.lang.reflect.Array`, to save and restore non-transactional shared state.

3.9.3 Mixed Issues

Garbage Collection JVSTM features a scheme that removes unnecessary bodies from boxes. A body becomes unnecessary if no active transaction has access to it, and if it does not belong to the newest snapshot available. A very complete description of how this scheme works can be found in (Fernandes & Cachopo 2011).

Our scheme fits JVSTM garbage collection algorithm naturally, because a `SpeculativeBox` holds speculative and final state separately. Therefore, the removal of a final body does not affect any speculative body, and vice-versa.

Read-only Transactions JVSTM provides abort-free read-only transactions. As we stated in Section 3.7.1, all transactions have to be committed according to the GSO, and by using JVSTM read-only transactions are not right out of the box. To overcome this problem, we associate read-only transactions with update transactions. When an update transaction is speculatively committed, it is made responsible for validating all the read-only transactions that were speculatively committed on the same thread since the last speculatively committed update transaction. Notice that when using a non-voting replication protocol, all nodes have to know the snapshots that were read by the read-only transactions sheltered by update transactions, so that the result of global validation is equal at all nodes.

Showing an example, let us assume that transactions tx_a, tx_b and tx_c were speculatively committed by this same order, on the same thread, and that tx_a and tx_b are read-only transactions, while tx_c is an update transaction. When tx_c is speculatively committed, it is made responsible for validating both tx_a and tx_b. Then, later, upon the global validation of tx_c, tx_a and tx_b are validated (before the validation of tx_c), thus the validity of tx_c depends on the validity of both tx_a and tx_b (due to a speculative flow dependency). If, for instance, a thread only speculatively commits read-only transactions, they are validated by the synchronization procedure.

Useless Computation Although SPECULA modifies the applications bytecode and automatically inserts the synchronization points needed to ensure its correctness, long computations can create high amounts of useless work, as they can be performed while is already known that the executing thread is doomed to be rolled-back due to a mis-speculation. In order to minimize the amount of discarded computation, the programmer can himself introduce periodic checks to function *mustSync* and act accordingly. There is an obvious trade-off between the number of calls to function *mustSync* and the amount of useless computation performed by the application.

Summary

This chapter has presented the SPECULA system: the motivations behind its development, its design and target environment. We have covered its operation in detail and discussed not only the desired properties for a system of this kind, but also their impact on real-world scenarios. We

also discussed the implementation of the developed prototype, and how we dealt with specific details of the JVM platform.

The next chapter will present the evaluation of the developed prototype.

4 Evaluation

Are you saying that I'm lazy?

– Misato Katsuragi, Neon Genesis Evangelion (TV Series)

This chapter reports the results of an experimental study aimed at evaluating and comparing the performance of the system described in the previous chapter, in face of a variety of synthetic workloads that mimic both extreme and more common situations. This was done using a prototype developed for the JVM platform.

It begins by describing, in Sections 4.1 and 4.2 the experimental environment, the employed settings, and the criteria used in our evaluation. Section 4.3 presents the results of executing a simple micro-benchmark. Section 4.4 presents the results of executing STMBench7, a highly complex benchmark. Finally, Section 4.5 presents a brief analysis over all results.

4.1 Experimental Environment and Settings

All the experiments presented here were performed in a cluster of eight nodes, each one equipped with two Intel Xeon E5506 at 2.13GHz and 8 GB of RAM, running GNU/Linux 2.6.32 – 64 bits. The nodes are interconnected via a private Gigabit Ethernet switch.

The JVM is a development snapshot of version 1.7.0 of the OpenJDK. The AB service provided by the Appia GCS toolkit (Miranda, Pinto, & Rodrigues 2001) uses a a sequencer-based algorithm to order messages on top of a multicast layer that relies on point-to-point TCP links (Défago, Schiper, & Urbán 2004; Cachin, Guerraoui, & Rodrigues 2011). It also implements batching of the sequencer messages. We have set the batching value either to the same value of $MAX\_COMMITING\_TXS$, or to one (i.e., no batching) on the baseline system. The batching timeout is set to 250 *ms*.

Unless stated otherwise, all runs were executed with a single thread per node producing

transactions. The coordination among replicas is achieved using a non-voting certification-based replication protocol. We compare our prototype with that of a system using the same configuration but with the speculative extension turned off.

We assume a that the environment is stable, in which no nodes crash/stop or deviate from their normal behavior.

4.2 Evaluation Criteria

The two main evaluation criteria for our system are speedup, and the average time required to execute and commit a transaction. We also consider, although as secondary criteria, the abort rate, the number of speculatively executed transactions that were committed, the overhead in the local execution of transactions and the amount of time that worker threads were halted by the execution control algorithm. These criteria allow us to measure the efficiency and the effectiveness of our system. Ideally, a system should achieve high throughput and low latency. However, in most practical system, there is a trade-off between these two aspects. In this evaluation we aim at assessing if SPECULA achieves a reasonable compromise between these two criteria.

The number of speculatively executed transactions that were committed and the overhead in the local execution of transactions allows us to assess the the effectiveness and the benefits of being speculative.

4.3 Bank Benchmark

4.3.1 Description

The bank benchmark was first proposed by Herlihy, Luchangco, & Moir (2006) as a performance evaluation test for the DSTM2 transactional engine. It creates a synthetic workload that mimics a bank environment. Update transactions transfer "money" between one pair of accounts, while read-only transactions sum the value present in all accounts, hence their result is (or should, in a opaque system) always be the same. It is a simple, yet powerful benchmark, as one can use it to generate very specific workloads.

4.3.2 Configuration

Configuration A In this configuration we set the percentage of update transactions to 100% and make them write on distinct "bank accounts" (there are $2 \times NMachines$ accounts, so that node with identifier I only writes on accounts $2 \times I$ and $(2 \times I) + 1$), thus there is no data contention as there are no concurrency conflicts.

Configuration B In this configuration we keep the percentage of update transactions to 100% but force all nodes to access the same accounts (i.e., the system is configured with just two accounts). In this scenario there is an extremely high data contention.

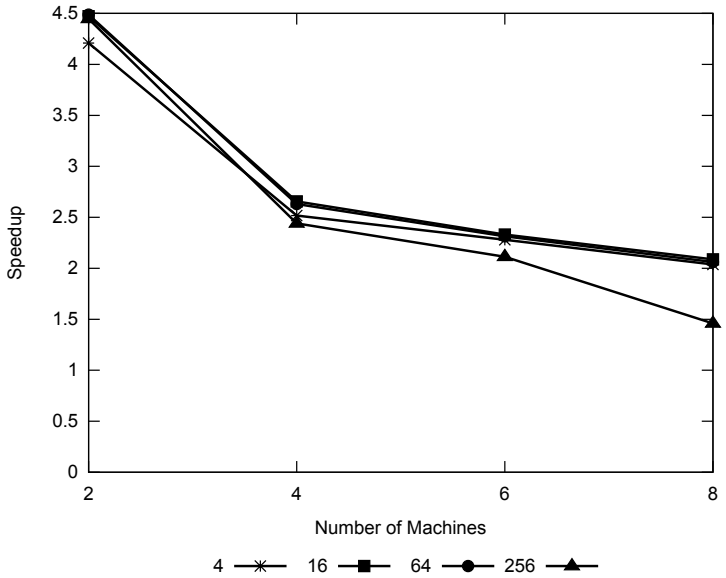
4.3.2.1 Results

All the results here presented were obtained by running the benchmark for 120 seconds.

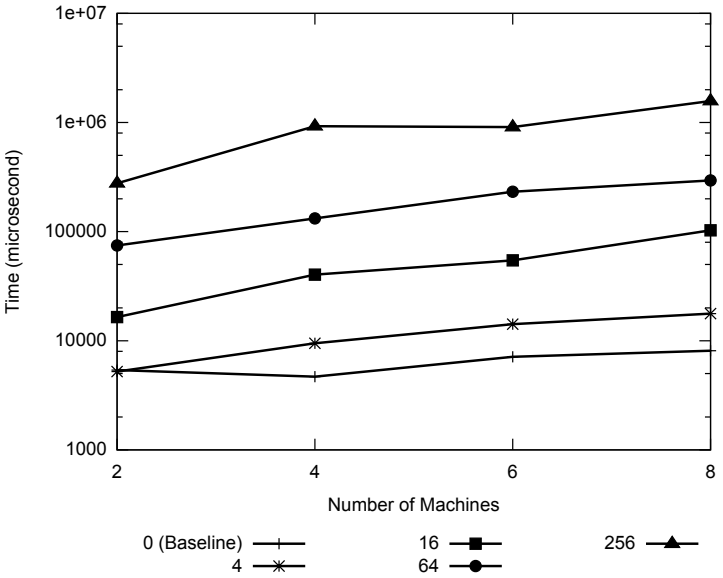
Configuration A Figure 4.1a depicts the speedup achieved in this configuration. As the number of replicas increased, the achieved speedup decreased. To understand this behavior we need to look at the results in detail.

