Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XXIV Ciclo – 2011

Speculative Protocols for Actively Replicated Transactional Systems

Roberto Palmieri

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XXIV Ciclo - 2011

Roberto Palmieri

Speculative Protocols for Actively Replicated Transactional Systems

Thesis Committee	**Reviewers**
Prof. Francesco Quaglia (Advisor)	Prof. Jennifer Lundelius Welch
Prof. Maurizio Lenzerini	Prof. Rachid Guerraoui

AUTHOR'S ADDRESS:
Roberto Palmieri
Dipartimento di Informatica e Sistemistica
Sapienza Università di Roma
Via Ariosto 25, I-00185 Roma, Italy
E-MAIL: `palmieri@dis.uniroma1.it`
WWW: `http://www.dis.uniroma1.it/~palmieri/`

To my mother and father

Acknowledgments

My first special thanks goes to my colleague and friend Pierangelo. The help and advices that he provided during my master thesis were invaluable. Thanks to his effort I started soon my Ph.D.; without his support, I would never have been able to publish my first international scientific article. Without his way of being everyday, these three years together in room B120 would have been so different.

There are not appropriate words in any language of the world for expressing my gratitude for my advisor prof. Francesco Quaglia. Francesco has been able to lead me in these three years, trying to solve together with me all the problems that manifested during my doctorate. He taught me the approach to scientific research. He taught me that the quality of the work is the only target that must be addressed. He is a wonderful person, a confidant, a friend and last but not least, a big AS Roma supporter. I hope, sooner or later, to give back to Francesco all the effort that he spent to help me pursuing my Ph.D.

Together with Francesco, a special thanks goes to my (non-formal) co-advisor, Paolo Romano. I met him seven years ago during my first level degree, and for at least five years he has been a reference point for his way to approach and solve the problems (scientific and not). All the works and the quality of this thesis would not be the same without his contribution.

I owe a debt of gratitude also to the leader of our research group, who is more than just a great scientist, but a father for all the group members and in particular for me. He transmitted to me the enthusiasm for the teaching activities but more important, he taught me to be a good person and to never compromise myself for any reason. Thank you Bruno.

I am also grateful to prof. Jennifer Lundelius Welch and prof. Rachid Guerraoui for having accepted to serve as external referees of this dissertation, and to prof. Maurizio Lenzerini for the support while the evolution and finalization of this thesis.

A special thanks to the “young” researchers who have been close to me in any situation (Roberto Vitali, Alessandro Pellegrini, Sebastiano Peluso and Diego Rughetti).

To my family (Nadia, Luciano and Marco), the only reason that made it possible. It is really impossible to think that their entire life has been dedicated to me and my brother. This thesis and the person I have become, have been possible only thanks to you!

Thanks to all the persons that have been close to me in these years.

Finally, a special thanks to a special person. Half of my thesis has been written in her house.
Thanks for the support and to make me happy...everyday...Marina.

Roberto Palmieri

Abstract

Nowadays, the role of transactions has become twofold:

- they are used in order to guarantee consistency and atomicity in applications manipulating data;
- they are used as a means to synchronize the activities of threads working concurrently within any software layer.

Overall, the concept of transaction, historically related to support data manipulation in the context of database systems, has been widened so to encapsulate synchronization aspects in the context of parallel and concurrent applications. The latter aspect found its expression via Software Transactional Memory (STM) technologies, which have been oriented to mask the complexity of synchronization to the application programmers, thus moving along the path of bringing the power of multi/many-core architectures into the hand of ordinary, non-specialized, software developers. Such a widened scope of transactions, together with significant technological innovations possibly impacting the execution profile/cost of traditional database transactions (e.g. the advent of SSD storage systems) and the level of transaction parallelism (e.g. the advent of many-core architectures), raise the need for reconsidering the design of protocols supporting fault tolerance.

In this thesis, I focus on fault tolerant protocols based on the active replication paradigm, which is done by systematically exploiting speculative computation approaches. More in detail, I worked on innovative speculative transactional replication protocols relying on Optimistic Atomic Broadcast group communication primitives, which have been used as a building block for replicas coordination. Some proposed results are mostly oriented to theory, while others have a more strict relation with pragmatic aspects associated with the design/implementation of replicated transactional systems.

Most of the material presented in this document can also be found in the following technical articles:

1. Roberto Palmieri, F. Quaglia and P. Romano
   OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems
   proc. 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011), Madrid, Spain, October 2011.

2. P. Romano, Roberto Palmieri, F. Quaglia, N. Carvalho and L. Rodrigues
   An Optimal Speculative Transactional Replication Protocol
   Proc. 8th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), Taiwan, Taipei, IEEE Computer Society Press, September 2010.

3. Roberto Palmieri, P. Romano, F. Quaglia
   AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing
   Proc. 9th IEEE International Symposium on Network Computing and Applications (NCA), Cambridge, Massachussets, USA, IEEE Computer Society Press, July 2010.

4. P. Romano, Roberto Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues
   On Speculative Replication of Transactional Systems (Brief Annuncement)
   Proc. 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Santorini, Greece, ACM Press, June 2010.

5. Roberto Palmieri, F. Quaglia, P. Romano and N. Carvalho
   Evaluating Database-oriented Replication Schemes in Software Transactional Memory Systems
   Proc. 15th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS), Atlanta, USA, IEEE Computer Society Press, April 2010.

6. P. Di Sanzo, B. Ciciani, F. Quaglia, Roberto Palmieri and P. Romano
   On the Analytical Modeling of Concurrency Control Algorithms for Software Transactional Memories: the Case of Commit-Time-Locking
   Elsevier Performance Evaluation Journal.

7. Paolo Romano, Roberto Palmieri, Francesco Quaglia, Nuno Carvalho and Luis Rodrigues
   On Speculative Replication of Transactional Systems
   Journal of Computer and System Sciences, pending revision.

Contents

Chapter 1

Introduction

Nowadays, more and more companies in the world rely on computing systems as a base of their productivity. For some of them, the core business consists in directly offering services to the users by exploiting the internet communication channel, which allows reaching a large number of customers (clients) anywhere in the world in a relatively simple and fast manner. Some other companies exploit computing systems to support their internal activities, without a direct influence on their finance, such as when decision support systems or work-flow management systems are adopted. In either case, the global productivity of the company takes advantages from the increasing level of quality of such systems. Further, the nature of the offered services or products is continuously evolving both thanks to the possibilities offered by emerging technologies and also due to the need for competing within the market. Thus, the complexity of the services hosted by modern computing infrastructures shows the trend of a constant increase.

In such a context, the only viable way to come out and to appeal an increasing number of users is to take quality aspects of the computing system into account at design time. Among them we find all the parameters belonging to the user-perception of a service, which globally impact the so called *abandon rate*, namely a measure for the phenomenon where users abandon their interactions with specific sites (systems) in favor of other providers just due to some quality lacks (e.g. excessively long response time, non-negligible frequency of system fault and reduced availability). The parameters to which reference was made up have also been grouped within a more general classification called Quality-of-Service (QoS). The macro-area of QoS is therefore an expression for all those parameters that describe the system goodness.

Due to the growing importance of the offered products and services, and of their impact on business, there are several key motivations to provide computing systems able to reach prefixed levels of quality. This expectation has

found its expression via the notion of Service Level Agreement (SLA), where the provider (or system's owner) engages the obligation to guarantee pre-negotiated quality levels. Anyway, even without any subscripted contract, high quality of the services offered by the supplier tends to grow up the fidelity of the users and, consequently, their appreciation.
As for the literature, the following three parameters are considered among the most relevant for QoS:

– availability: used to characterize the readiness and the continuity of the system;

– fault tolerance: used to characterize the capability of the system to ensure correct processing in the presence of faults;

– performance: used to characterize system latency and throughput.

1.1 QoS Historical Perspective

Historically, QoS has become fundamental for Content Delivery Networks (or Content Distribution Networks) [4, 44], that have been for a long time the target of several research communities' studies. This is because, at the beginning of the Web, a great part of its contents was represented by static objects. Hence, the most effective means employed to disseminate such information was based on caching Web contents over apposite and possibly cooperative hosts disseminated over the Internet, exactly referred to as CDNs.
In the last few years, the distributed platforms supporting CDNs have undergone a large growth so, currently, there is a number of commercial infrastructures, such as those owned by Akamai and Edgix, which are made up by even more than 10.000 nodes distributed on a planetary scale.
When the number of nodes in the system increases, centralized approaches for supporting services could fall short of scalability and, consequently, the technology has further moved to distributed approaches, based on the so called Edge Computing [32] paradigm. As its name expresses, Edge Computing pushes applications, data and computing power (services) away from centralized points towards the logical extremes of a network. Edge Computing replicates fragments of information across distributed servers, that may be vast and span over many networks. To ensure adequate quality levels of widely-dispersed distributed services, large organizations typically implement Edge Computing by deploying server farms with clustering.
When transactions are included as a support for applications deployed on an ECP (Edge Computing Platform), like more recently happened for, e.g., e-commerce applications, there are many solutions to improve service levels via the exploitation of caching methodologies [35, 18]. Specifically, ECP

servers can use a caching repository to promptly access and return any information requested by clients. This yield an improvement in response time and system availability, however limitedly to the case of requests entailing read-only transactions. In fact, when the requests entail transactional data updates, with workloads exhibiting write-intensive profiles or phases, traditional caching strategies cannot be successfully applied, and other strategies, e.g. synchronized (or coordinated) replication schemes, are required in order to assure the coherence among replicated data across the nodes within the system.
In the latter scenario, the system needs to guarantee an additional property called data consistency. Briefly, if in a transactional system both read and write operations are performed on the same data entry, and the write precedes the read, the latter must return the correct, updated value, independently of whether there are cache servers collocated on the edges of the network. Overall, in order to correctly support transactional data manipulations, while still taking advantage from replication, the system must implement some (efficient) coordination protocol exactly aimed at guaranteeing the aforementioned consistency property.

1.2 Hints on Replication Approaches

As pointed out, replication is the process of sharing information. It could be used for improving systems performance, e.g., like with caching mechanisms in the presence of read-only workloads. However, it could be used as a form of consistency guarantee among redundant resources, such as software or hardware components, in order to improve reliability, fault-tolerance, or accessibility. There can be data replication, if the same data is stored on multiple storage devices, or computation replication, if the same computing task is executed multiple times. Also, a task is typically replicated in space, i.e. executed on separate devices, but it could be replicated even in time, if it is executed repeatedly on a single device.

When focusing on computing systems offering services, and considering the presence of (transactional) update requests, the data consistency issue can be tackled via differentiated replication approaches whose typical classification [83] is shown in Figure 1.1. As a very high level of abstraction, it is common to talk about two fundamental classes of replication techniques: *active* and *passive*. In passive replication there is a server (called primary) that receives and processes the client requests. Subsequently, the primary updates the state of the other (backup) servers and sends back the response to the client. If the primary server fails, one of the backup servers takes its place. For the case of a single primary server, the scheme is called primary-backup. Otherwise,

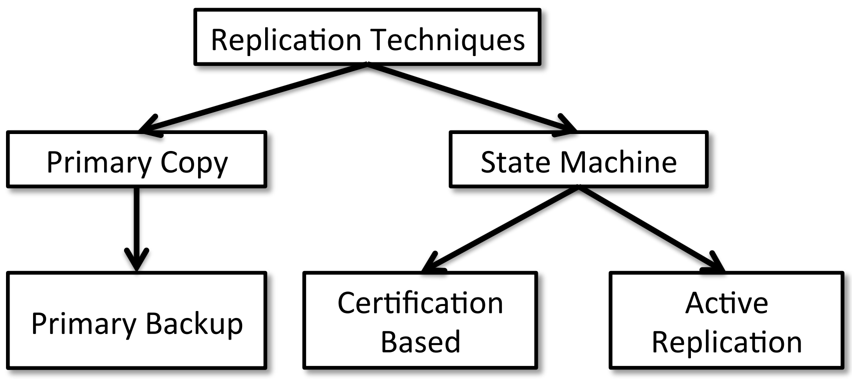


Figure 1.1: Classification of Replication Techniques.

in presence of a group of primary servers, it is instead referred to as multi-primary.
Communication between the primary and the backups has to guarantee that updates are received and then processed in the same order, which is the case if primary backup communication is based on FIFO channels. However, FIFO channels are not enough to ensure correct execution in case of failure of the primary. For example, consider the case where the primary fails before all the backups have received the updates for a certain request, and another replica takes over as a new primary. Some mechanism has to ensure that updates sent by the new primary will be properly ordered as occurring after the updates sent by the faulty primary.
View Synchronous Broadcast (VSCAST) is a mechanism that guarantees these constraints, and can therefore be used to implement the primary backup replication technique [83]. Briefly, it is based on the notion of a sequence of views of a group. Each view defines the composition of the group at any time, i.e. the members of the group that are perceived as being correct at a certain time. Whenever a process in some view is suspected to have crashed, or some process wants to join, a new view is installed, which reflects the membership change. Roughly speaking, VSCAST of message m by some member of the group g currently belonging to view $v_i(g)$ ensures the following property:

– if one process p in $v_i(g)$ delivers m before installing view $v_{i+1}(g)$, then no process installs view $v_{i+1}(g)$ before having delivered m.