Figure 4.1d depicts the network latency observed. It can be verified that latency values were very high. Although the batching layer is a source of additional latency, batching alone is not the cause for latency observed. In fact, what happens is that transactions are very short, therefore the system generated messages at a rate higher than the GCS could sustain, congesting the networking layer. This can be explained as follows. Figure 4.1c depicts the local execution time of transactions, which includes execution, local validation and the capture of a continuation. Even if the overhead observed is not negligible, it was exacerbated by the fact that the transactions in this benchmark are uncommonly short. Nevertheless, considering the worst case scenario observed, where a transaction took 200 *ms* to execute, this means the systems should have been able to execute 5000 transactions per second. Therefore, the batching threshold can be achieved very quickly, even when set to the maximum of 256.

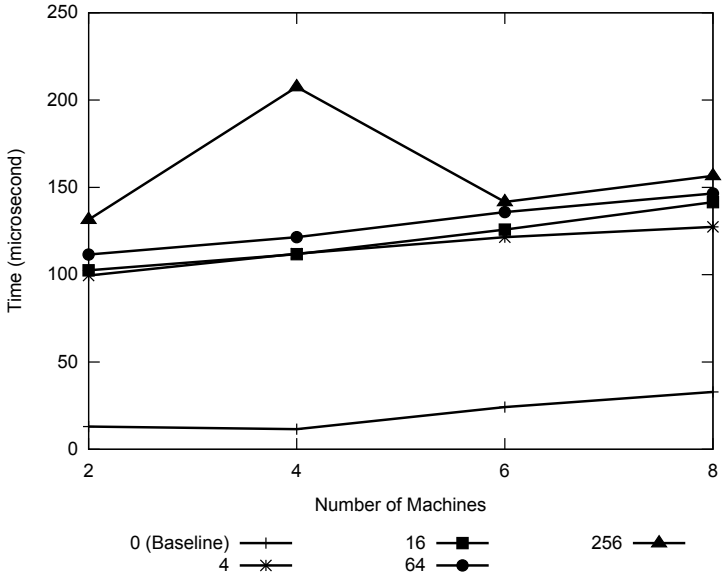
Due to the saturation of the network, the execution control algorithm kept blocking the creation of new transactions. Looking to the rows "Blocked Time" in tables 4.1 we can see that in some scenarios execution stayed blocked for more than 100 of the 120 seconds run. With



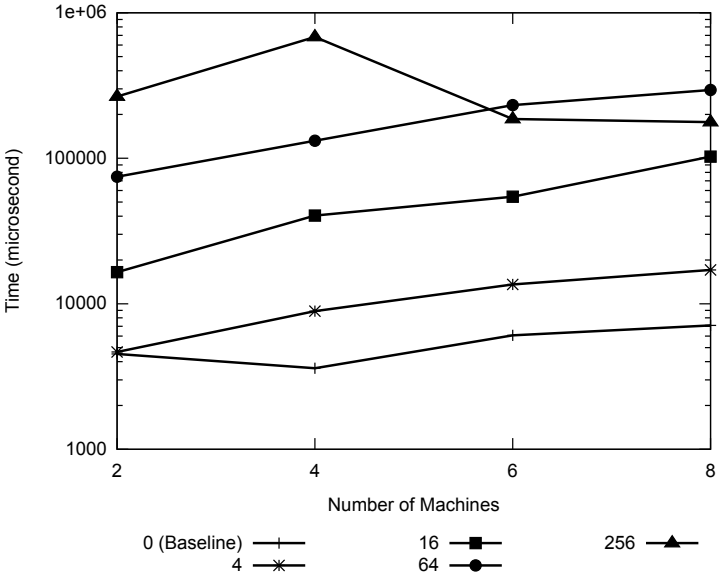
(a) Speedup – varying the value of $MAX\_COMMITTING\_TXS$.



(b) Total execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.



(c) Local execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.



(d) Network latency – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4.1: Bank Benchmark – Configuration A.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	47 243	28 145	35 677	19 337	25 785	32 427	16 780	30 591
Committed Transact.	47 239	28 141	35 673	19 333	25 781	32 427	16 776	30 587
Spec. Transact. Committed	47 243	28 145	35 677	19 337	25 785	32 423	16 780	30 591
Blocked Time (*ms*)	94 843	103 156	99 591	106 551	103 672	101 442	108 328	102 030
Total Committed Transact.	229 728							

(a) $MAX\_COMMITTING\_TXS$ equal to 4.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	84 734	13 903	39 953	20 136	7 425	41 423	17 927	16 391
Committed Transact.	84 734	13 903	39 953	20 136	7 425	41 423	17 927	16 391
Spec. Transact. Committed	84 730	13 899	39 949	20 132	7 421	41 439	17 923	16 387
Blocked Time (*ms*)	78 683	109 141	97 718	105 501	113 235	97 358	107 386	108 223
Total Committed Transact.	241 892							

(b) $MAX\_COMMITTING\_TXS$ equal to 16.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	53 262	17 734	16 192	21 146	23 384	41 906	30 521	35 156
Committed Transact.	53 262	17 734	16 192	21 146	23 384	41 906	30 521	35 156
Spec. Transact. Committed	53 258	17 730	16 188	21 142	23 380	41 902	30 517	35 152
Blocked Time (*ms*)	90 717	106 034	107 472	105 113	103 448	95 762	100 994	98 520
Total Committed Transact.	239 301							

(c) $MAX\_COMMITTING\_TXS$ equal to 64.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	21 137	13 969	14 768	19 344	17 225	19 160	23 737	41 907
Committed Transact.	47 243	28 145	35 677	19 337	25 785	25 785	16 780	30 591
Spec. Transact. Committed	47 239	28 141	35 673	19 333	25 781	25 781	16 776	30 587
Blocked Time (*ms*)	104 875	109 646	108 694	106 557	107 197	106 386	103 931	96 790
Total Committed Transact.	171 247							

(d) $MAX\_COMMITTING\_TXS$ equal to 256.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	26 580	17 122	10 128	10 034	10 202	10 118	20 621	10 972
Committed Transact.	26 580	17 122	10 128	10 034	10 202	10 118	20 621	10 972
Total Committed Transact.	115 777							

(e) Baseline system.

Table 4.1: Bank Benchmark – Configuration A (values per run).

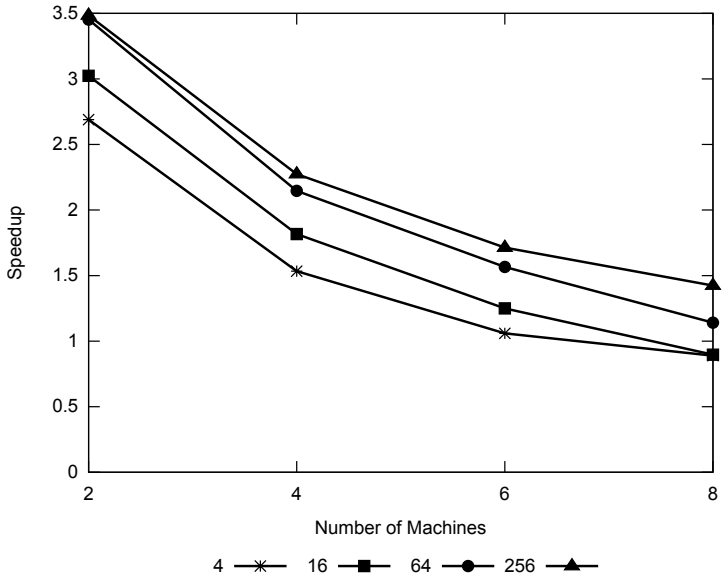
$MAX\_COMMITTING\_TXS = 4$, the network latency and the time required to commit a transaction (depicted in Figure 4.2c) were reasonable, as the latter includes the former. However, 17 *ms* is more than enough time to run four transactions, so execution was also kept blocked for most of the time.