Passive replication can tolerate non-deterministic servers (e.g., multi-threaded servers) and uses reduced processing power when compared to other replication techniques. However, passive replication may suffer from high reconfiguration overhead when the primary fails and may exhibit scalability problems vs the volume of clients, since processing is carried out by a single server (at least in the primary backup scheme).

Active replication, also called the state machine approach [63], is a non-centralized replication technique. Its key concept is that all the replicas receive and process the same sequence of client requests. Consistency is guaranteed by assuming that the servers deterministically process the requests (a.k.a. messages) in the same order. Determinism means that, given the same initial state and the same sequence of messages, all processes will produce the same sequence of replies and end up in the same final state. Clients do not contact one particular server, but the whole set of servers as a single group. The main advantages of active replication are related to the relatively simple way for its implementation (e.g., same code/logic everywhere) and to its failure masking capabilities. Failures are in fact fully hidden from the clients (also in terms of perceived latency), since if a replica fails, the requests are still processed by the other replicas. To guarantee determinism, active replication techniques require at least the following two building-blocks:

– a distributed coordination protocol guaranteeing a deterministic delivery order for all the messages among the members of the system;

– a local deterministic concurrency control within each replica that processes the messages such in a way to respect a defined serialization order.

A typical group communication primitive used to support active replication is Atomic Broadcast [75, 1] (AB), that is a broadcast messaging protocol which ensures that messages are received reliably and in the same order by all the replicas in the system. AB enables the sending of messages to a group of processes, with the guarantee that processes agree on the set of messages to be delivered, and on their delivering order.

A number of AB protocols have been proposed in literature (see, e.g., [30, 29, 28, 3]), under various assumptions about the network, failure models, availability of hardware supports for multicast, and so forth. However, in order to ensure especially the agreement and total order properties, these protocols prove expensive in terms of both messages exchange and delivery latency [33, 25, 34, 79]. This is essentially due to the fact that they address the issue of consensus [78, 57, 7], which possibly requires, multiple rounds in order for the agreement to be reached among all the involved replicas.

For specific system settings, those delays introduced by the coordination phase could be hard to tolerate. For this reason, an improvement of Atomic Broadcast called Optimistic Atomic Broadcast (OAB) [62, 48] was introduced. OAB relies on an early event, termed optimistic delivery, that notifies to the replicas the existence of a new message (request) to be process, additionally guessing its final (non-optimistic) delivery order. Thanks to the optimistic delivery, some active replication protocols try to overwork time between deliveries to make use of idle resources to pre-process messages before they are finally delivered (and hence finally ordered). A reconciliation phase is needed to avoid inconsistencies when the message total order gets notified.

1.3 The Need for Reconsidering Replication Management

At the time most of the aforementioned solutions were proposed, the technological features associated with computing systems and infrastructures were significantly different from those related to the actual trends. As a consequence, in literature there exist several solutions tailored to (or shown as suited for) settings that do not necessarily match the current ones. Reassessing those protocols in the context of modern architectures is the first step ahead for understanding how, and if, those solutions still fit last generation execution environments. Also, proper features offered by the current technology could even open to the designers the possibility to create innovative solutions that were unfeasible so far, or to optimize existing solutions in order to exploit the new possibilities.

In terms of architectural trends, the multi-core paradigm is surely the breaking innovation of the last years. From the recent desktop processors equipped with two processing cores, in a relatively short time, off-the-shelf architectures accessible at non-prohibitive costs have moved to massive multi-cores, entailing 12 or 24 cores per CPU. The key factor contributing to this paradigm shift is the slowdown of Moore's Law. Over the past 40 years, the number of transistors on a microchip doubled every 18 to 24 months [11]. Increasing the number of transistors in a constant area generally yields proportional increase in a single processor's clock-speed and, therefore, in its computation speed. But as the industry approaches the physical limits for the transistor size, the demand for continued improvements in computation speed is driving hardware manufacturers to produce multi-core processors, with multiple processing units on a single chip.

As a reflection, while in the past the exploitation of the architectural properties of a parallel machine was confined to a narrow field of programming experts, the shift of the hardware industry towards (massively) parallel computing architectures imposes that a wider set of programmers and algorithm developers are mandatorily requested to take the potential offered by these computer architectures into account. This is true also for the case of transactional systems at the base of modern services and applications, which means that existing transactional replication protocols would need to be demonstrated to be able (or not to be able) to effectively cope with the case of largely scaled up hardware parallelism, which potentially offers the possibility to support scaled up levels of parallelism while processing transactions.

Spanning on the recent technological innovations, another one potentially disruptive is related to the evolution of storage components. In particular, we have moved from devices based on magnetic disk towards innovative ones, which are based on Solid State technology. The advent of the NAND-flash based solid-state storage device (SSD) certainly represents a sea change in the architecture of computer storage subsystems. These devices are capable of delivering not only large bandwidth, but also random I/O performance that is orders of magnitude better than that of traditional rotating disks. Moreover, SSDs offer both a significant savings in power budget and an absence of moving parts, thus leading to an improvement on the side of reliability. Although the costs of solid-state disks are significantly higher than their rotating counterparts, there are numerous applications where they can be employed with great benefits. For example, in transactional systems, many rotating disks are deployed to increase I/O parallelism. SSDs, suitably optimized for random read and write performance, could effectively replace whole farms of (slow) rotating disks. In addition, the cost of SSD technology continues to decrease, so that the scope for its employment will certainly continue to grow. Moreover, by exploiting the reduced latency to read/write data from/to SSDs, the performance of I/O intensive applications can be revolutionary improved [2]. Focusing the discussion on algorithms for replicated transactional systems, the SSD-based storage system would knock down the latency for executing transactions with the consequence of determining a significant change of the ratio between group communication (i.e. coordination) latency and transactions granularity. The latter phenomenon could lead to a reduction of the effectiveness of the proposed solutions (even if they already rely on optimized group communication schemes such as OAB), in terms of both client perceived latency and resource utilization.

Finally, from the software-architecture point of view, the need to develop applications with the aim at parallelizing the tasks, which would fit the multi-core paradigm, has induced the rising of innovative programming paradigms, such as the Software Transactional Memory (STM) one. The main benefit from STM libraries is towards synchronization transparency in concurrent applications. In fact, leveraging on the proven concept of atomic and isolated transactions, STMs spare programmers from the pitfalls of conventional manual lock-based synchronization, significantly simplifying the development of parallel and concurrent applications. When using STMs, the programmer is not required to deal explicitly with concurrency control mechanisms. Instead, he has only to identify the sequence of instructions, or transactions, that need to access and modify shared objects atomically. As a result, code reliability increases and the software development time gets shortened. This paradigm has grown with the widespread of multi-core architecture, simply because these architectures can naturally support an increased level of application parallelism.

As a matter of fact, the use of STMs has changed (broadened) the nature of transactional operations, which are no more seen as a means to exclusively manipulate application data in a consistent manner, but also as a means to synchronize worker threads acting within any software layer. Additionally, they have induced a shift in terms of the properties to be guaranteed and how these guarantees should be conveniently supported. According to the historical perspective, transactional data manipulation supports were required to enforce the so called ACID (Atomicity, Consistency, Isolation, Durability) properties. On the other hand, durability via the logging of transaction updates onto stable storage systems, like it occurs in traditional DBMS architectures, may become suboptimal in STM-based environments, and could be instead demanded to replication strategies exclusively acting in main memory. This shows how the execution profile of STM transactions can significantly deviate from the one characterizing database transactions, hence again inducing a change of the relative cost of coordination (vs computation) when considering a replicated architecture. Also, new properties, such as *opacity* [39], have become of interest given that STM layers may give rise to scenarios where transactional threads are not executed within a sand-box, hence again showing a potential impact of the execution profile of STM transactions wrt their database counterpart. As a consequence, there would be the need for reshuffling the design of transactional replication protocols in order to make them able to cope with such a change of the execution profile.

1.4 Outline of Innovative Contributions

In the light of the aforementioned technological trends, the aim of this dissertation is to reconsider replication mechanisms for transactional system in order to devise innovative solutions able to effectively cope with the change of the relative cost of computation vs communication, and to exploit hardware parallelism.

Our starting point is represented by recent achievements in the field, which can be identified as the set of active replication protocols where coordination relies on the OAB service. On the basis of comparative studies provided in literature (see, e.g., [81]), these have revealed as highly promising.

On the other hand, the innovation by this dissertation will be represented by the reshuffle of OAB based transactional replication mechanisms in order to systematically exploit the notion of speculative computation. We will address the design of speculative transactional replication schemes by providing both theoretical results and solutions more closely related to system design and implementation aspects. Along the presentation path, particular attention has been posed on discussing how the proposed solutions would fit emerging systems setting, e.g., in terms of the ratio between communication and computation granularity, and the level of parallelism offered by the computing platform. Finally, the spectrum of results we provide copes with differentiated levels of reliability for the OAB service, again representative of scenarios where the OAB layer operates on top of (overlay) networks with different levels of message latency predictability. The latter aspect will be also considered in relation to the actual level of transaction concurrency natively associated with the changes of the volume of requests over time.

Chapter 2

State of the Art

Replication techniques have been widely studied in the scientific literature. In the beginning, replication was essentially seen as a way to provide system availability, thus neglecting potential advantages that could be offered by replicated architectures in terms of performance [83]. On the other hand, when considering performance as a joint target wrt fault-tolerance and availability, proper trade-offs between consistency and efficiency need to be devised.
Upon the growth of the interest in replication techniques, most commercial [45, 22] and non-commercial [83] products were oriented to support replication schemes essentially prone to reduced coordination overhead, namely *asynchronous* replication (a.k.a. *lazy update model*). This type of replication can cause inconsistencies among the replicas, since remote members within the system are updated only after a request has been locally processed and the response was already issued to the client. Clearly, the advantages of this type of replication are limited regarding fault-tolerance, while they can be significant on the side of availability and performance (given the presence of multiple processes acting in parallel on different sets of client requests).

A step ahead along the path of increasing replication robustness wrt fault-tolerance has been done via the introduction of the *synchronous* replication model (a.k.a. *eager update model*) [63]. This model drops any possibility to fall in an inconsistent state on any replica. On the other hand, designing synchronous solutions providing adequate performance is far from being trivial [37].

Another aspect of interest in the design of replication protocols is related to the type of faults they should cope with. On this side, the traditional classification entails *Byzantine* faults as well as *Crashes* [27]. Each of them requires proper mechanisms, which may in some cases be orthogonal to each other. In this dissertation, our focus is on the crash-failure model, thus the reminder of this chapter is essentially focused on the presentation of the more relevant details related to replication protocols tailored for coping with process crashes. Also, special attention is devoted to protocols oriented to transactional systems replication. For the case of Byzantine faults, examples of replication protocols can be found in [26, 52, 47] and in works therein referenced.

The reported overview relies on a classification of the replication protocols based on a common view point related to whether an individual request is, or is not, allowed to be processed in parallel across the replicas upon its arising [38]. As a last preliminary note, we essentially overview protocols adhering to the synchronous replication model, which is the one our solutions have been targeted to. As already hinted they represent a complex case to be addressed in order to provide adequate performance while guaranteeing important property of replicated-state consistency.

In the end of this chapter a brief discussion on the main limitations of state of the art solutions that will be addressed by the results provided within this dissertation is presented.

2.1 Primary Copy

The Primary Copy (PC) replication approach [41], also known as passive replication [63], associates to each object to be replicated a specific site (a.k.a the primary copy). Any update request for that object must be sent to the primary copy, which processes the request and then propagates the updates to all the other sites. A variation of this scheme is when the primary is in charge to determine the serialization order according to which all the requests must be processed. Actual processing activities can then occur on all the replicas, such in a way to comply with the established serialization order. This approach is typically complemented by an election protocol (see, e.g., [77, 46] for instantiations of the leader election protocol suited for differentiated settings) which is in charge of assigning the role of new primary among backup servers in case of failure of the original primary.

2.1.1 Primary Backup

The Primary Backup (PB) approach [5] is a widely diffused implementation of PC. As summarized in [41], in this technique one replica, the primary, plays a special role: it receives invocations from the clients, processes the requests and returns the replies. On the other hand, the backups only interact with the primary, without exchange any data with the clients. Denoting with $prim(x)$ a primitive that defines whether the server x currently represents the primary, and with $op(arg)$ the client message together with its arguments, the below steps provide an outline of PB:

1. process p_i sends $op(arg)$ to $prim(x)$ together with a unique invocation identifier $invID$.

2. $prim(x)$ invokes $op(arg)$, which generates the response res. $prim(x)$ updates its state and sends the update message $(invID, res, state - update)$ to its backups, where the $state - update$ defines the new state reached by the primary. Upon receiving the update message, the backups update their state and return an acknowledgment to $prim(x)$.

3. Once the primary replica receives the acknowledgements from all the correct (non-crashed) backups, it sends the response to p_i.

This scheme ensures linearizability [68] since the order according to which the primary receives the invocations defines a total order for all the invocations. On the other hand, the receipt of the $state{-}update$ message by all the backups ensures the atomicity property. In case of crash of the primary, the protocol selects a new primary exploiting leader-election primitives. If a perfect failure detection mechanism can be assumed [20], the PB replication technique is relatively easy to implement but it becomes much more complicated when the failure detection mechanism is not reliable (e.g. the case in which a client may incorrectly suspect that the primary replica has crashed). The view-synchronous communication paradigm can be used to ensure correctness of the PB technique despite an unreliable failure detection mechanism [63].

Early implementations of PB were prone to scalability problems due to the presence of a single *active* node in charge of processing all the client requests [83]. More recently, a number of improvements of the original PB approach have been proposed. The work in [73] provided the possibility to execute read-only requests on top of backup servers, thus entailing the possibility to process in parallel operations that only query the state of the replicated objects without changing them. This helps improving the throughput especially in case of read-intensive workloads.

In [16] the PB scheme is applied in the context of in-memory database systems (IMDB). This is done by providing an algorithm for ensuring high availability of an IMDB with a low replication overhead. In this approach, an innovative middleware-level distributed algorithm exploiting assumptions properly valid for IMDB, reduces to two communication steps the latency needed to commit update transactions.

In [80] the authors propose two implementations of PB tailored for java objects. The first one uses the JAVA *remote invocation method* (RMI) to implement a simple and fast PB replication scheme for shared objects. The second one, called replica-proxy, is a more complex implementation only relying on JAVA network packages, which however yields improved performance.

2.2 State Machine

The State Machine (SM) approach has been proposed in [74]. It defines a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. By definition, a state machine consists of state variables, which encode its state, and commands that update the state. Each command is implemented by a deterministic program. So the abstraction devises two key components to be implemented within a SM protocol, which are:

- a communication system used by the replicas to coordinate with each others;
- a support, if needed, to execute the commands (or operations) in a deterministic fashion.

There are several aspects that make SM different wrt PC, but the most important is probably related to the assurance on the side of fault-tolerance. SM provides full failure-masking, which means that, in case of failure, the protocol does not pay direct additional costs (e.g. by delaying the processing of the request) to re-configure the replicas, as instead occurs in PC, where the system needs to re-elect a new primary in case the original one fails. Further, the SM approach may offer advantages also because, instead of allowing a single (primary) server to process the requests, multiple servers (possibly all) are able to take care of request processing.

Exploiting the absence of *passive* [1] nodes, in literature there are several works that ensure availability also empathizing performance indexes, e.g., system response time and throughput, just by relying on SM. In [65] a state machine replication approach has been used in the context of advanced simulation systems. This work is aimed at improving the response time of the application level software by exploiting the fastest instantaneous reply among diversity-based replicas of a same simulation component. Another example of state machine replication used as a means for jointly coping with availability and performance is in [64] where the authors propose a replicated multi-version cache that reduces the performance impact associated with the need for accessing underlying databases. The proposed solution has been fully embedded within the application server-tier on top of an off-the-shelf database. In particular, in this architecture, each node is not a single server but is instead formed by a pair that includes an application server and its local copy of the database. In the experimental study, the prototype outperforms the non-replicated application server and shows a good scalability vs the number of replicas in terms of both throughput and response time.

[1]Passive nodes are the processes that do not manage client requests.

Another approach oriented to performance improvements can be found in [70] where a state machine replication protocol explicitly tailored for multi-tier data acquisition systems is provided. This protocol does not rely on any replica coordination mechanism to be actuated prior to message processing but rather, it enforces consistency across the replicated data-gathering sinks only when strictly required, namely when some sink externalizes its current state by producing an output message destined to back-end applications.

As for SM protocols specifically oriented to replication of transactional systems, which are more strictly related to the results presented within this dissertation, we provide an overview in the next sections, also classifying them on the basis of the adopted approach to replication, namely *certification* vs *active replication*.

2.2.1 Certification

Certification-based protocols ensuring serializability over replicated transactional systems have been presented in [82, 49, 50]. As also pointed out in [83], one core aspect characterizing all these protocols is related to determinism, and on how it may (or may not) impact coherence of the state of the replicas. With these schemes, each transaction is locally processed without pre-imposing a deterministic execution path. Coherence is therefore demanded to the certification phase, where the local site broadcasts to the other replicas the state updates performed by the transaction. This is the phase that requires to be carried out deterministically (i.e. globally ordered) among all the members of the system, so that all the replicas can instal the updates in a coherent manner.

Clearly, the certification phase may fail, since the updates to be installed might be incoherent with the local state trajectory (due to locally processed concurrent transactions), in which case the transaction gets aborted at the origin site and then reprocessed. Certification-base techniques are intrinsically optimistic, thus potentially leading to significant abort/retry rate in specific scenarios.

Summarizing, certification-based protocols start the execution of a transaction on one delegate server in a non-deterministic fashion. Upon transaction completion, the delegate sends to the other replicas information about the data accessed (in either read or write mode) using a total order primitive that ensures globally ordered communication. When a replica receives the latter information, it installs the changes (if compliant with the local state) and sends back an acknowledgement. Only when all the confirmations have been received, the delegate replies to the client.

2.2.2 Active Replication

Active Replication (AR) is a technique that gives to all replicas the same role wrt any client request. In literature, it has been studied in a number of works [1, 49, 48, 60, 12]. In AR, a client can submit the request to any server in the system, in fact, at this stage, the role of the server is simply to act as a forwarder of the request to all the other replicas using a total order broadcast primitive. When a replica delivers the totally ordered message, it processes the transactional operations using a deterministic concurrency control that is able to provide a serializable history among concurrent operations respecting the total order imposed by the group communication protocol. Also, the transactional code needs to execute deterministically as well, which would require to filter-out requests exhibiting non-deterministic nature, or to adopt ad-hoc schemes to support their processing. The above features would lead each replica end-up in the same final state. The client delivers the response coming from the fastest replica that completes transaction processing.

The main advantage of AR is that it does not need a distributed deadlock detection system, since transactional requests are spread using a total order broadcast. Hence, in case of lock based concurrency control, locks for the whole transaction can be acquired atomically and in the same order at all the sites thereby preventing deadlocks. The bottleneck for such an approach is the time spent to reach an agreement among replicas about the delivery order, which appears on the critical path of request processing.

The problem of establishing, in a non-blocking fashion, the agreed upon total order is typically encapsulated by the so called Atomic Broadcast (AB) primitive. It is a group communication service, which represents a convenient abstraction of consensus, for which a wide spectrum of alternative implementations have been proposed [31] and prototyped [6, 55] in literature. Among them, we can find solutions that significantly improve performance in terms of throughput by, e.g., batching multiple messages [55].

The traditional version of AB has been anyway pointed out as too much conservative. This is because the client perceived response time is always composed by the replicas coordination delay (which, as hinted, appears onto the critical path of the client-server interaction) plus the execution time of the corresponding transaction. To tackle the latter issue, a variant of atomic broadcast has been proposed, termed Optimistic Atomic Broadcast (OAB) [48] which, based on optimistic assumptions about the spontaneous order of network deliveries, reduces the average delay for message notifications to the application [61]. OAB early delivers messages as soon as they arrive (typically after a single communication step) by optimistically guessing that the final ordering will comply with the arrival order. This provides the application layer with indications about the existence, and the possible final ordering, of the request, thus potentially allowing request pre-processing out of the critical path of coordination. As it will be discussed further in the next section, pre-processing optimistically delivered transactions (thus entailing a form of speculation) has been explicitly addressed by the work in [48], which provides therefore an innovative approach along the direction of overlapping the coordination phase associated with the atomic broadcast algorithm with the local transactional processing activities. It is worthy to note that pre-processing might require rollback mechanisms to be actuated in case the final delivery reveals as not compliant with the ordering guessed via optimistic deliveries.

2.3 Speculative Processing

Speculative processing (SP) is a well known approach aimed at improving performance. It has been applied in a number of different fields such as pipelined computing architectures [53, 72] and high performance computing systems and applications [21, 71, 76, 66].

In literature, the idea of exploiting speculation in transaction processing environments has been investigated in [10] and [67]. The first work [10] targets non-replicated real-time databases and shows the benefits, in terms of transaction timeliness, by speculatively forking, upon detection of a conflict, a copy of the current transaction that remains idle and serves as a save-point to reduce the rollback cost. The solution in [67] targets distributed databases relying on distributed locking and atomic commit for transaction validation. A speculative locking protocol is presented in which the waiting transaction is allowed to access the locked data objects whenever the lock-holding transaction makes after-images available during its execution. By exploring both the execution paths related to before-images and after-images, different speculative executions are carried out, one of which is retained (being it valid) upon the final outcome of the preceding conflicting transaction.

In [12] a novel approach is presented, which provides a low overhead active replication scheme for event-stream processing applications. The key contribution has been the usage of a speculation mechanism based on Software Transactional Memories. This approach enables the replicas to make progress based on optimistic deliveries while waiting for the final ordering of messages. It also enables nodes to forward speculative information, and downstream-nodes to pre-process such a speculative input, while also allowing multi-threaded processing in the context of transactional data manipulation.

In [13], speculation is used for out-of-order event processing. This work shows how speculation avoids idle periods due to the notification of the right processing order and, by experiments, shows how this strategy brings substantial performance benefits.

In the context of certification-based replicated transactional systems, the key idea at the core of [19] is to reduce via OAB-based schemes the time to disseminate the updates generated by committing transactions in order to provide executing transactions with fresher snapshots, thus reducing the probability of abort due to reads from stale data, and increasing the probability to detect conflicts earlier during transaction execution. This would lead to a reduction of the amount of wasted computation and useless waiting time caused by transactions doomed to abort.

For actively replicated transactional systems, a speculative processing framework has been proposed, based on the reliance on OAB [48]. We overview this framework detailedly since it will be used at some extent within this dissertation as a recent reference-result for the assessment and evaluation of the innovative approaches we propose. This framework aims at taking advantages from the overlap between the replica coordination phase, supported via OAB, and the (speculative) processing of transactions. The framework is based on optimistically broadcasting requests to all the replicas and on using the total order provided by the OAB service to serialize the processing of the requests coherently at all the sites. The transaction manager uses the tentative order determined by the optimistic delivery to establish a convenient schedule for the transactions, and then starts executing them.

Within the schedule proposed in [48], a single transaction is allowed to be optimistically processed along a chain of conflicting transactions. This corresponds to the top standing conflicting transaction according to the optimistic delivery order. The commitment of the optimistically processed transaction, however, is postponed, as well as the activation of the subsequent transactions along the conflicting chain. When the OAB has determined the definitive total order for the corresponding request, it delivers a confirmation (or not) of the delivery order for the optimistically delivered message. If tentative and definitive orders are the same, the transaction is committed. Otherwise, further actions need to be taken to guarantee that the serialization order will obey the definitive total order. This typically implies reordering of the chains of conflicting transactions that have been left pending for rearranging the optimistically explored serialization order.

Another important aspect for the framework in [48] is that the transactions that live within the system need to be pre-catalogued on the basis of the data they will access. This is required in order to build the chains of conflicting transactions associated with the schedule. One way to address this is to statically determine conflict classes under the assumption that transactions of a given class are only allowed to access objects of a certain partition of the whole data-set, consequently different conflict classes work on different data partitions (thus determining different chains within the schedule). This leads to the implicit assumption that transactions within one conflict class have a high probability of actually exhibiting conflicts, while transactions from different conflict classes do not conflict.

Overall, reshuffling the framework description on the basis of the notion of conflict classes, we have the below depicted behavior. For each conflict class C there exists a FIFO class queue CQ. The semantics associated with the conflict class structure is the following. When a transaction $T \in C$ is optimistically delivered, it is added to CQ. When T is the only transaction in C, then its (optimistic) execution can be started. When there are other transactions already queued in C, T has to wait. When a transaction commits (recall that it must be the first one in its queue), it is removed from the queue and the next waiting transaction starts to execute. Transactions in the same conflict class are executed sequentially, conversely when they are in different classes, their relative ordering is not pre-determined. Upon final delivery from the OAB service, which notifies that a transaction could be committed (according to a specific serialization order), if the transaction is correctly serialized, it gets committed if already executed. If the transaction is not yet fully executed, it is simply marked as committable. If no match occurs between the finally delivered transaction and the executed (or executing) transaction, then a mismatch happened within OAB. Hence the transaction must be aborted and the correctly ordered one starts its execution.

2.4 Discussion

As hinted, the results presented within dissertation have very strict relations with the framework in [48]. On the side of performance and efficiency, for this framework we can draw the following considerations. If the time it takes to receive confirmation of the message order is comparable to the time it takes to execute a single transaction, and the tentative (optimistic) delivery order mostly matches the final order, then successful overlap is achieved, since the latency in between optimistic and final deliveries is fully exploited in term of communication-overlapped processing activities. On the other hand, with different ratios between OAB-finalization latency and transaction granularity, the achievable benefits get likely reduced. As an example, if the transaction granularity results very small, as it may be the case for, e.g., STM-based environments, then the overlap likely comes out very partial, thus inducing system-stall periods. Similar considerations can be made in case the scheme is used on top of a massively parallel architecture (i.e. many-core machines), since the degree of parallelism allowed while speculatively processing transactions is limited by the amount of distinct conflicting chains (or conflict classes) that are determined on the basis of the transaction profiles. In fact, a single transaction along any chain of conflicting transactions is allowed to be speculatively processed.