The difference between the number of committed transactions and the number of committed speculative transactions is just the first transaction of the benchmark, which obviously cannot have any speculative dependency, and a few transactions that we use to coordinate the replicas of the system (i.e., barriers).

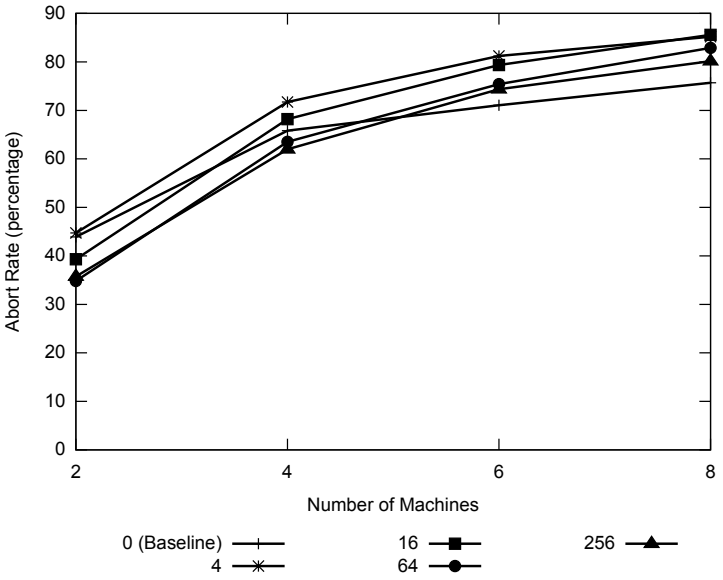
This benchmark is very poor in application logic, having just a few variables of control. The time spent building undo logs of modifications to non-transactional shared state accounted for less than 2% of the whole run.

Based on these results, the best value for $MAX\_COMMITTING\_TXS$ in this configuration is four, as all different values tested achieved very similar speedups, but with $MAX\_COMMITTING\_TXS = 4$ the system's response time was significantly lower.

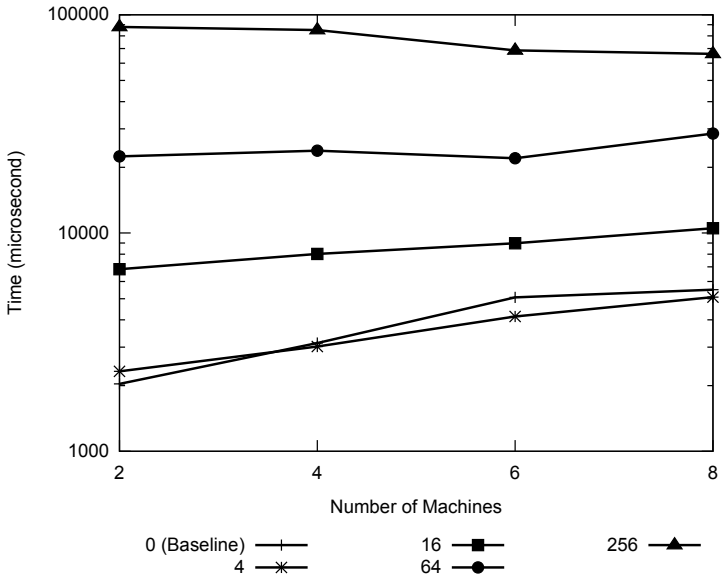
Configuration B Figure 4.2a depicts the speedup achieved in this configuration. At first glance, the fact that we almost always observed positive speedups with such high data contention might be surprising. However, the abort rate depicted in Figure 4.2b explains why we got these results. Notice that the abort rate decreased as we increased the value of $MAX\_COMMITTING\_TXS$. In fact, with just two machines, the abort rate achieved with $MAX\_COMMITTING\_TXS = 64$ was significantly lower than the abort rate of the baseline system's configuration (from around 44% to 35%). This was the due to a large unbalance in the abort rate distribution, with one node committing noticeably more transactions than all the others. The tables in 4.2 depict the situation. As we already stated, certification-based replication is an optimistic scheme, so it delivers poor performance in high contention scenarios. Therefore, the results we observed are normal, as confirmed by the results in Tables 4.2e and 4.2f that show the number of committed transactions by the baseline system with one and four threads, respectively. These results show the baseline system executing and committing less transactions when there is more computational power trying to do work. It happens that the extra computational power only overloads the system, increasing the network latency and the data contention.



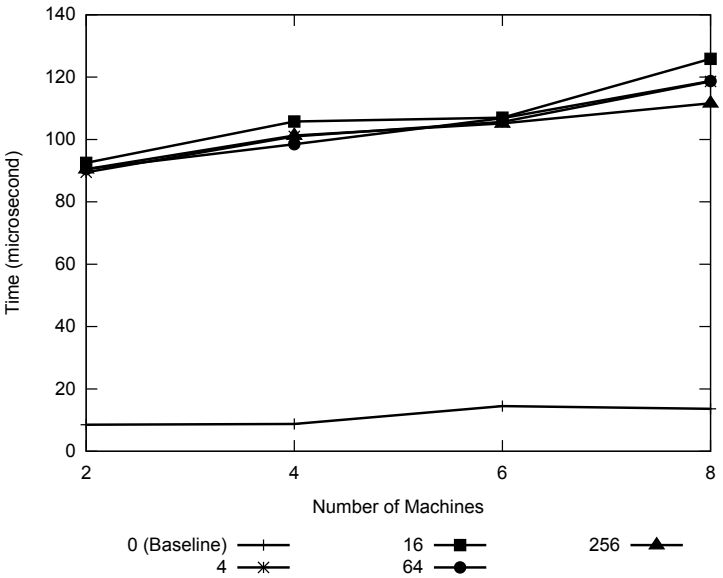
(a) Speedup – varying the value of $MAX\_COMMITTING\_TXS$.



(b) Abort rate – varying the value of $MAX\_COMMITTING\_TXS$.



(c) Network latency – varying the value of $MAX\_COMMITTING\_TXS$.



(d) Local execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4.2: Bank Benchmark – Configuration B.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	34 381	30 992	27 876	32 702	34 153	33 902	27 133	28 482
Abort Rate (%)	43.827	75.455	99.358	91.031	96.393	93.95	93.963	92.904
Committed Transact.	19 313	7 607	179	2 933	1 232	2 051	1 638	2 021
Spec. Transact. Committed	19 282	7 569	175	2 909	1 220	2 023	1 625	2 006
Blocked Time (*ms*)	98 833	100 130	102 209	100 080	99 987	98 937	103 341	102 327
Total Committed Transact.	36 974							

(a) $MAX\_COMMITTING\_TXS$ equal to 4.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	32 249	30 519	32 777	32 527	32 664	32 831	32 337	33 574
Abort Rate (%)	77.478	65.91	92.989	99.988	99.994	88.977	99.988	58.748
Committed Transact.	7 263	10 404	2 298	4	2	3 619	4	13 850
Spec. Transact. Committed	7 259	10 397	2 293	1	0	3 613	1	13 840
Blocked Time (*ms*)	97 276	102 683	99 235	98 986	99 923	99 274	101 302	99 935
Total Committed Transact.	37 444							

(b) $MAX\_COMMITTING\_TXS$ equal to 16.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	37 766	36 322	30 033	34 221	34 416	34 095	38 269	34 048
Abort Rate (%)	58.873	45.124	89.988	99.997	99.997	99.997	75.672	99.997
Committed Transact.	15 532	19 932	3 007	1	1	1	9 310	1
Spec. Transact. Committed	15 528	19 929	3 004	0	0	0	9 307	0
Blocked Time (*ms*)	96 398	101 521	101 300	98 378	99 839	102 538	98 947	100 234
Total Committed Transact.	47 785							

(c) $MAX\_COMMITTING\_TXS$ equal to 64.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	31 573	31 444	38 420	59 667	31 602	37 796	31 802	38 121
Abort Rate (%)	99.997	99.997	99.997	0	99.991	99.997	99.997	99.997
Committed Transact.	1	1	1	59 667	3	1	1	1
Spec. Transact. Committed	0	0	0	59 665	1	0	0	0
Blocked Time (*ms*)	102 048	101 773	101 152	90 854	100 581	99 839	104 461	100 869
Total Committed Transact.	59 676							

(d) $MAX\_COMMITTING\_TXS$ equal to 256.