These performance and resource usage limitations will be addressed by the innovative proposals provided within this dissertation. Overall, these proposals will provide theoretical and more pragmatical results on the context of replicated transactional systems, which will still deliver the same advantages already provided by the framework in [48]. In particular, these advantages can be expressed in terms of failure masking capabilities (thanks to the reliance on the active replication approach) and the exploitation of a highly optimized group communication primitive like OAB for coordinating the replicas, which, as also shown in [81], provides the potential for outperforming other transactional replication approaches. On the other hand, the innovative proposals guarantee additional advantages in terms of:

(a) the ability to cope with scenarios exhibiting reduced transaction granularity (like, e.g., for STM environments and/or when SSD technology is adopted);

(b) full exploitation of the hardware parallelism offered by the emerging technology, which would lead to benefits on the side of, e.g., response time and system throughput.

Chapter 3

Model of the Target System

In this chapter we introduce the general model of the distributed system we consider as the target for the transactional replication schemes presented within this dissertation. Additional model specifications or notations will be provided whenever required while presenting individual solutions to the problem of replicating transactional systems.

3.1 Distributed Processes and Coordination Primitives

We consider a classical distributed system [40] consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ communicating via message passing, which can fail according to the fail-stop (crash) model. The number of correct processes (i.e. processes that do not fail) and the system synchrony level are assumed suffice to implement an Optimistic Atomic Broadcast (OAB) protocol [62], for which we list below the interfaces we assume to be provided:

– *TO-broadcast*(m): which broadcasts message m to all the processes in Π;

– *Opt-deliver*(m): which delivers message m to a process in Π in a tentative, also called optimistic, order (a.k.a. optimistic delivery);

– *TO-deliver*(m): which delivers a message m to a process in Π in a so called *final order* that is the same for all the processes in Π (a.k.a. final delivery or conservative delivery);

– *Opt-DeliveredMsgs*$()$: returning the totally ordered list of optimistic delivered messages;

– *TO-DeliveredMsgs*(): returning the totally ordered list of final delivered messages.

The latter two interfaces are not classical AOB primitives. However, with no loss of generality, they have been added in order to ease the description of the protocols we present in the subsequent chapters.
The OAB service enforces the following classical properties [62]:

– **Termination**: If a correct process TO-broadcasts m, it eventually Opt-delivers m;

– **Global Agreement**: If a process Opt-delivers m, every correct process eventually Opt-delivers m;

– **Local Agreement**: If a correct process Opt-delivers m, it eventually TO-delivers m;

– **Global Order**: If two processes p_i and p_j TO-deliver messages m and m', they do so in the same order;

– **Local Order**: If a process TO-delivers m, it does this only after it Opt-delivers m.

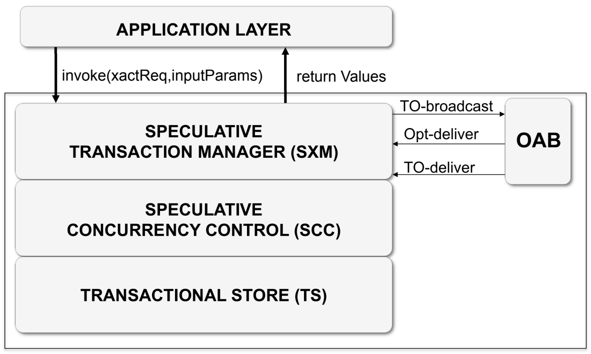

APPLICATION LAYER
invoke(xactReq,inputParams)
return Values
SPECULATIVE TRANSACTION MANAGER (SXM)
TO-broadcast
Opt-deliver
TO-deliver
OAB
SPECULATIVE CONCURRENCY CONTROL (SCC)
TRANSACTIONAL STORE (TS)

Figure 3.1: Replicated Process Architecture.

3.2 Internal Architecture of the Replicated Transactional Processes

Figure 3.1 shows the software architecture of each replicated process $p_i \in \Pi$. Applications, which represent the clients of the replicated transactional systems, generate transactions by calling the `invoke` method of the local Speculative Transaction Manager (SXM), specifying the business logic to be executed (e.g. the name of a DBMS stored procedure or of a method in a transactional parallel applications) and the corresponding input parameters (if any). SXM is responsible for:

– managing the `invoke` call, performed by the application, and propagating (via the *TO-broadcast* primitive) the transactional request across the set of replicated processes;

– executing the transactional logic also exploiting the functions offered by the underlying Transactional Store (TS);

– returning the corresponding result to the user-level application.

TS is the layer maintaining the local state of the replica, modelled as a set of (multi-versioned) data items. It provides classical facilities for making atomically (all-or-nothing) visible a new version of data item generated by a write operation performed by a transaction. Each data item X, maintained by the TS layer, is associated with a set of versions $\{X^1, \ldots, X^n\}$, where each version X^i stores:

– the data item value; and

– the identity of the creating transaction, namely the transaction that wrote the data item, thus generating that version.

A single version of X can be committed at any time. Uncommitted versions, residing within TS, are reflections of speculative computations. These uncommitted versions are used to propagate updates along chains of speculative serialization orders. Further, TS exposes proper primitives to mark/update the state of stored data item versions. These primitives could anyway change depending of the specific protocol requirements, so they will be specified on the basis of proper needs right upon presenting differentiated replication approaches and protocols within subsequent chapters.

At a logical level, the TS layer abstracts underlying storage mechanisms, which may encompass RAM-only memory accesses (as for the case of transactional memories) or logging on persistent storage to ensure transaction durability (as for the case of conventional DBMSs). Such an abstraction decouples the proper mechanisms related to the transactional replication protocol wrt the specific implementation of the underlying transactional store, bringing benefits in terms of protocol design and flexibility.

As depicted in Figure 3.1, the interactions between SXM and TS are mediated by the Speculative Concurrency Control (SCC) layer. It is responsible of executing concurrent transactions implementing the actual business logic. It also offers the supports for the activation (or re-activation) of speculative instances of not yet committed transactions. Abstracting from its internal mechanisms, it externalizes a classical interface for the execution of read/write operations on the data items, as well as for finalizing transactions with a commit or an abort.

In order to possibly manage multiple speculative instances of the same transaction T_i, and consequently to permit their univocal identification among all the activated transactions, SCC relies on an additional identifier, namely *specId*. The notation T_i^{specId} identifies therefore the speculative instance *specId* of the transaction T_i. Overall, all the speculative instances of transaction T_i are marked as different members of the same *family* i. All the transactions within the same family are named *sibling* transactions. Finally, the notation T_i^* identifies any speculative instance of transactions belonging to the family i.

3.3 Transaction Model

Each transaction T is modelled through a sequence of operations, denoted with $\mathcal{O}_T = \{o_1, \ldots, o_n\}$, where each operation is either a read or a write operation on a data item. To be highly general, it is assumed that neither the sequence of operations to be executed within a transaction, nor the data items to be accessed, are a-priori known. Conversely, it is assumed that the transaction data access pattern can vary depending on the current state of the underlying transactional store. More precisely, the transactional business logic is considered to be *snapshot deterministic* in the sense that, if a transaction is activated multiple times, the sequence of read/write operations performed by each activation of the transaction does not change unless the return value of any of its reads changes. In other words, if whichever instance T_i^j of a transaction T_i always sees a snapshot S, defined as the set of values returned by all its read operations, then it behaves deterministically by always executing the same sequence of read/write operations. On the other hand, if different instances of a transaction T_i are activated on different snapshots, they may generate different sequences of operations.

More formally, consider two executions of the generic transaction T, producing respectively the two sequences of operations $\mathcal{O}'_T = \{o'_1, \ldots, o'_n\}$ and $\mathcal{O}''_T = \{o''_1, \ldots, o''_k\}$ and assume, with no loss of generality, that $k < n$. Let $j \in [1 \ldots k]$ be the index of the first operation in $\mathcal{O}'_T$, $\mathcal{O}''_T$ for which $o'_j \neq o''_j$. This implies that before executing o'_j, respectively, o''_j, a read on a data item X was executed both in $\mathcal{O}'_T$ and $\mathcal{O}''_T$, which returned two different values.

A transaction instance T_i^j that has fully executed its sequence of operations $\mathcal{O}_{T_i^j}$ (and produced a result ready to be delivered to the application layer once, and if, the transaction is actually committed) is called a *completed* transaction instance. With no loss of generality we assume the possibility to intercept completion events, which could be done in actual transactional architectures by mostly wrapping transaction finalization (internal) commands.

Chapter 4

A Quantitative Reassessment of Literature Proposals

In this chapter we provide some experimental results for a quantitative reassessment of the efficiency of transactional replication protocols presented in literature. This has been done in the light of the provided considerations about the need for a verification of whether these protocols can still well fit (or not) execution dynamics induced by the technological changes discussed in Section 1.3. The latter deal with reduction of the computation to communication ratio within the replicated transactional system, as well as to the potential for largely scaled-up parallelism.

In order to consider a scenario representative of such changes, the present study is based on simulation results related to the context of applications relying on the Software Transactional Memory (STM) paradigm. In terms of abstractions, the STM case exactly models reduced ratio between transaction granularity and coordination latency, which may even mimic contexts where traditional database applications are run in top of SSDs based storage systems.

In the present study we consider a traditional SM approach to replication, and also the relatively recent active replication framework proposed in [48], relying (as detailedly explained) on the OAB service for replica coordination, and on some form of transaction speculation. As already hinted, this framework can be considered (and has been shown to actually be [81]) an optimized solution for both its failure-masking capabilities and the reliance on last generation, performance prone group communication facilities.

4.1 The Trace Based Simulation Model

The simulation study presented in this chapter has been based on data access traces related to commonly used (S)TM benchmark applications [42]. The provided performance evaluation study is based on a process-oriented simulator developed using the JavaSim simulation package [54].

In order to accurately model the execution dynamics of transactions in STM systems, we rely on a trace-based approach. Traces related to data accesses and transaction duration have been collected by running a set of widely used, standard benchmark applications for (S)TMs. The machine used for the tracing process is equipped with an Intel Core 2 Duo 2.53 GHz processor and 4GB of RAM. The operating system running on this machine is Mac OSX 10.6.2, and the used STM layer is JVSTM [15]. The simulation model of the replicated STM system comprises a set of 4 replicated STMPs, each of which hosted by a multi-core machine with 8 cores exhibiting the same power as in the above architecture. During the tracing phase, we configured the benchmarks to run in single threaded mode, so to filter out any potential conflict for both hardware resources and data. Also, we extended JVSTM in order to transparently assign a unique identifier to every object within the STM memory layout and to log every operation (namely, begin/commmit/abort operations, and read/write memory-object access operations) along with its timestamp. This allowed us to gather accurate information on the data access pattern proper of the benchmark applications and on the time required for processing each transaction (in the absence of any form of contention).

Since any tracing strategy unavoidably introduces overheads, especially in STM applications where transaction execution times are often less than 1 msec, in order to ensure the accuracy of the information concerning the duration of transactions, we repeated each benchmark run (ensuring the deterministic re-execution of an identical set of transactions) by also disabling the logging functionality. Then we compared the resulting mean transaction execution time with the one obtained when logging is enabled. This allowed us to compute a per-benchmark scaling factor (that was found to be on average around 15x) that was used to adjust the duration of the transaction execution, filtering out the overheads associated with the logging layer, before feeding this information to the simulator.

The traces were collected running three benchmark applications, RB-Tree, SkipList and List, that were originally used for evaluating DSTM2 [42] and, later on, adopted in a number of performance evaluation studies of (S)TM systems [15, 23]. These applications perform repeated insertion, removal, and search of a randomly chosen integer in a set of integers. The set of integers is implemented either as a sorted single linked list, a skip list, or a red-black tree. We configured the benchmark to initialize the set of integers with 128 values, and allowed it to store up to a maximum of 256 values. Finally, we configured the benchmark not to generate any read-only transaction (i.e. searches). This has been done in compliance with the operating mode of the protocol in [48], where read-only transactions can be executed locally at each single replicated process, without the need for propagation via the atomic broadcast (read-only transactions do not alter the state of the replicated transactional memory system). By only considering update transactions in our study, we can therefore precisely assess the impact of the coordination latency on the performance of a replicated STM, as well as the performance gains achievable by means of the optimistic approach proposed in [48]. The below table reports the average transaction execution time observed for the three benchmarks via the aforementioned tracing scheme:

Benchmark	**Average Transaction Execution Time (μsec)**
RB-Tree	77
SkipList	281
List	324

As for the delay due to the OAB layer (i.e. the delay of the Opt-delivery and of the corresponding TO-delivery), several studies have shown that OAB implementations typically tend to exhibit flat message delivery latency up to saturation [33], namely when the number of incoming messages is so high to congest the network and the OAB service. On the other hand, our study is not targeted to explicitly assess the saturation point of the OAB group communication subsystem. For this reason we decided to run the simulation by assuming that the OAB layer does not reach its saturation point. Therefore, independently of the value of the message arrival rate λ, we use in our simulations an average latency of 500 microseconds for the Opt-delivery, and of 2 milliseconds for the TO-delivery. These values have been selected based on experimental measures obtained running the Appia [55] GCS Toolkit on a cluster of 4 quad core machines (2.40GHz - 8GB RAM) connected via a switched Gigabit Ethernet, sending 100 bytes messages at a throughput ranging from 1 up to about 4000 messages per second.

Finally, we consider an ideal scenario with no mismatch between the optimistic and the final (totally ordered) message delivery order. This is because the protocol in [48] has been explicitly targeted to environments where such a spontaneous ordering property holds (e.g. LAN based environments). This clearly represents a best-case scenario for the protocol in [48], which allows us to establish an upper bound on the performance achievable by this protocol vs the behavior of OAB protocol (in terms of message ordering).

4.2 Simulation Results

In Figure 4.1 we report response time and CPU utilization values for the three benchmarks, while varying the arrival rate of transactional requests λ from the

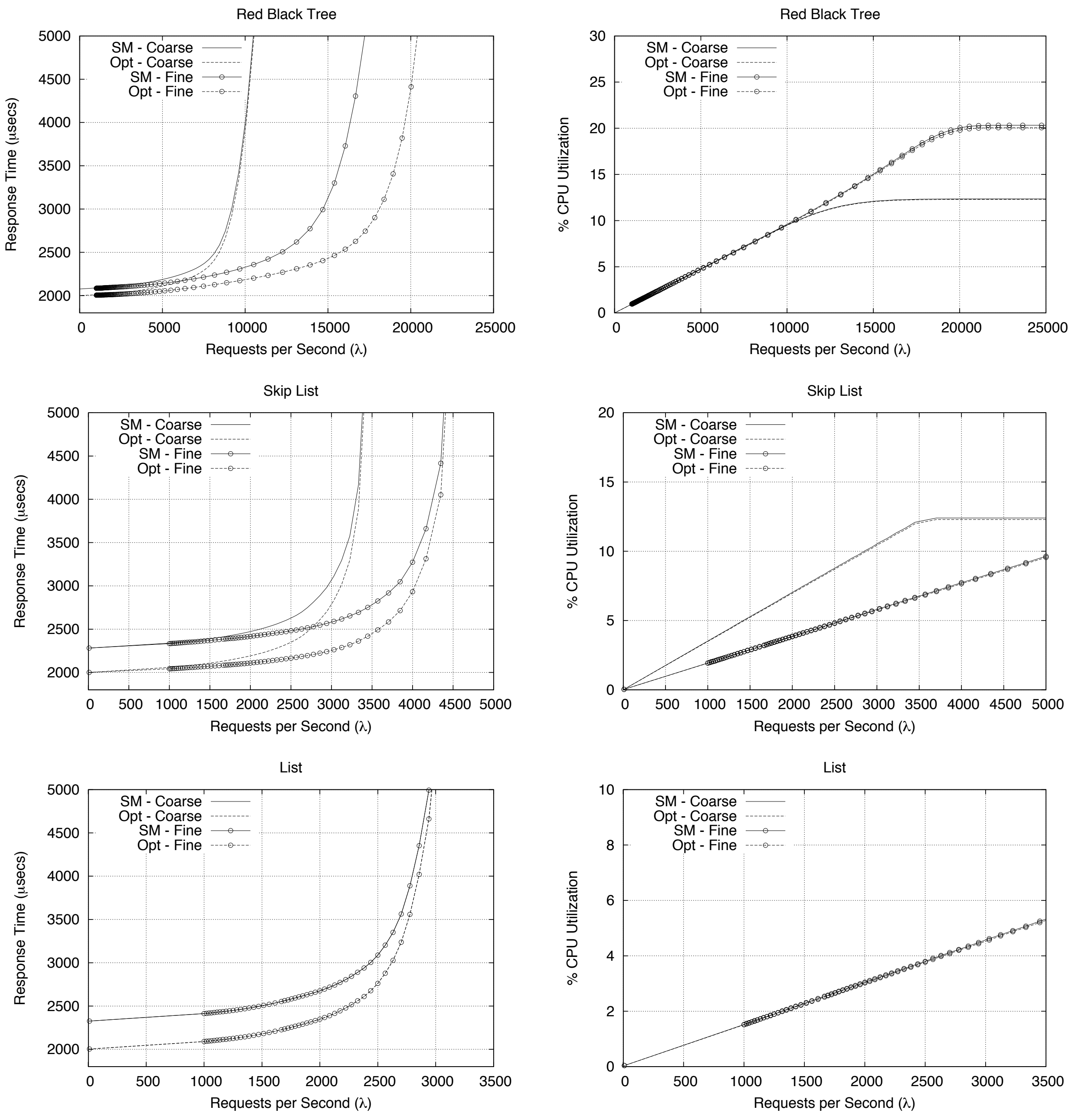


Figure 4.1: Performance of SM vs Opt.

OAB layer. We plot results related to 4 different configurations for the replication protocols. Opt-Fine and Opt-Coarse refer to the values observed when employing the replication protocol in [48] by using either the actual transaction conflicts (as determined by using the exact access to memory objects as determined by the benchmark execution traces) or a coarse conservative estimation where each pair of concurrent transactions is assumed to always conflict on some object inside the STM memory layout. For these two different conflict patters we also report the performance observed via a State Machine (SM) approach relying on traditional, non-optimistic Atomic Broadcast, where transactions are activated only after the group communication layer has notified their correct position.

As for the Opt-Coarse configuration, we note that it is an expression of adverse data access patterns, where according to the protocol in [48], speculation can occur along a single chain of (hypothetically) conflicting transactions, with a single transaction at any time being allowed to be optimistically processed (namely, the top standing one within the chain). However, we also note that a relevant point for the viability of the approach in [48] is that it requires a-priori knowledge of both read and write sets associated with transactions, in order to a-priori identify conflicting transaction classes. As for this aspect, the difficulty to exactly identify the data items to be accessed by transactions before these are actually executed, may lead to adopt conservative conflict assumptions based on coarse data granularity, e.g. whole, or large slices of, database tables [64]. However, unlike relational database systems, STM-based applications are characterized by arbitrary memory layouts and access patterns which may make significantly harder, or even impossible, to a-priori identify, with a reasonable accuracy (or a reasonable conservative approach), the boundaries of the memory regions that will be accessed by transactions prior to their execution. On the other hand, large over-estimation of the actual transaction conflicts is an additional performance adverse factor since it can strongly hamper concurrency, leading to significant resource under-utilization in (massively) parallel systems. This is exactly the reason why we feel it important to assess in this simulation study what may happen, in terms of performance, when considering such an (over-estimated) coarse grain approach to the identification of the potential conflicts.

By the results we can observe two major tendencies.

– For all the benchmarks, CPU utilization at the saturation point is always lower than 20%. Given that in our simulation the OAB layer is configured to respond in its flat region, this is a clear indication that data conflicts are the cause of system saturation. The worst case for the impact of data conflicts on the saturation point can be observed for List, where the CPU utilization does not even reach 6%, and the response time curves in case of actual conflicts exactly coincide with the corresponding ones obtained via coarse conflict estimations. The latter phenomenon is less evident for the other benchmarks, especially RB-three. However, it is a clear indication that data access patterns in STM environments may anyway exhibit actual conflict levels significantly grater than in database applications, thus requiring investigations on optimized concurrency control schemes, especially when employed in replicated environments where transaction blocking up to the determination of the final order for already active conflicting transactions may have excessive impact on resources under-utilization.

– The second tendency we can observe is related to limited advantages provided by the Opt scheme over SM in terms of effects of the overlapping between coordination and computing phased vs the transaction execution latency. This is noted especially for the very fine grain RB-three benchmark, exhibiting mean transaction execution time significantly lower that the OAB delay (when considering final TO-deliveries). Such a reduced transaction granularity, in combination with non-minimal transaction conflict levels observed even for the fine conflict determination approach based on actual accesses, leads to very reduced gains from the overlap. In fact, the coordination phase goes in overlap with a very reduced amount of fine grain computing activities, whose individual delay is actually negligible compared to the coordination latency.

4.3 Outcomes

By the performance results shown in the previous section, there is a clear need for revising replication management approaches for setting resembling, e.g., STM-based transactional systems, but more generally for systems that bring an unbalancing of the ratio between the replica-coordination delay and local transaction execution time, compared to their counterpart for traditional database systems. We envisage at least two ways according to which the revision could be actuated:

A. The first one regards the local concurrency control within each replica. It should allow an increased level of overlap between coordination and computing phases. This would entail increasing the level of optimism in transaction processing activities by avoiding the stall until the total order is reached, in case of Opt-delivered transactions that conflict with already active ones.

   Such an aggressive-optimistic approach can provide advantages on the response time. The maximum response-time benefits from this approach are expected to be observed still when the Opt-delivery order well matches the final TO-delivery. This is because optimistic anticipation in the execution of chains of conflicting transactions before the corresponding total order gets determined will not likely result in cascading abort scenarios otherwise caused by discrepancies between Opt-delivery and TO-delivery orders when spontaneous ordering properties do not hold.

This approach should anyway provide no advantage in terms of CPU utilization for processing activities associated with optimistic delivered transactions that match the final total order. In fact, compared to the scheme in [48], an increased level of optimism would simply yield to anticipate the execution of conflicting transactions before the total order for the oldest transaction along the conflict-serialized order is reached.

B. The second one regards the need to amortize response time penalties that could arise in case when spontaneous ordering is not guaranteed by concurrently exploring differentiated serialization orders, possibly according to the aggressive-optimistic scheme provided in the above point. This would lead to increase hardware resources utilization without negatively impacting response time. The set of serialization orders to be explored while notification of the correct one takes place via the finalization of the OAB service could be determined on the basis of heuristics or a clear theoretical analysis, allowing the avoidance of redundant execution of equivalent serialization orders on the basis of actual transaction conflicts.

Both the above depicted paths for reorganizing the design of actively replicated transactional systems are actually passed through in the subsequent chapters. In particular, in Chapter 5 we explore point (A) by the introduction of a protocol based on an innovative, aggressively optimistic concurrency control scheme to be locally actuated at the level of each individual replicated transactional process, tailored for predictable networks (e.g., networks ensuring spontaneous order). In Chapter 6 and in Chapter 7 we instead investigate along the line of point (B) by providing, respectively, (i) a theoretical framework for non-redundant speculation along any meaningful serialization path, to be possibly established by the final deliveries within the OAB-based coordination, and (ii) an instance of a more pragmatic protocol where the serialization orders to be speculated are dynamically identified in an opportunistic manner depending on run-time factors such as the actual level of concurrency among transactions, as well as the actual level of transaction conflict.

Chapter 5

Speculative Replication in Predictable Networks

In this chapter we present a protocol named *AGGRO* aimed at boosting replicated transactional systems via an AGGRessively Optimistic transaction processing scheme. It is particularly suited for systems deployed on networks characterized by the spontaneous order property (i.e. optimistic and final delivery orders mostly match).

The key idea behind AGGRO is to seek maximum overlap between replica coordination and transaction execution phases by propagating the (uncommitted) post-images of complete, but not yet finally delivered, transactions across chains of conflicting transactions speculatively executed in a serialization order compliant with the optimistic delivery order. To ensure that the actual transaction schedule matches the serialization order determined by the sequence of optimistic deliveries, AGGRO relies on an innovative concurrency control mechanism that, unlike existing OAB-based replication approaches, does not require information on the transactions' data access patterns prior to their actual execution. Conversely, it detects any possible discrepancy between the transaction schedule and the optimistic delivery order a posteriori, namely as soon as (and if) conflicts materialize.

As we will show by means of a detailed simulation study in the context of STM applications, AGGRO allows achieving up to 75% reduction of the transaction execution latency and 6x throughput increase with respect to state of the art OAB-based replication schemes when deployed on replicas equipped with an eight-core CPU (which today represents a typical configuration for commodity server systems). Such performance gains are obtained without sacrificing consistency. In fact, beyond ensuring 1-copy serializability, AGGRO also enforces opacity [39] by guaranteeing that the snapshot observed by any (eventually committed or aborted) transaction is always equivalent to one generated by a serial schedule, albeit possibly not matching the one associated with either the optimistic or the final delivery order.

5.1 System Model

We consider the same system model already presented in Chapter 3. Actually, AGGRO does not need to demarcate transactions according to the notation T_i^{specId}, where i is the family identifier and *specId* is the speculative instance identifier. This is due to the fact that with AGGRO a single speculative instance of a given family can be active at any time. Therefore, for simplicity, the *specId* superscript has been dropped while presenting the AGGRO protocol.