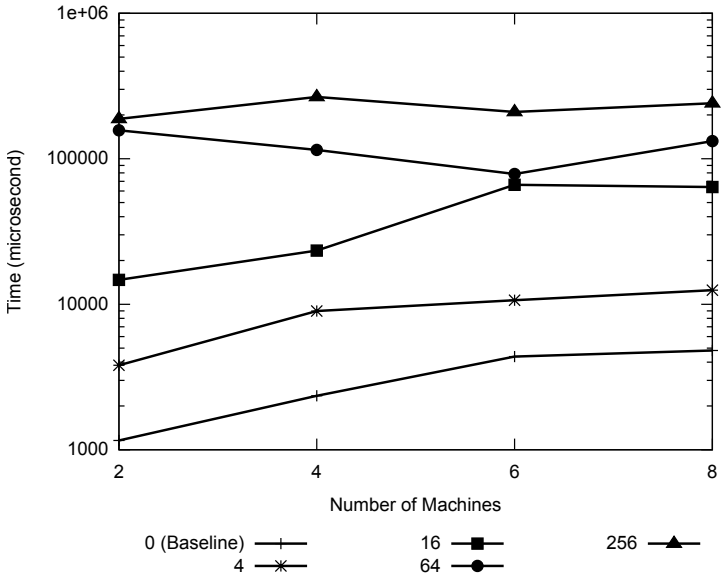
	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	33 484	18 572	19 505	20 121	16 607	17 338	23 431	21 056
Abort Rate	48.423	85.09	81.882	84.017	87.198	85.95	75.622	79.588
Committed Transact.	17 270	2 769	3 534	3 216	2 126	2 436	5 712	4 298
Total Committed Transact.	41 361							

(e) Baseline system – 1 thread.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	37 917	27 562	20 162	27 830	31 452	34 431	28 589	309 35
Abort Rate	84.682	91.833	93.82	92.012	90.315	88.641	92.927	91.44
Committed Transact.	5 808	2 251	1 246	2 223	3 046	3 911	2 022	2 648
Total Committed Transact.	23 155							

(f) Baseline system – 4 threads.

Table 4.2: Bank Benchmark – Configuration B (values per run).



(e) Total execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4.2: Bank Benchmark – Configuration B.

Figure 4.2e depicts the network latency observed. Like in Configuration A, there was a significant latency increase as we increased the value of $MAX\_COMMITTING\_TXS$, namely to 256. Although the total number of executed transactions was similar to that obtained in Configuration A, there was a large number of transactions that were locally aborted, thus never reaching the network. With eight machines and $MAX\_COMMITTING\_TXS = 256$, 18% of all executed transactions did not pass on their local validation.

The time spent building undo logs accounted for less than 0.7% of the whole run, while the time spent restoring non-transactional shared state was inferior to 0.2%.

With these results, the best value for $MAX\_COMMITTING\_TXS$ in this configuration depends of whether latency is an issue or not. If it is, then $MAX\_COMMITTING\_TXS = 4$ and $MAX\_COMMITTING\_TXS = 16$ presented a total execution time of transactions in the same order of magnitude of the baseline system. Otherwise, if the system is not latency sensitive, then $MAX\_COMMITTING\_TXS = 256$ offered consistently the higher speedup.

4.4 STMBench7 Benchmark

4.4.0.2 Description

The STMBench7 benchmark was introduced by Guerraoui, Kapalka & Vitek (2007) as a complex benchmark that generates realistic workloads, by mimicking the operations present in large object-oriented applications, such as CAD/CAM applications. Operations vary significantly in complexity, containing both short and trivial transactions that read a few items, and highly read/write intensive ones, which perform not only deep structural modifications to an object graph with millions of vertices, but also do long transversals on the same graph, resulting in transactions with huge data-sets.

The configuration parameters of this benchmark allow the selection of one of three different workloads: one read-dominated, one with a balanced mix of read-only and update transactions, and one write-dominated. It is also possible to enable/disable the execution of both long transversals and deep structural modifications.

4.4.0.3 Configuration

We analyze the results of both the write and read-dominated workloads. Long transversals and deep structural modifications were disabled, otherwise the number of conflicts becomes unbearable.

The operation ratios (in percentage) of the write-dominated workload are depicted in Table 4.3, and of the read-dominated workload in Table 4.4.

TRAVERSAL:	0.00
TRAVERSAL_RO:	0.00
SHORT_TRAVERSAL:	0.00
SHORT_TRAVERSAL_RO:	47.06
OPERATION:	47.65
OPERATION_RO:	5.29
STRUCTURAL_MODIFICATION:	0.00

Table 4.3: STMBench7 – Write-dominated workload – Operation ratios.

TRAVERSAL:	0.00
TRAVERSAL_RO:	0.00
SHORT_TRAVERSAL:	0.00
SHORT_TRAVERSAL_RO:	47.06
OPERATION:	5.29
OPERATION_RO:	47.65
STRUCTURAL_MODIFICATION:	0.00

Table 4.4: STMBench7 – Read-dominated workload – Operation ratios.

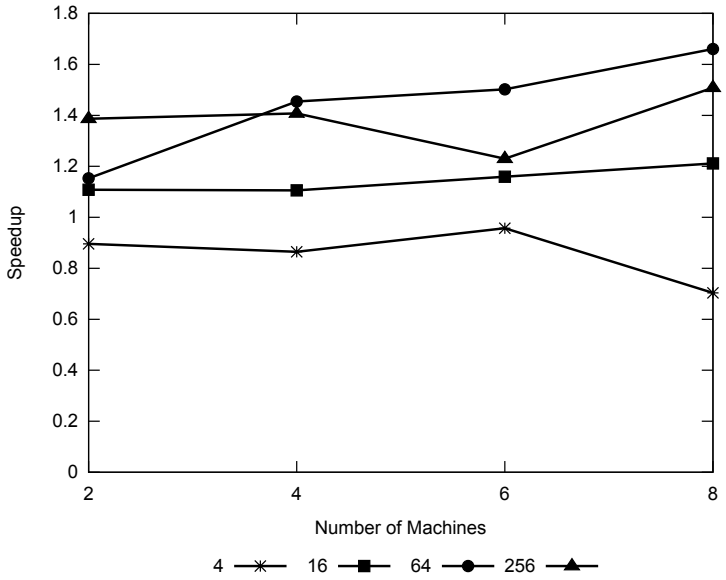
4.4.0.4 Results

All the results here presented were obtained by running the benchmark during 60 seconds.

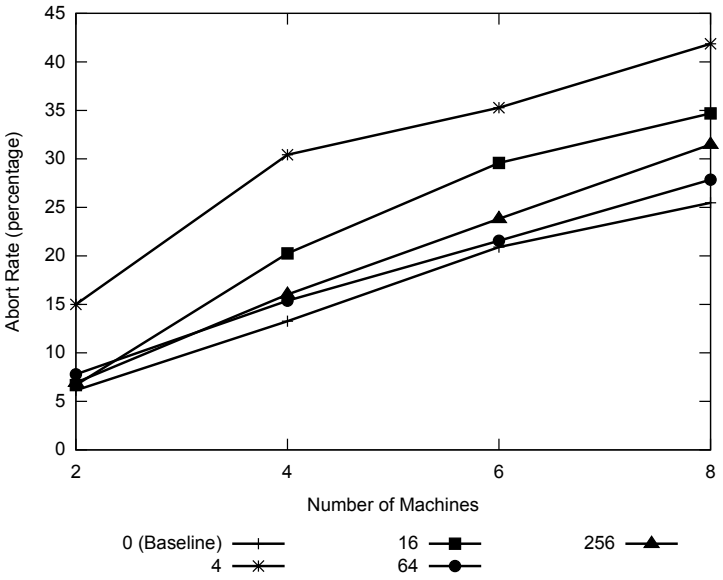
Write-dominated Workload Figure 4.3a depicts the speedup achieved in this configuration. For values of $MAX\_COMMITTING\_TXS > 4$ we always observed positive speedups, with a maximum gain of 1.66 times when $MAX\_COMMITTING\_TXS = 64$.

The abort rate depicted in Figure 4.3b indicates that this configuration suffers from the same unbalance in the abort rate distribution as the Configuration B used in Bank Benchmark. Indeed, the tables in 4.5 confirms it. Tables 4.5e and 4.5f show, once more, that with the increase of the number of working threads in the baseline system, its performance decreased, and in a very considerable manner.