Regarding the diagram in Figure 3.1, which depicts the reference software architecture of each replicated process, in addition to the functionalities already listed for the Speculative Concurrency Control (SCC), with no loss of generality, we assume the existence of another function Complete(), used to explicitly inform the Speculative Transaction Manager (SXM) about the completion of the execution of a transaction.

Further, the Transactional Store (TS) is extended in order to take into account a new state for each version of data item stored. Summarizing, with AGGRO, a single version of data item X could be in the following different states:

– committed;

– uncommitted.

Moreover, the uncommitted versions are further classified as:

– *Work-in-progress (Wip)*, that are versions for which the creator transaction has not yet reached the complete stage;

– *Complete (Comp)*, that are version for which the creator transaction has reached the complete stage, but is not finalized as committed or aborted yet.

More in detail, complete data versions are generated by fully executed transactions, and are used to aggressively propagate updates to conflicting transactions. On the other hand, declaration of the existence of Wip versions is used by AGGRO as a means to early express that a given data item is being currently manipulated by some transaction.
The manipulation of the data items occurs via the following primitives offered by the TS layer:

– `MarkAsWip(`T`,`X^T`)`, which is used for declaring the existence of a Wip version of data item X created by transaction T;

– `UnmarkAsWip(`T`,`X^T`)`, which is used for un-declaring the existence of a previously declared Wip version of data item X by transaction T;

– `MarkedAsWip(`T`,`X`)`, which is used to query the existence of a Wip declaration on X by transaction T;

– `setCompleteVersion(`X^T`,`T`)`, which is used for updating the state of a Wip data item X^T created by T (hence belonging to the write-set of transaction T) to Comp;

– `unsetCompleteVersion(`X^T`,`T`)`, which is used for removing a complete data item version X^T originally created by T.

5.2 The AGGRO Protocol

In our architecture, the speculative concurrency control (SCC) exploits the aforementioned data item versioning mechanism to locally drive the execution of transactions. Data item versions in the *Comp* state are aggressively made visible to other transactions independently of whether the creating transactions will be eventually committed. On the other hand, the SCC selects the complete/committed data item versions to be returned by read operations in order to match a serialization order compliant with the order in which transactions are optimistically/finally delivered within the OAB scheme. For environments where the spontaneous network ordering property holds, the optimistic delivery order highly likely reflects the final total order.
Hence, transactions reading *Comp* versions on the basis of the order according to which they have been optimistically delivered are expected not to be eventually (cascading) aborted. In other words, aggressiveness in transaction processing via access to uncommitted (but complete) data items is expected to pay-off:

(i) by avoiding to stall processing waiting for the finalization of the delivery order;

(ii) by not requiring transaction abort and restart.

On the basis of the above considerations, the role of *Wip* data items becomes central. They represent an early declaration about the fact that a new data item version is likely to reach the *Comp* state in the (immediate) future. Hence, the SCC can exploit the presence of *Wip* versions to regulate concurrency in a way to temporarily suspend the execution of a transaction T that requires read-access to that data item, and that follows the creating transaction T' in the optimistic/final delivery order. On the other hand, an adverse schedule may lead T to execute the read operation before T' has been able to issue its

write on that data item, thus not being able to declare the existence of its *Wip* version before T issues the read operation.
To cope with such a case, we have introduced within SCC an early abort mechanism ensuring that T gets aborted as soon as the *Wip* version by T' gets produced.

As for the above point, for fine granularity transactional applications (like with STM applications) hosted by massively (or even conventional) multi-core architectures, unless for extremely high request-concurrency scenarios, we expect minimal likelihood for the optimistically/finally delivered transaction T' not to have reached the complete phase (or to have declared the existence of *Wip* versions) before the subsequent optimistically/finally delivered transaction T gets activated (thus accessing the post image of data with respect to T'). This is because:

(A) transactions typically exhibit very fine granularity;

(B) as we have also shown in Section 4.2, in typical settings, there are normally available computational resources to start processing transactions immediately upon their delivery.

On the other hand, in environments with stricter hardware resources (CPU-cores) limitations, the AGGRO concurrency control scheme can be easily integrated with a CPU scheduling scheme (supported at the level of SXM-handled threads) based on dynamic priorities, which can favor older transactions within the optimistic/final delivery order. This would create a time-sharing execution that is likely to allow the older transaction T' to declare the existence of *Wip* versions, or to even run to completion, before T gets actually executed. We omit such a CPU schedule integration mechanism in the presentation of the AGGRO pseudo-code exclusively for simplicity.

The behavior of the SCC within the AGGRO protocol relies on a precedence relation between transactions, defined on the basis of the order according to which they are optimistically and/or finally delivered. The relation is expressed as a function of the state of two lists maintained by the SXM:

– OptDelivered;

– TODelivered.

These lists keep, respectively, transactions that have been either optimistically or finally delivered, and are sorted according to the corresponding delivery order.

When a transaction T is optimistically delivered, it gets recorded at the tail of the OptDelivered list. Upon the corresponding final delivery, the transaction is moved from the OptDelivered list (whichever is its current position within this list) to the tail of the TODelivered list. The move operation between the two lists is handled by the SXM as an atomic action. Finally, the transaction is removed from the TODelivered list upon commit. In case of no discrepancy between the OAB optimistic and final delivery orders, the transaction moved at the tail of the TODelivered list is always the head-standing one (namely the oldest one) of the OptDelivered list.

By exploiting the above ordered lists, the precedence relation among transactions is expressed as follows. We say that transaction T_i precedes transaction T_j according to the current state of the OAB protocol (as expressed by the lists), using the notation $T_i \overset{OAB}{\to} T_j$, if one of the three below mutually exclusive conditions holds:

1. T_i and T_j are both currently recorded within OptDelivered, with T_i ordered before T_j;
2. T_i is currently recorded within TODelivered, while T_j is currently recorded within OptDelivered;
3. T_i and T_j are both currently recorded within TODelivered, with T_i ordered before T_j.

We note that the $\overset{OAB}{\to}$ relation is dynamic, in the sense that, when considering a couple of transactions T_i and T_j, their respective $\overset{OAB}{\to}$ ordering can change over time. This may occur in case they get sorted within the TODelivered list in the opposite manner, compared to the sorting they had within the OptDelivered list (i.e. in the case of discrepancy between optimistic and final delivery orders for the two transactions).
However, once that both these transactions are recorded within the TODelivered list, their respective $\overset{OAB}{\to}$ order becomes stable (it can no longer be inverted), and depends on which of the two transactions is ordered (and hence TO-delivered) before the other one in the list (see point 3 above).
This relative order persists until the preceding transaction gets removed from the TODelivered list upon its commit.

The pseudo-code for the behavior of the SCC is shown in Figure 5.1. For focusing on protocol, we do not explicitly show the handler for the receipt of transactional requests by the overlying application, as this simply entails a TO-broadcast operation for propagating the request to the replicated sites via the OAB service. Similarly, we do not explicitly show the logic for the retrieval of the transaction result upon a commit operation, and the delivery of the result to the overlying application. In other words, the pseudo-code presentation is focused on the core mechanisms associated with transaction processing and concurrency regulation, which are activated as soon as a TO-broadcast transactional request gets Opt-delivered to the SXM by the OAB layer. Via the

`Opt-deliver` handler, a transaction is inserted within the OptDelivered list, and then gets activated via the ActivateTransaction() function, which we use to encapsulate the transaction processing logic triggering an a-priori unknown sequence of read and write operations.
Whenever a write on a data item X is issued, the SCC activates the Write() function, which first checks whether X already belongs to the transaction write-set. In the positive case, the working copy within the write-set gets updated, and then the Write() function simply returns.
On the other hand, if X does not currently belong to the write-set, it is added to it along with the to-be-written value.
Successively, the SCC declares via the `MarkAsWip()` primitive the existence of a *Wip* version associated with the currently writing transaction, say T_i.
Then the SCC verifies whether there are active transactions that follow T_i according to the $\overset{OAB}{\rightarrow}$ relation, and that read X from a transaction T_k different from T_i such that $T_k \overset{OAB}{\rightarrow} T_i$. These transactions are not correctly serialized according to $\overset{OAB}{\rightarrow}$ since they should have read X from T_i or a subsequent transaction within the $\overset{OAB}{\rightarrow}$ ordering. Hence an abort event for these transactions is issued.
By the above explanation of write operations, the multi-versioning mechanism supported by TS, and exploited by the SCC, actually provides a means for avoiding stalls upon write/write conflicts.

In case the requested operation is a read on data item X, the SCC activates the Read() function, which first checks whether X is already registered within the transaction write-set/read-set. In the positive case, the registered copy is returned. Otherwise, the SCC checks whether the reading transaction, say T_i, follows, according to the $\overset{OAB}{\rightarrow}$ relation, some transaction for which a working copy of X is declared to exist. In the positive case, transaction T_i is temporarily suspended until the above condition is no more verified. Afterwards, the complete or committed version of data item X wrote by the latest transaction preceding T_i according to $\overset{OAB}{\rightarrow}$ is selected and added to the read-set. Then the read-from set of T_i is updated in order to include the read-from dependency associated with the transaction that wrote the selected version of X. Finally, this version is returned.

In the Complete() function, the SCC simply removes the declaration about the existence of *Wip* versions associated with the transaction, and then marks each data item X belonging to the write-set as *Comp*. Afterwards, the transaction enters in the complete state.

In the Commit() function, the SCC installs the *Comp* versions of the data items written by the transaction as committed versions. Then the transaction is removed from the TODelivered list.

In the Abort() function, the SCC triggers a (cascading) abort event for all the transactions that read whichever data belonging to the write-set of the currently aborting transaction. These data are then simply discarded, the current transactional context is released, and a new thread for reactivating the transaction is spawned.

Via the `TO-deliver` handler, ther SXM moves the transaction from the OptDelivered list to the TODelivered list. Then the execution of this handler is suspended until the finally delivered transaction enters in the complete state. The suspend condition also depends on whether there are other transactions that precede the currently TO-delivered one according to $\overset{OAB}{\rightarrow}$. In such a case, the handler waits until both of the following conditions are verified for the currently TO-delivered transaction:

(i) it is fully executed; and

(ii) it becomes the minimum element within the $\overset{OAB}{\rightarrow}$ relation.

Note that in AGGRO, waiting until a TO-delivered transaction T_i becomes the minimum element of the $\overset{OAB}{\rightarrow}$ relation ensures that every transaction preceding T_i according to $\overset{OAB}{\rightarrow}$ has already been safely committed. At this point T_i can be safely validated by checking whether all the values read by T_i coincide with the ones that are currently in the committed state. If the validation phase is successfully passed, the `TO-deliver` handler generates the commit event for the transaction, which causes the installation of the data items written by the transaction, and the corresponding values, as committed, as well as the removal of the transaction from the TODelivered list.
This enables the redefinition of a new minimum element, which iteratively allows generation of the commit event for the corresponding transaction, once it reaches the complete stage.

5.3 Protocol Correctness

In this section the set of safety, liveness and correctness properties ensured by AGGRO is discussed.
As for safety, AGGRO ensures *opacity* [39] and *1-Copy Serializability* [8].
The opacity property guarantees that:

(O.1) committed transactions should appear as if they were executed sequentially, in an order that agrees with their real-time ordering;

(O.2) no transaction should ever observe the modifications to shared state done by aborted or live transactions;

(O.3) all transactions, including aborted and live ones, should always observe a consistent state of the system.

In each replica, AGGRO ensures property (O.1) by committing transactions only after a validation phase that would detect any unserializable behavior. It ensures (O.2) because read operations can only return either a committed value, or the value generated by a transaction whose execution has already reached the complete phase (and hence is neither live nor aborted at the time of the read). It ensures (O.3) since the read of a transaction T_i always returns the value generated by the latest complete transaction that precedes T_i according to $\overset{OAB}{\rightarrow}$.
Hence, the only possible anomaly that could affect T_i arises whenever T_i observes a value for a data item X generated by a transaction T_j such that $T_j \overset{OAB}{\rightarrow} T_i$, and then a transaction T_k, where $T_j \overset{OAB}{\rightarrow} T_k \overset{OAB}{\rightarrow} T_i$, writes X. In this case, if T_i were to issue a read on any data item generated by T_k, T_i would observe an inconsistent state (having already been serialized before T_k when it issued the read on X), thus violating (O.3). On the other hand, this scenario is avoided by AGGRO since, as soon as T_k writes on X, it would detect that T_i has been scheduled in a way that is inconsistent with $\overset{OAB}{\rightarrow}$, and would immediately abort T_i.
Concerning 1-Copy Serializability, this is ensured by AGGRO since transactions are committed at every site only upon a deterministic validation that is executed by all replicas in the same total order, i.e., the final delivery order of the OAB service.
As for liveness, AGGRO ensures *lock-freedom*, which guarantees that there is always at least a thread to make progress, thus ruling out deadlock and livelock scenarios. This is a direct consequence of the fact that the transaction currently representing the minimum element according to $\overset{OAB}{\rightarrow}$ always experiences an abort free (re)run.

5.4 Simulation Study

Our performance evaluation study is based on a process-oriented simulator developed using the JavaSim simulation package which implements:

(i) the OAB-based replication protocol in [51], referred to as OPT in the following;

(ii) the proposed AGGRO protocol.