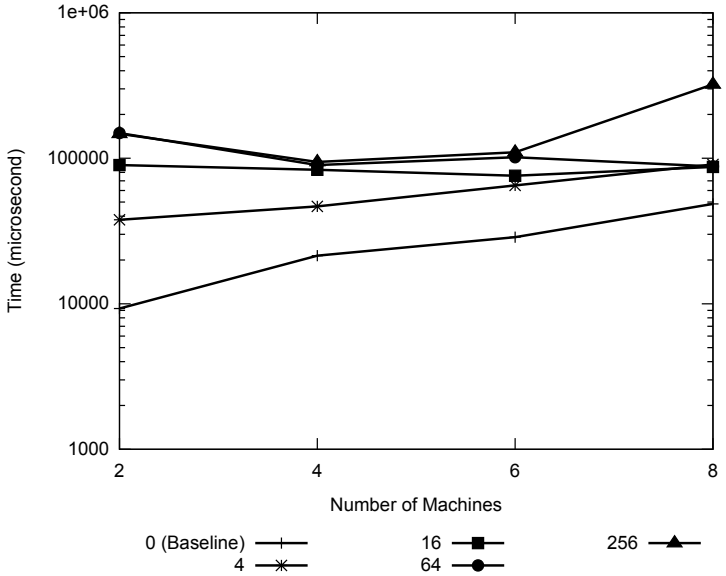
Figure 4.3d depicts the local execution time of transactions. The maximum overhead registered was 257%, a value that is much lower than the 1804% observed in the results of Configuration A used in Bank Benchmark. In MVCC, the longer a transaction is, the more expensive is for it to read, as it has to iterate over newer committed versions to find the version of its snapshot. Also, the longer the transaction, the higher the probability that there were actually other transactions committing during its execution. Besides this, in SPECULA, the writes to non-transactional shared state are logged in undo logs. This represented 15% off the total time of execution in each node (however, using undo logs to restore non-transactional shared state accounted for less than 0.01%), because STMBench7 has a rich application logic. Moreover, although the configuration we evaluated was the so called write-dominated, it has a ratio of read-only transactions over 50%, and read-only transactions incur into a meaning performance penalty in SPECULA, as they their read-sets have to be tracked and validated.



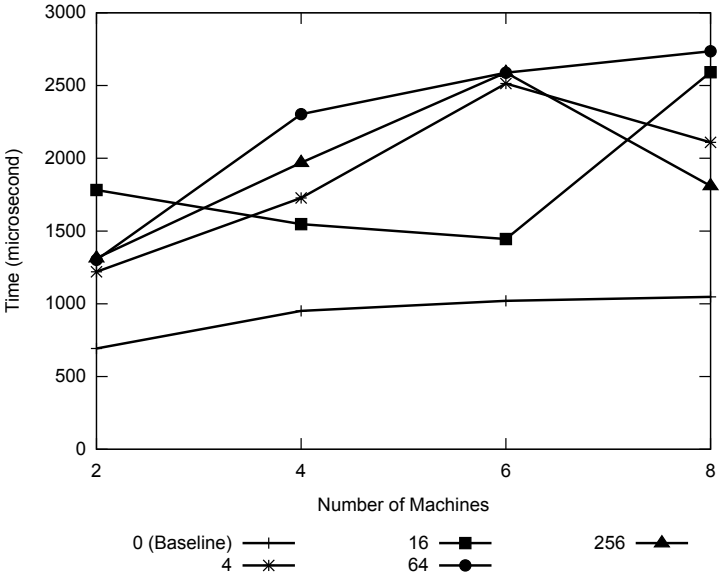
(a) Speedup – varying the value of $MAX\_COMMITTING\_TXS$.



(b) Abort rate – varying the value of $MAX\_COMMITTING\_TXS$.



(c) Total execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.



(d) Local execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4.3: STMBench7 – Write-dominated workload.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	816	2 227	642	715	705	5 733	682	701
Abort Rate (%)	85.049	39.38	99.221	98.741	85.248	7.256	89.15	82.311
Committed Transact.	122	1 350	5	9	104	5 317	74	124
Spec. Transact. Committed	115	1 312	2	5	100	5 294	70	119
Blocked Time (*ms*)	49 844	42 739	50 707	51 070	51 061	34 732	52 157	49 738
Total Committed Transact.	7 105							

(a) $MAX\_COMMITTING\_TXS$ equal to 4.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	993	1 106	917	1 201	874	839	834	11 966
Abort Rate (%)	98.59	96.926	95.856	67.86	99.085	99.166	99.041	1.905
Committed Transact.	14	34	38	386	8	7	8	11 738
Spec. Transact. Committed	9	30	32	381	4	3	5	11 732
Blocked Time (*ms*)	48 542	49 662	51 949	51 075	51 543	53 225	52 865	15 964
Total Committed Transact.	12 233							

(b) $MAX\_COMMITTING\_TXS$ equal to 16.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	840	824	810	854	832	1 056	16 978	1 042
Abort Rate (%)	97.5	99.393	99.753	97.892	99.76	97.822	1.708	99.424
Committed Transact.	21	5	2	18	2	23	16 688	6
Spec. Transact. Committed	17	2	0	14	0	19	16 685	2
Blocked Time (*ms*)	52 254	54 089	52 203	54 248	52 659	52 550	986	52 705
Total Committed Transact.	16 765							

(c) $MAX\_COMMITTING\_TXS$ equal to 64.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	1 002	1 018	1 025	742	977	14 809	1 653	1 000
Abort Rate (%)	98.104	99.804	99.122	97.978	96.418	1.992	62.613	98.2
Committed Transact.	19	2	9	15	35	14 514	618	18
Spec. Transact. Committed	14	0	5	11	31	14 511	614	13
Blocked Time (*ms*)	53 180	54 400	53 260	53 499	53 736	9 605	49 550	54 676
Total Committed Transact.	15 230							

(d) $MAX\_COMMITTING\_TXS$ equal to 256.

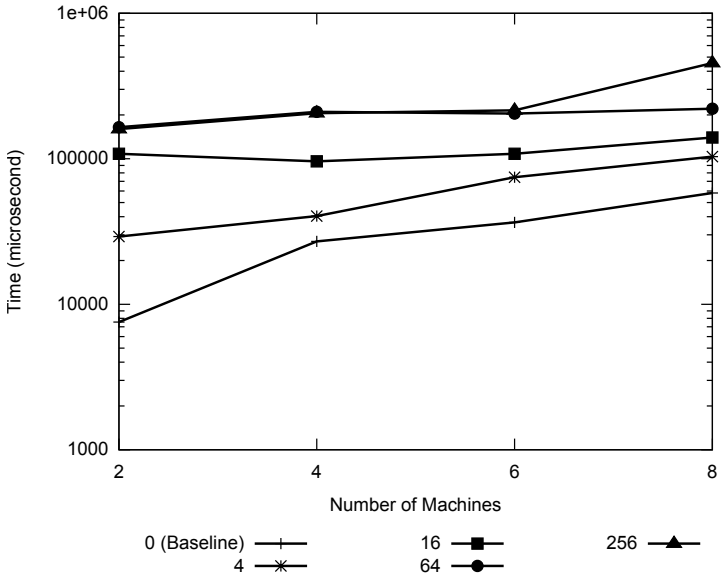
	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	1 983	1 708	1 476	2 127	1 173	405	2 258	2 419
Abort Rate	17.549	27.518	33.266	21.392	45.269	78.025	18.822	17.197
Committed Transact.	1 635	1 238	985	1 672	642	89	1 833	2 003
Total Committed Transact.	10 097							

(e) Baseline system – 1 thread.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	4 376	1 150	797	428	551	290	1 254	2 415
Abort Rate	21.938	84.174	92.472	86.916	82.214	90	76.874	43.188
Committed Transact.	3 416	182	60	56	98	29	290	1 372
Total Committed Transact.	5 503							

(f) Baseline system – 4 threads.

Table 4.5: STMBench7 – Write-dominated workload (values per run).



(e) Network latency – varying the value of $MAX\_COMMITTING\_TXS$.