As hinted, this study deals with STM applications, and in order to accurately model the execution dynamics of transactions in STM systems, we rely on a trace-based approach. Traces related to data accesses and transaction duration have been collected by running a set of widely used, standard benchmark applications for STMs. The machine used for the tracing process is equipped with an Intel Core 2 Duo 2.53 GHz processor and 4GB of RAM. The operating system running on this machine is Mac OS X 10.6.2, and the used STM layer is JVSTM [14]. The simulation model of the replicated STM system comprises a set of 4 replicated STM processes, each hosted by a machine equipped with eight cores processing transactions at the same rate as in the above architecture. As for the study presented in Chapter 4, we again configured the benchmarks to run in single threaded mode, so to filter out any potential conflict for both hardware resources and data. Also, we extended JVSTM in order to transparently assign a unique identifier to every object within the STM memory and to log every operation (namely, begin/commit/rollback operations, and read/write memory-object access operations) along with its timestamp. This allowed us to gather accurate information on the data access patterns of the benchmark applications and on the time required for processing each transaction (in absence of any form of contention). The traces were collected running the three benchmark applications already presented in Section 4.1, RB-Tree, SkiptList and List. Further, we configured the benchmark not to generate any read-only transaction because in both protocols considered in this study, read-only transactions can be executed locally, without the need for propagation via the atomic broadcast. By only considering update transactions, we can therefore precisely assess the impact of the atomic broadcast latency on the performance of a replicated STM, as well as the performance gains achievable by AGGRO. The transactions' arrival process via optimistic and final message deliveries from the OAB layer is modeled in our simulations via a message source that injects messages having as payload a batch of β transactions with an exponentially distributed inter-arrival rate, having mean λ. We recall that batching is a technique very commonly employed to optimize the performance of (Optimistic) Atomic Broadcast protocols [31]. By amortizing the costs associated with the (O)AB execution across a set of messages, batching schemes have been shown to yield considerable enhancement of the maximum throughput achievable by (O)AB protocols. The inclusion of batching schemes in our study of OAB-based replication protocols for transactional systems allows keeping into account optimized configurations for this important building-block group communication primitive.

As for the delays of optimistic and final message deliveries, several studies have shown that OAB implementations typically tend to exhibit flat message delivery latency up to saturation [33]. On the other hand, our study is not targeted to explicitly assess the saturation point of the OAB group communication subsystem. For this reason we decided to run the simulations by assuming that the OAB layer does not reach its saturation point. Therefore, independently of the value of the message arrival rate λ, we use in our simulations an average latency of 500 microseconds for the Opt-delivery, and of 2 milliseconds for the TO-delivery. As for the study in Chapter 4, these values have been selected on the basis of experimental measures obtained running the Appia [55] GCS Toolkit on a cluster of 4 quad-core machines (2.40GHz - 8GB RAM) connected via a switched Gigabit Ethernet.

5.5 Simulation Results

The plots in Figure 5.2 report the results of the simulation study. For each benchmark are plotted the mean transaction response time, i.e. the average time since the TO-broadcast of a transaction till its commitment, and the CPU utilization of single replica (we recall that all the replicas have the same hardware architecture and process the same sequence of transactions in the same order, so, fixing a time slice, each replica in the system will produce the same CPU utilization) for both AGGRO and OPT. These performance issues are measured as a function of the transactions' arrival rate and the batching factor β. We recall that AGGRO is tailored for networks ensuring spontaneous order so, in the simulation we focus on the case of absence of mismatches between the optimistic and final delivery.

By the plots, we can draw two main considerations.
First, AGGRO allows achieving a striking increase in terms of maximum sustainable throughput by a factor that, independently of the considered settings, fluctuates around the 6x value. The reason underlying this impressive performance gain is associated with AGGRO's ability to make effective use of the locally available computational resources. Specifically, the average CPU utilization with OPT ranges between 5% and 20%, depending on the considered benchmark, even when the system has reached the saturation point. Conversely, as the load increases, AGGRO succeeds in fully utilizing the whole set of cores (that we recall being equal to 8 in this study) locally available at each replica. This depends on the fact that the concurrency control policy adopted by OPT results way too conservative, inducing very long (relatively speaking) periods of stall in the processing activities. It is interesting to highlight that AGGRO's performance gains are achieved despite the rate of aborted transactions grows significantly at high load (getting over 50% close to the saturation point). This is a direct consequence of the aggressively optimistic approach to concurrency control undertaken by AGGRO, which opts for incurring the risk of (user transparent) cascading aborts in order to achieve maximum overlap between processing and communication.
It is also interesting to note how, at low load, e.g. around 1000 transactions per second, the performance of OPT rapidly degrades as the batching factor β

increases. This phenomenon is particularly manifest for the List benchmark, where the mean transaction response time when β=8 is around 75% larger than in absence of batching (β=1). In fact, when a batch of transactions is Opt-delivered, in scenarios characterized by non-negligible conflict probability, they are likely to create convoys.
In OPT, only the first transaction of a convoy is immediately processed, whereas the remaining ones stall till the final order notification. Conversely, in AGGRO, the whole batch of delivered transactions is very likely to have been completely processed in the interval since the optimistic to the final order notifications. This makes the transaction response time at low load almost insensitive to the variation of the batching factor (at least for the explored values of β).

```
List<Transaction> TODelivered,OptDelivered;

upon Opt-deliver(Transaction T_i) do
  OptDelivered.add(T_i); // transaction $T_i$ is added at the tail of the OptDelivered list
  ActivateTransaction(T_i);

void ActivateTransaction(Transaction $T_i$) { ... }

void Write(Transaction $T_i$, DataItem X, Value v)
  if ($\nexists X \in$ WriteSet$_{T_i}$)
    WriteSet$_{T_i}$.add(X,v);
    MarkAsWip(X,$T_i$);
    $\forall$ Transaction $T_j$ s.t. $T_i \overset{OAB}{\rightarrow} T_j${
      if ($X \in$ ReadSet$_{T_j}$ and $T_k \in$ ReadFrom$_{T_j}$ with $T_k \overset{OAB}{\rightarrow} T_i$) event Abort($T_j$);
  else update $X$ within WritSet$_{T_i}$; // if X already in WriteSet it gets over-ridden

DataItemValue Read(Transaction $T_i$, DataItem X)
  if ($X \in$ WriteSet$_{T_i}$) return WriteSet$_{T_i}$.get(X).value;
  if ($X \in$ ReadSet$_{T_i}$) return ReadSet$_{T_i}$.get(X).value;
  until (MarkedAsWip($X, T_j$) s.t. $T_j \overset{OAB}{\rightarrow} T_i$) suspend;
  select version of $X$ wrote by $T_j = max\{T_j | T_j \overset{OAB}{\rightarrow} T_i\}$;
  ReadFrom$_{T_i}$.add($T_j$);
  return selected version of $X$

void Complete(Transaction $T_i$)
  $\forall X \in$WriteSet$_{T_i}$ atomically do
    UnmarkAsWip($X, T_i$);
    setCompleteVersion(X,$T_i$);
  set $T_i$ complete;

upon Commit(Transaction $T_i$)
  $\forall X \in$WriteSet$_{T_i}$ atomically do
    UnmarkAsWip($X, T_i$);
    setCommittedVersion($X$,$T_i$);
  TODelivered.remove($T_i$);

upon Abort(Transaction $T_i$) do
  $\forall$ $T_j$ s.t. $T_i \in$ ReadFrom$_{T_j}$ event Abort($T_j$);
  if ($T_i$ is complete) unsetCompleteVersion($X$,$T_i$);
  else UnmarkAsWip($X$,$T_i$);
  release transactional context of $T_i$;
  new thread(ActivateTransaction($T_i$));

upon TO-deliver(Transaction T_i ) do
  atomically do
    OptDelivered.remove(T_i);
    TODelivered.add(T_i);
  until $T_i$ not complete or $\exists T_j$ s.t. $T_j \overset{OAB}{\rightarrow} T_i$: suspend;
  if ($\exists X$ s.t. $X \in$ ReadSet$_{T_i}$, $X \in$WriteSet$_{T_j}$, $\neg(T_i \overset{OAB}{\rightarrow} T_j)$ event Abort($T_i$);
  else event Commit(T_i);
```

Figure 5.1: Behavior of AGGRO's SCC.

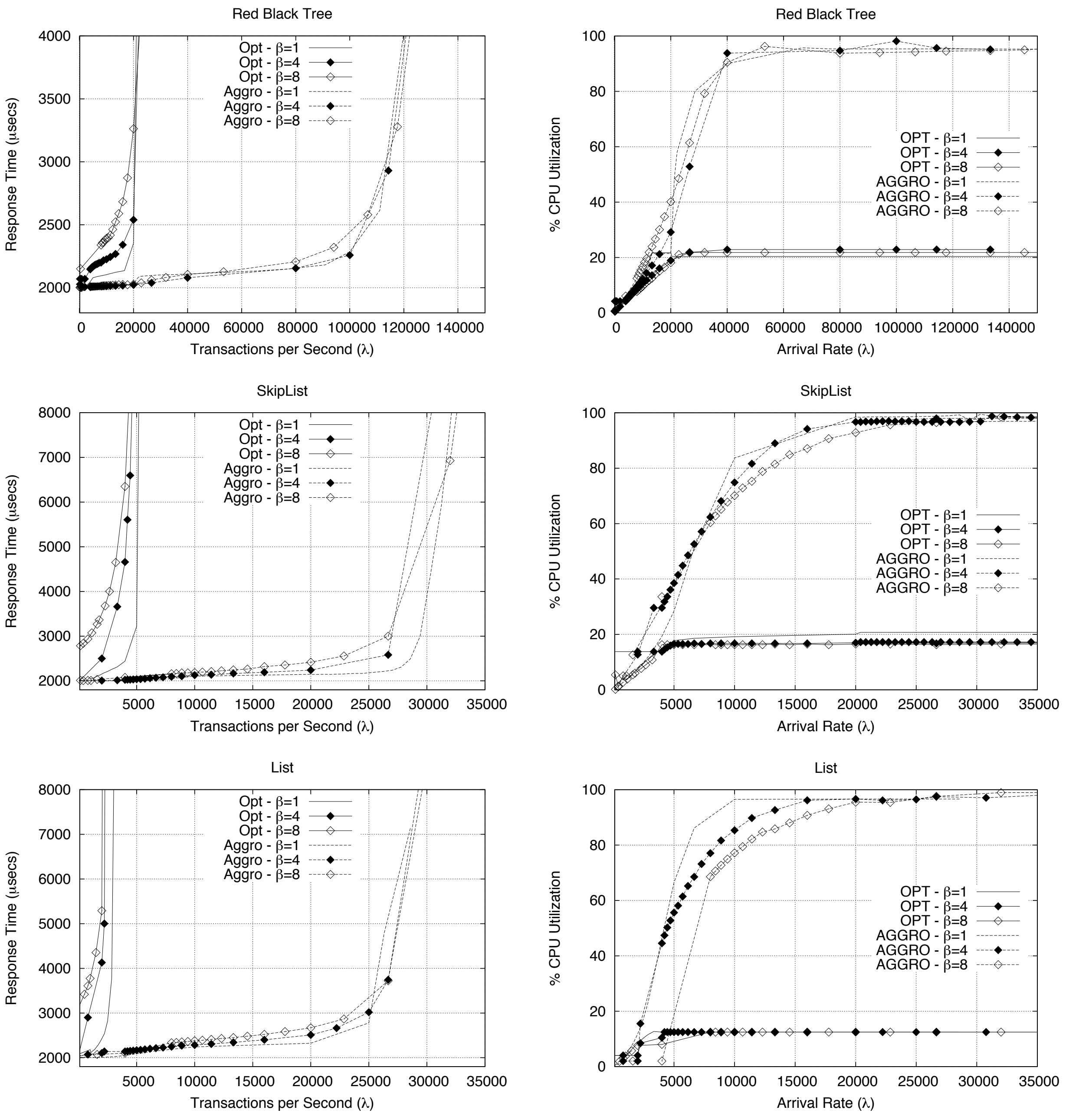


Figure 5.2: AGGRO vs OPT Performance Comparison.

Chapter 6

Optimality of Speculative Replication in Generic Networks

In this chapter a theoretical framework and an associated protocol are presented for actively replicated transactional systems relying on the Optimistic Atomic Broadcast (OAB) service.

In Section 4.2 we have shown how the existing OAB-based solutions only permit the parallel activation of optimistically delivered transactions that are known not to conflict with each other [48, 59] and, such a choice, can seriously constrain the achievable degree of parallelism. Additionally, existing approaches require a-priori knowledge of both read and write sets associated with incoming transactions, raising the non-trivial problem of systematically predicting data access patterns, which may force to significant over-estimations of the likelihood of transaction conflicts, with an obvious negative impact on concurrency. The above drawbacks are exacerbated in a number of realistic scenarios such as large scale geographical replication, where guessing the final order can be very challenging [56], or in systems where the ratio between the communication delay and the computation granularity is very large, such as Transactional Memories [69].