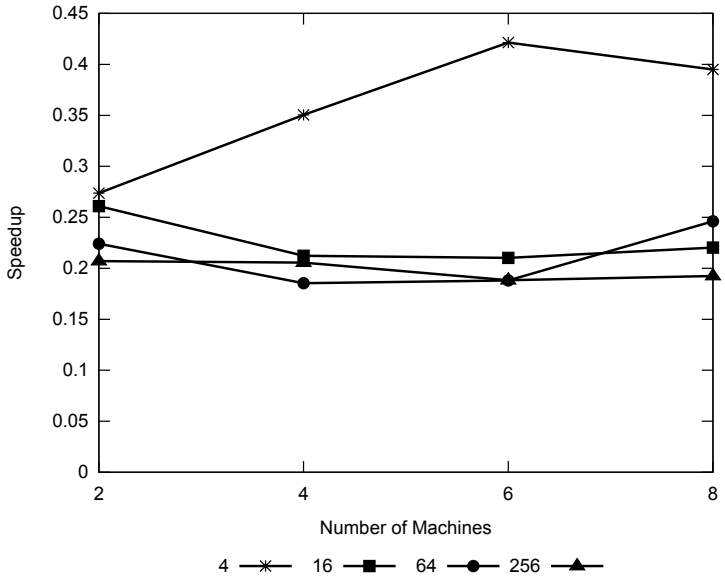
Figure 4.3: STMBench7 – Write-dominated workload.

Figure 4.3e depicts the network latency values that were observed. As STMBench7's transactions are much longer than for instance the ones generated by Bank Benchmark, the GCS had to handle less work, which made it perform more consistently. Nevertheless, latency was still abnormally high for a LAN environment, even for the baseline system.

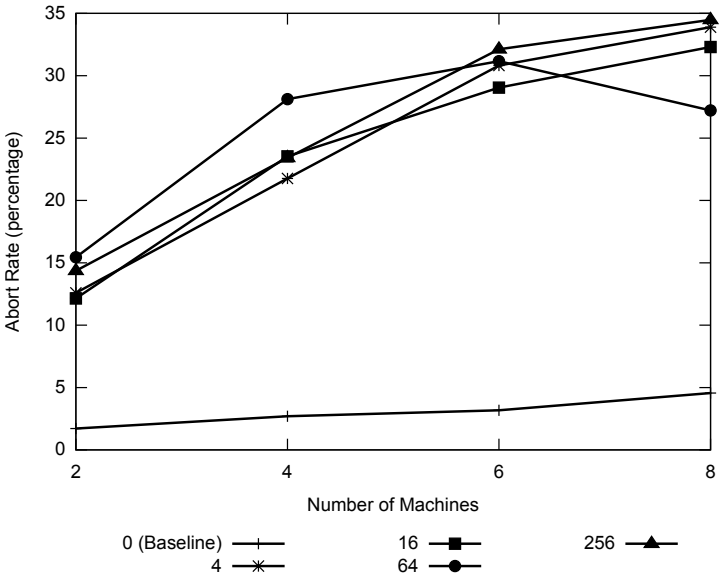
The tables in 4.5 show that even with an abort rate such high, there was a big amount of speculative transactions being successfully committed. This is a very good sign and one that empowers the motivations behind this work. Unfortunately, the same tables also show that worker threads were blocked by the execution control algorithm for most of the run.

With $MAX\_COMMITTING\_TXS = 64$ the system achieved a significant speedup and a consistent response latency, therefore, based on these results it is a adequate value for high contention scenarios with not so short transactions.

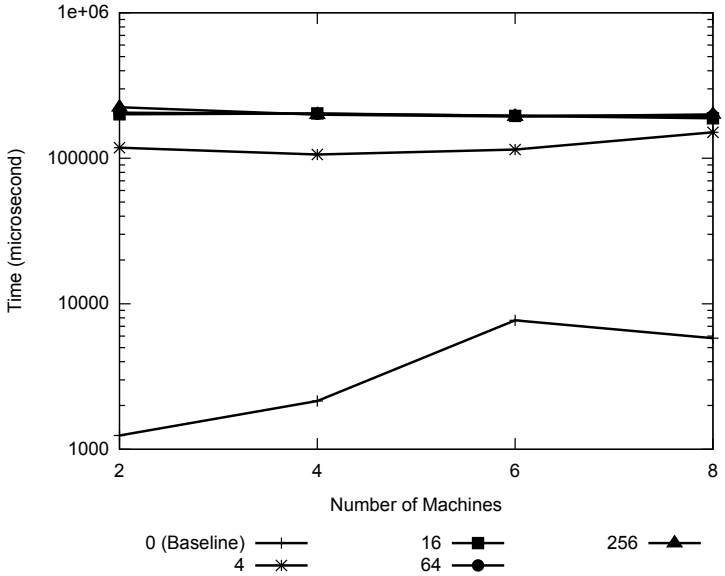
Read-dominated Workload Figure 4.4a depicts the speedup achieved in this configuration. As expected, it is negative, since as already stated, read-only transactions incur into a significant overhead that comes from the need of tracking and validating their read-sets. Furthermore, the application logic in this configuration is even richer than in the write-dominated workload, and so building undo logs accounted for 20% of the whole execution. Using them to restore non-



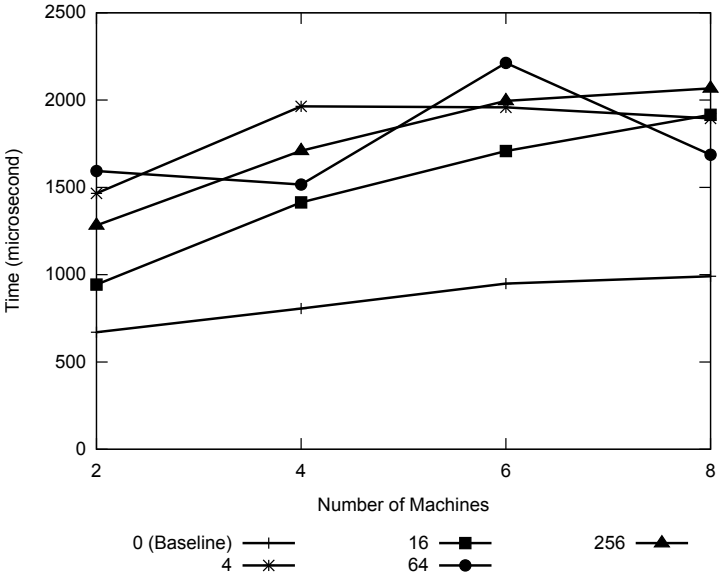
(a) Speedup – varying the value of $MAX\_COMMITTING\_TXS$.



(b) Abort rate – varying the value of $MAX\_COMMITTING\_TXS$.



(c) Total execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.



(d) Local execution time of transactions – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4.4: STMBench7 – Read-dominated workload.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	4 639	10 869	9 745	10 292	3 595	3 290	11 442	3 810
Abort Rate (%)	49.364	31.824	30.251	29.431	50.598	49.696	22.103	48.399
Committed Transact.	2 349	7 410	6 797	7 263	1 776	1 655	8 913	1 966
Spec. Transact. Committed	2 299	7 313	6 712	7 172	1 727	1 619	8 830	1 918
Blocked Time (*ms*)	36 466	28 822	29 731	28 563	39 876	41 274	25 600	40 248
Total Committed Transact.	38 129							

(a) $MAX\_COMMITTING\_TXS$ equal to 4.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	3 421	3 711	6 198	6 501	5 792	1 595	2 219	1 989
Abort Rate (%)	41.333	41.471	28.138	24.258	26.968	33.605	43.894	40.573
Committed Transact.	2 007	2 172	4 454	4 924	4 230	1 059	1 245	1 182
Spec. Transact. Committed	1 969	2 137	4 427	4 895	4 201	1 049	1 227	1 166
Blocked Time (*ms*)	4 5925	4 7441	4 5525	4 1594	4 5324	5 2383	5 0470	5 0590
Total Committed Transact.	21 273							

(b) $MAX\_COMMITTING\_TXS$ equal to 16.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	6 314	2 387	2 802	2 063	1 589	6 236	7 673	3 584
Abort Rate (%)	26.655	41.307	27.73	31.508	50.472	25.289	17.842	29.074
Committed Transact.	4 631	1 401	2 025	1 413	787	4 659	6 304	2 542
Spec. Transact. Committed	4 601	1 380	2 009	1 401	774	4 636	6 276	2 528
Blocked Time (*ms*)	42 348	51 415	49 206	52 620	52 388	44 512	41 080	48 331
Total Committed Transact.	23 762							

(c) $MAX\_COMMITTING\_TXS$ equal to 64.

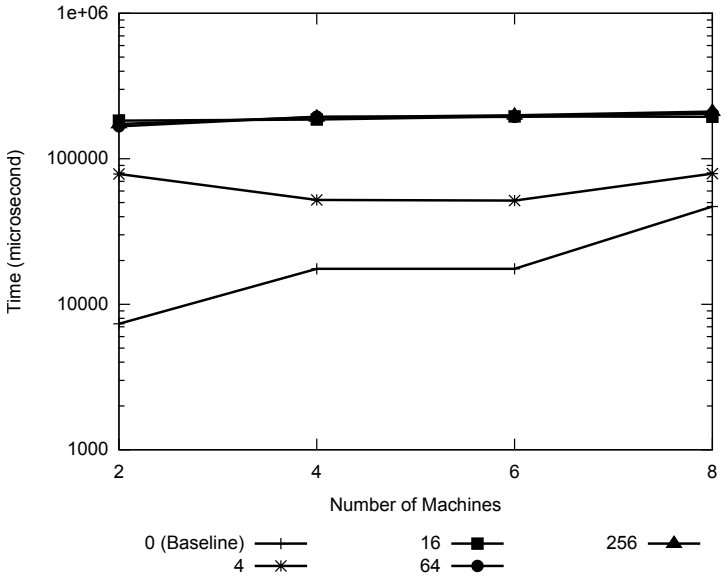
	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	7 661	3 467	2 149	2 141	3 089	3 474	4 648	1 727
Abort Rate (%)	26.041	40.554	33.411	31.854	40.207	30.829	41.007	43.833
Committed Transact.	5 666	2 061	1 431	1 459	1 847	2 403	2 742	970
Spec. Transact. Committed	5 621	2 026	1 419	1 448	1 820	2 379	2 708	955
Blocked Time (*ms*)	36 366	47 149	50 644	53 610	49 262	46 798	47 956	52 732
Total Committed Transact.	18 579							

(d) $MAX\_COMMITTING\_TXS$ equal to 256.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Exec. Transactions	11 582	13 293	8 953	16 612	19 536	4 330	13 375	13 474
Abort Rate	4.274	4.461	6.802	3.732	2.943	11.224	4.344	4.809
Committed Transact.	11 087	12 700	8 344	15 992	18 961	3 844	12 794	12 826
Total Committed Transact.	96 548							

(e) Baseline system.

Table 4.6: STMBench7 – Read-dominated workload (values per run).



(e) Network latency – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4.4: STMBench7 – Read-dominated workload.

transactional shared state accounted for less than 0.01%.

Figure 4.4b depicts the abort rate observed. Again, as expected, it was significantly higher than the one verified in the baseline system, which is totally normal as it ensures that read-only transactions are abort-free and SPECULA does not. For instance, with eight machines and $MAX\_COMMITTING\_TXS = 256$, from all the 1222 aborted transactions, 909 were read-only.

Figure 4.4d depicts the local execution duration of transactions that was observed. The explanation for these is similar to the one present in the write-dominated configuration.

Figure 4.4c depicts the total execution time of transactions. The graphic is however misleading, because, as described in Section 3.9.3, we have to associate read-only transactions with update transactions. Therefore, the average total execution time of read-only transactions is similar to the total execution time of update transactions, although they are executed in a totally uncoordinated fashion.

Figure 4.4e depicts the network latency observed. Since this workload is composed by few update transactions, the network latency represents a minor issue in this case.

The tables in 4.6 show that, unfortunately, all nodes remained blocked by the execution

control algorithm for most of the run. This was not due to the network latency but to the high abort rate, which causes the execution control to slice the value of *specLimit* by half. However, the same tables also show that a high amount of all speculatively executed transactions were final committed.

4.5 Discussion

Results show that SPECULA is suitable even for high contention scenarios. In fact, the unbalance in the abort rate distribution ends up benefiting our approach; in the baseline system the node that features a low abort rate is able to commit less transactions than in SPECULA.

Unfortunately, with very small transactions, the high rate of (speculative) commits quickly saturates the network. In this scenario, the pipelining effect created by SPECULA is constantly halted by the execution control mechanism. Thus, the network is still a bottleneck in the system operation.

Configuration A in Bank Benchmark achieved meaningful speedups with just around 20 seconds of execution in each node. We can further increase the value of $MAX\_COMMITTING\_TXS$ but the results have showed that the system's responsiveness was already poor with $MAX\_COMMITTING\_TXS = 256$. The time required to fully execute a transaction is of course the sum of time needed by all the phases in its critical path, so it is as fast as the slowest link in the chain. We have assumed the network latency to be just one order of magnitude higher than local execution of transactions, for which low contention scenarios like the Configuration A used in Bank Benchmark should have never required the execution control algorithm to act with $MAX\_COMMITTING\_TXS = 256$, namely after the *warm-up* phase. Instead, we verified that network latency was in some cases three orders of magnitude higher than the local execution time of transactions, which has seriously hampered SPECULA.

It is difficult for SPECULA to be competitive in scenarios with mostly read-only transactions, as in the baseline system they are guaranteed to be abort-free, hence their read-sets do not need to be tracked and validated, all properties that SPECULA does not share. However, if we model read-only and update transactions according to the Amdahl's law, as update transactions have to serialized whereas read-only transactions can be freely executed concurrently, we notice that SPECULA opens rooms for larger speedups in highly concurrent systems, as it reduces the

percentage of serial execution.

Nevertheless, SPECULA achieved meaningful speedups even in complex scenarios like the write-dominated configuration of the STMBench7 benchmark. Although we were expecting to achieve better results, the ones we got are definitely encouraging.

Summary

This chapter presented the evaluation of our SPECULA prototype. The results obtained show evidence that the system is capable of deliver boosts in performance in face of very different scenarios. Both a relatively simple micro-benchmark and more complex applications like the STMBench7 benchmark benefited from the optimistic approach introduced with SPECULA. We presented a detailed discussion and analyzed the possible sources of inefficiency in each benchmark. Finally, the chapter concluded with a brief general analysis.

The next chapter presents the conclusions about this work and directions for future work.

5 Conclusions and Future Work

Survivability takes priority.

– Misato Katsuragi, Neon Genesis Evangelion (TV Series)

DRSTMs allows programmers to develop highly concurrent and dependable systems with minimal effort. Unfortunately, the implementation of this powerful abstraction with good performance is still an open challenge. This thesis addressed this problem by analyzing, implementing and evaluating techniques to improve the performance of DRSTMs.

To this end, this thesis presents SPECULA, a novel system that can speculatively execute transactional and non-transactional code in a safe manner, by being able to rollback all changes made to memory if a mis-speculation is detected. SPECULA relies on a certification-based replication scheme to enable the speculative commit of memory transactions. This allows transaction processing to proceed while the global serial order of (speculatively) committed transactions is defined in background. For this purpose, SPECULA manages access to speculatively committed data and performs modifications to the application at the bytecode level that enables it to undo writes made to non-transactional shared state, and also to rollback the execution flow of threads, all in a fully transparent fashion for both the application and the programmer.

Experimental results show that SPECULA achieves significant speedups in low contention scenarios and can be also useful in high contention scenarios. By extending the level of optimism of standard certification-based replication solutions, SPECULA promotes a better use of the computational resources available in the system.

SPECULA does not address the problem of optimizing network usage, a topic that is orthogonal to the focus of this thesis. Therefore, as other certification-based approaches, the costs of inter-replica synchronization can saturate the underlying group communications system, with the resulting penalties in the resulting coordination latency. This fact constrained the system's throughput in low contention scenarios and made its responsiveness significantly lower as the

level of optimism increased.

As future work the system would benefit from better execution control mechanisms, feedback and auto-tuning schemes, in order maximize its throughput without affecting its responsiveness. Static analysis of the application's code can also help to reduce the overhead of building undo logs, by identifying write operations that never need to be undone. A reevaluation of the system in a low network latency environment should also be performed to further validate this work. To this end, the performance of the system should be assessed with various AB protocols (e.g., token based) and GCSs (e.g., JGroups).

Bibliography

Allman, M., V. Paxson, & W. Stevens (1999). RFC 2581: TCP congestion control.

Ananian, C. S., K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, & S. Lie (2005, February). Unbounded transactional memory. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture*, HPCA'05, San Francisco, CA, USA, pp. 316–327. IEEE.

Bernstein, P. A., V. Hadzilacos, & N. Goodman (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Birman, K. P. & T. A. Joseph (1987, January). Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*(1), 47–76.

Bloom, B. H. (1970, jul). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*(7), 422–426.

Bobba, J., N. Goyal, M. D. Hill, M. M. Swift, & D. A. Wood (2008, June). Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*, ISCA'08, Beijing, China, pp. 127–138. IEEE.

Bruneton, E., R. Lenglet, & T. Coupaye (2002, November). ASM: A code manipulation tool to implement adaptable systems. In *In Proceedings of Adaptable and Extensible Component Systems*, Grenoble, France.

Cachin, C., R. Guerraoui, & L. Rodrigues (2011). *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer.

Cachopo, J. (2007). *Development of Rich Domain Models with Atomic Actions*. Ph. D. thesis, Technical University of Lisbon.

Cachopo, J. P. & A. R. Silva (2006, December). Versioned boxes as the basis for memory transactions. *Sci. Comput. Program. 63*(2), 172–185.

Carvalho, N., P. Romano, & L. Rodrigues (2011, May). Scert: Speculative certification in replicated software transactional memories. In *Proceedings of the 4th Annual International Systems and Storage Conference*, SYSTOR'11, Haifa, Israel, pp. 10. ACM.

Chockler, G., I. Keidar, & R. Vitenberg (2001, December). Group communication specifications: a comprehensive study. *ACM Comput. Surv. 33*(4), 427–469.

Couceiro, M., P. Romano, N. Carvalho, & L. Rodrigues (2009, November). D2stm: Dependable distributed software transactional memory. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC'09, Shanghai, China, pp. 307–313. IEEE.

Coulouris, G., J. Dollimore, & T. Kindberg (2002). *Distributed systems - concepts and designs (3. ed.)*. International computer science series. Addison-Wesley-Longman.

Damron, P., A. Fedorova, Y. Lev, V. Luchangco, M. Moir, & D. Nussbaum (2006, October). Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'06, San Jose, CA, USA, pp. 336–346. ACM.

Défago, X., A. Schiper, & P. Urbán (2004, December). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*(4), 372–421.

Dice, D., O. Shalev, & N. Shavit (2006, September). Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC'06, Stockholm, Sweden, pp. 194–208. Springer.

Dice, D. & N. Shavit (2007, March). Understanding tradeoffs in software transactional memory. In *Proceedings of the 5th International Symposium on Code Generation and Optimization*, CGO'07, San Jose, CA, USA, pp. 21–33. IEEE.

Ennals, R. (2006, Jan). Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report.

Fernandes, J., N. Carvalho, P. Romano, & L. Rodrigues (2011). SPECULA: um protocolo de replicação preditiva para memória transaccional por software distribuída. In *Actas do Terceiro Simpósio de Informática*, INForum'11, Coimbra, Portugal.

Fernandes, S. M. & J. P. Cachopo (2011, February). Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM SIGPLAN Symposium on*

Principles and Practice of Parallel Programming, PPOPP'11, San Antonio, TX, USA, pp. 179–188. ACM.

Guerraoui, R. & M. Kapalka (2008a, June). On obstruction-free transactions. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'08, Munich, Germany, pp. 304–313. ACM.

Guerraoui, R. & M. Kapalka (2008b, February). On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'08, Salt Lake City, UT, USA, pp. 175–184. ACM.

Guerraoui, R. & M. Kapalka (2009, January). The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'09, Savannah, GA, USA, pp. 404–415. ACM.

Guerraoui, R., M. Kapalka, & J. Vitek (2007, March). Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2007 EuroSys Conference*, EuroSys'07, Lisbon, Portugal, pp. 315–324. ACM.

Hammond, L., V. Wong, M. K. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, & K. Olukotun (2004, June). Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, ISCA'04, Munich, Germany, pp. 102–113. IEEE.

Härder, T. & A. Reuter (1983, December). Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*(4), 287–317.

Harris, T., S. Marlow, S. L. P. Jones, & M. Herlihy (2005, June). Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'05, Chicago, IL, USA, pp. 48–60. ACM.

Harris, T. L. & K. Fraser (2003, October). Language support for lightweight transactions. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA'03, Anaheim, CA, USA, pp. 388–402. ACM.

Haynes, C. T., D. P. Friedman, & M. Wand (1984, August). Continuations and coroutines. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, Austin,

TX, USA, pp. 293–298. ACM.

Herlihy, M., V. Luchangco, & M. Moir (2006, October). A flexible framework for implementing software transactional memory. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'06, Portland, OR, USA, pp. 253–262. ACM.

Herlihy, M. & J. E. B. Moss (1993, May). Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA'93, San Diego, CA, USA, pp. 289–300. IEEE.

Herlihy, M. & J. M. Wing (1990, July). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*(3), 463–492.

Jr., R. L. B., V. S. Adve, & B. L. Chamberlain (2008, February). Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'08, Salt Lake City, UT, USA, pp. 247–258. ACM.

Kemme, B., F. Pedone, G. Alonso, A. Schiper, & M. Wiesmann (2003, July). Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng. 15*(4), 1018–1032.

Kotselidis, C., M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, & I. Watson (2008, September). Distm: A software transactional memory framework for clusters. In *Proceedings of the 2008 International Conference on Parallel Processing*, ICPP'08, Portland, OR, USA, pp. 51–58. IEEE.

Kumar, S., M. Chu, C. J. Hughes, P. Kundu, & A. D. Nguyen (2006, March). Hybrid transactional memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'06, New York, NY, USA, pp. 209–220. ACM.

Lamport, L. (1978, July). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*(7), 558–565.

Laurent Baduel, Francoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, & Romain Quilici (2006). Programming, Composing, Deploying for the Grid. In Jose C.; Rana Omer F. Cunha (Ed.), *Grid Computing: Software Environments and Tools*, pp. 205–229. Springer.

Manassiev, K., M. Mihailescu, & C. Amza (2006, March). Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'06, New York, NY, USA, pp. 198–208. ACM.

Marathe, V. J., W. N. S. Iii, & M. L. Scott (2004, October). Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, LCR'04, Houston, TX, USA, pp. 1–7. ACM.

Marathe, V. J., W. N. S. III, & M. L. Scott (2005, September). Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, DISC'05, Cracow, Poland, pp. 354–368. Springer.

Miranda, H., A. S. Pinto, & L. Rodrigues (2001, May). Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, ICDCS'01, Phoenix, AZ, USA, pp. 707–710. IEEE.

Palmieri, R., F. Quaglia, & P. Romano (2010, July). Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications*, NCA'10, Cambridge, MA, USA, pp. 20–27. IEEE.

Palmieri, R., F. Quaglia, P. Romano, & N. Carvalho (2010, April). Evaluating database-oriented replication schemes in software transactional memory systems. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'10, Atlanta, GA, USA, pp. 1–8. IEEE.

Papadimitriou, C. H. (1979, October). The serializability of concurrent database updates. *J. ACM 26*(4), 631–653.

Pedone, F., R. Guerraoui, & A. Schiper (2003, July). The database state machine approach. *Distributed and Parallel Databases 14*(1), 71–98.

Perelman, D., R. Fan, & I. Keidar (2010, July). On maintaining multiple versions in stm. In *Proceedings of the 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'10, Zurich, Switzerland, pp. 16–25. ACM.

Powell, D. (1996, April). Group communication (introduction to the special section). *Com-*

mun. ACM 39(4), 50–53.

Romano, P., N. Carvalho, & L. Rodrigues (2008). Towards distributed software transactional memory systems. In *Proceedings of the 2nd ACM Workshop on Large-Scale Distributed Systems and Middleware*, LADIS'08, Watson Research Labs, Yorktown Heights, NY, USA. ACM. (invited paper).

Saha, B., A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, & B. Hertzberg (2006, March). Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'06, New York, NY, USA, pp. 187–197. ACM.

Schneider, F. B. (1990, December). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*(4), 299–319.

Shavit, N. & D. Touitou (1995, August). Software transactional memory. In *Proceedings of the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'95, Ontario, Canada, pp. 204–213. ACM.

Tabba, F., M. Moir, J. R. Goodman, A. W. Hay, & C. Wang (2009, August). Nztm: nonblocking zero-indirection transactional memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'09, Calgary, Canada, pp. 204–213. ACM.

Tomic, S., C. Perfumo, C. E. Kulkarni, A. Armejach, A. Cristal, O. S. Unsal, T. Harris, & M. Valero (2009, December). Eazyhtm: eager-lazy hardware transactional memory. In *Proceedings of the 42st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'09, New York, NY, USA, pp. 145–155. ACM.