The approach proposed in this chapter addresses these challenges by exploring the use of speculative transaction processing, hinted in Section 2.3, motivated by the widespread use of multi-core parallel machines whose computational power can be fully unleashed using such an approach. We investigate, from a theoretical perspective, the issues related to the adoption of a speculative approach to replication of transactional systems, which we call *Speculative Transactional Replication* (STR).

The idea underlying STR is rather simple: exploring multiple serialization orders for the optimistically delivered transactions, letting them observe the snapshots generated by conflicting transactions, rather than pessimistically blocking them waiting for the outcome of the coordination phase. We note that this approach significantly differs from the one illustrated in Chapter 5 since AGGRO does not speculatively process transactions in multiple serialization orders. Rather, it early aborts and restarts transactions in order to align them to the a serialization order at anytime compliant with the optimistic delivery order. Overall, one significant difference between the two approaches is that STR is suited for networks not necessarily guaranteeing spontaneous ordering, hence generic networks.

We frame the problem in a model in which we do not assume the availability of any a-priori information on the set of data items to be accessed by the transactions (in either read or write mode), and in which we allow data access patterns to be influenced by the state observed during the execution. Next, a set of correctness and optimality criteria for the speculative exploration of the permutations of the optimistically delivered transactions are discussed, demanding the *on-line* identification of all and only the transaction serialization orders that would cause the optimistically executed transactions to exhibit distinct outcomes. Also, a specification of an optimal STR protocol is presented. It relies on a novel graph-based construct, named *Speculative Polygraph* (SP), which encodes information on the conflict relations developed during the speculative execution of transactions. SPs are designed to exactly identify what subsets of the speculatively available data item versions would be visible in any view-serializable execution, thus ensuring optimality of the STR protocol, in terms of completeness and non-redundancy of the set of explored speculative serialization orders.

To assess the viability of our proposal, we also report simulation results based on the data access patterns produced by a well-known benchmark for Software Transactional Memories [42].

6.1 System Model

The system model for STR is compliant with the one presented in Chapter 3. The only extensions are discussed below.

We assume that presence of a method `Complete`, used to explicitly inform the Speculative Transaction Manager (SXM) about the completion of the execution of a transaction. Further, the the `getResult` method is also provided, which allows the retrieval of the result generated by (completed) transactions.

Finally, the manipulation of the versions of the data items stored by TS occurs via the following primitives:

– `addSVersions(`T_i^j`)`, which makes the write set of a completed transaction T_i^j visible;

- `removeSVersions(`T_i^j`)`, which removes from TS the write set of T_i^j;
- `commitSVersions(`T_i^j`)`, which commits the write set of transaction T_i^j by replacing the corresponding existing committed version of any data item.

6.2 Problem Formalization

From the perspective of the replicated transactional system as a whole, our target correctness criteria is classic 1-copy serializability [9], which ensures that a transaction execution history $\mathcal{H}$ across the whole set of replicated processes Π is equivalent to a serial transaction execution history on a non-replicated system. More specifically, we are interested in *view serializability* [9, 58] defined as a property of $\mathcal{H}$ such that, for any prefix $\mathcal{H}'$ of $\mathcal{H}$, its committed projection $C(\mathcal{H}')$ (obtained from $\mathcal{H}'$ by deleting all operations not belonging to transactions committed in $\mathcal{H}'$) is *view equivalent* to some serial history. Roughly speaking, view equivalence of two histories $\mathcal{H}_1$ and $\mathcal{H}_2$ is defined as the property by which, for any data item X:

(i) if T_i reads X from T_j in $\mathcal{H}_1$, then T_i reads X from T_j also in $\mathcal{H}_2$;

(ii) for each data item X, if X^w is the final written value of X by T_i in $\mathcal{H}_1$, then it is also the final value written value of X by T_i in $\mathcal{H}_2$.

We now introduce the notion of optimality for a speculatively replicated transactional system. This is done by formalizing a set of properties jointly ensuring the consistency of speculative transactions, as well as the exploration of *all and only* the speculative serialization orders in which the transactions observe *distinct* states of the transactional store.
Let $\Sigma = \{T_1, \ldots, T_n\}$ be the set of Opt-delivered but not yet TO-delivered transactions. We denote with with $\Sigma' = \{T_1^1, \ldots, T_1^k, \ldots, T_n^1, \ldots, T_n^m\}$ the set of the corresponding speculative transactions that have run to completion, namely that have fully executed their sequence of read and write operations but have not been committed yet. We say that the system is quiescent if the OAB service stops Opt-delivering and TO-delivering transactions, which ensures that Σ does not change over time. Finally, let us denote with $\pi(\Sigma)$ the set of all the possible permutations of Σ.

We say that a Speculative Transactional Replication (STR) protocol is optimal if it guarantees the following properties:

- **Consistency:** *the history of execution of each speculative transaction in Σ' is view serializable.*

– **Non-redundancy:** *no two sibling transactions in Σ' observe the same snapshot.*

– **Completeness:** *if the system is quiescent then for every permutation $\sigma \in \pi(\Sigma)$ and for every transaction $T_i \in \Sigma$, there eventually exists a speculative transaction $T_i^j \in \Sigma'$ that executes on (i.e. observes) the same snapshot that would have been produced by sequentially executing all the transactions preceding T_i in σ.*

The non-redundancy property filters out trivial solutions based on the exhaustive enumeration of every possible permutation of the Opt-delivered transactions for the construction of plausible serialization orders. Such an approach would certainly enumerate the permutation that will be eventually established by the final TO-deliver order, thus providing completeness. On the other hand, denoting with n the number of Opt-delivered but not yet TO-delivered messages, this approach would *always* require executing $\sum_{i=1...n} \frac{n!}{(n-i)!} = \Theta(n!)$ speculative transactions (i.e. the number of nodes of a permutation tree for a set of cardinality n), independently of the conflict relations actually developed by the corresponding transactions. This would likely cause the useless exploration of a (possibly very large) number of redundant serialization orders in which transactions execute along identical trajectories, thus observing the same snapshots and externalizing the same results.

As a final note, the assumption on the system's quiescence in the specification of the completeness property is a formal requirement ensuring that Σ does not change over time. This assumption allows excluding scenarios where processes are never provided with sufficient time to complete the speculative exploration of the set of distinct serialization orders admitted by the STR protocol. In other words, completeness is not intended as a property related to the timing of actions by the STR protocol, but related to its own execution logic.

6.3 An Optimal STR Protocol

This section is organized as follows. In Section 6.3.1 we provide a global overview of the proposed STR protocol. In Section 6.3.2, we specify the set of primitives used within the protocol's pseudo-code to abstract over the activation and termination of threads, and the management of the data items' versions by TS. Section 6.3.3 and Section 6.3.4 present, respectively, the pseudo-codes of SXM and SCC. Finally, in Section 6.3.5, we prove that the presented protocol is optimal according to the STR optimality properties specified in Section 6.2.

6.3.1 Protocol Overview

In the proposed STR protocol, each replica immediately starts processing transactions as soon as these are optimistically delivered by the OAB service. The issue of generating a speculated set of different serialization orders is tackled by the SCC layer, which dynamically tracks the dependencies developed during the execution of the transactions through a novel graph based construct which we name Speculative Polygraph (SP).

SPs can be viewed as an extension of Papadimitriou's polygraphs [58], which were introduced to determine view-serializability for (non-speculative) transaction histories. More in detail, SCC exploits knowledge on conflict dependencies tracked by a SP associated with each transaction T_i^j in order to determine which subset $V(X)$ of the currently available versions of a data item X, can be returned by T_i^j upon its n-th read operation (on data item X) without violating view-serializability, and given the history of execution of its former $n-1$ read operations. By forking from T_i^j a number of $|V(X)|-1$ sibling transactions, and delivering to each forked transaction, and to the parent, a different value of X in the set $V(X)$, SCC completely covers all the distinct execution trajectories that T_i^j could undertake by letting the read operation return a different value (though representative of some view-serializable execution history) among those already available for X.

This read-triggered forking mechanism is however insufficient to ensure the complete exploration of differentiated speculative serialization orders. In fact, new versions of a data item X can become available after the execution of the read on X by transaction T_i^j. This happens if some transaction writes on X after T_i^j carries out its read on X. To tackle these situations, the SCC relies on an additional a-posteriori transaction re-spawning mechanism. The re-spawning of transaction T_i^j, which leads to the re-start of a new sibling transaction, say, T_i^k, is triggered as soon as a transaction T completes its execution and makes available a new version of a data item X, which could have been visible by T_i^j at the time of its read on X in some legal sequential history. SCC serves all the reads issued by the re-spawned transaction T_i^k, up to the read on data item X, by returning the same values already observed by transaction T_i^j. Since we are assuming snapshot determinism, this implies that T_i^k will "clone" the execution of T_i^j up to the read on X, which, conversely, will return the version created by transaction T.

As hinted in Chapter 3, we made the choice to make visible the data item versions written by a transaction only when the transaction completes its execution, rather than as soon as the write operation is completed. In the re-spawning mechanism, this avoids scenarios in which a transaction that writes multiple times the same data item causes the activation of new speculative transactions that observe "intermediate" values that would have never been visible in any view-serializable history. Such a phenomenon would in fact lead to violations of the Consistency property.

Note that while such phenomena be ultimately tackled by aborting the transactions that have observed not serializable snapshots, they would not only lead to suboptimal usage of the available computing resources (which would be wasted executing doomed speculative transactions), but could also lead the transactional logic to generate serious anomalies in contexts, such as transactional memories [43], where the transactional system is not fully isolated by the external environment [39].

6.3.2 Primitives and Notations used in the Pseudo-code

We assume that the business logic associated with a transaction T_i is activated via the `startSXact` and `startNonSXact` primitives. These take as input parameter a Transaction instance T_i and activate a new thread which starts a new transaction T_i^j (where j is a freshly generated specId identifier) in a speculative and, respectively, non-speculative mode. As it will be further discussed in the following, a transaction T_i is activated in non-speculative mode only if its final TO-order has already been established and all the transactions that precede T_i in the final TO-order have already been committed. Otherwise, the transaction gets activated in speculative mode.

We associate each transaction with an instance of the class Transaction and abstract over the implementation of the application-level business logic by assuming that, whenever the thread executing a transaction issues a read/write operation, the read/write method of the corresponding transaction's object is invoked.

6.3.3 Speculative Transaction Manager

The Speculative Transaction Manager (SXM) intercepts the application level's transactional requests, and interacts with the Optimistic Atomic Broadcast (OAB) service and with the Speculative Concurrency Control (SCC) layer to consistently orchestrate the set of speculatively activated transactions.

The pseudo-code for SXM is reported in Figure 6.1. SXM relies on three data structures, namely the *ActivatedXacts*, *CompletedXacts* and *CommittedXacts* sets, which contain references to T_i^j transactions within the corresponding execution stages. To simplify the pseudo-code, we assume that whenever a transaction is started, forked or respawned it is inserted into the *ActivatedXacts* set, and that it is inserted into the *CompletedXacts* when the `completed` method is invoked. Also, whenever a transaction is aborted, it is removed from the *ActivatedXacts* and *CompletedXacts* sets. When it is committed, it is removed from the *CompletedXacts* set and is added to the

```
Set<Transaction> ActivatedXacts;
Set<Transaction> CompletedXacts; // CompletedXacts ⊆ ActivatedXacts
Set<Transaction> CommittedXacts;

Result invoke(TransactionalLogic T, inputParams p) do
 Transaction $T_i$ = new Transaction(T, p, getNewXactID());
 OAB.TO-broadcast($T_i$);
 wait $\exists T_i^j \in CommittedXacts$
 return $T_i^j$.getResult();

upon Opt-Deliver(Transaction $T_i$) do
 startSXact($T_i$);

upon TO-Deliver(Transaction $T_i$) do
 if ($\exists T_i^j \in$CompletedXacts s.t. $T_i^j$.validateTransaction())
  $T_i^j$.commit(); //the commit method aborts any sibling transaction of $T_i^j$
 else
  $\forall T_i^j \in$ CompletedXacts do
   $T_i^j$.abort(); //any completed $T_i^j$ is invalid
  if ($\nexists T_i^k \in ActivatedXacts$)
   if (($\forall T_r$ s.t. ($T_r \rightarrow T_i$)$\in$OAB.TO-DeliveredMsgs() $\exists T_r^s \in$CommittedXacts))
    startNonSXact($T_i$);
   else
    startSXact($T_i$);
```

Figure 6.1: Pseudo-code for the Speculative Transaction Manager.

CommittedXacts set.

When the application calls the `invoke` method, SXM marshals a message containing the input parameters specified by the application, generates a unique transaction identifier through the `getNewXactID` primitive (which we denote with i in the pseudo-code) and TO-broadcasts the transaction through the OAB service. Next it waits for the commitment of a transaction T_i^j in order to return the associated result (retrieved through the `getResult` primitive) to the overlying application.

The activities of SXM are also triggered by two additional events, namely Opt-deliver and TO-deliver of a transaction T_i. In the former case, SXM invokes the `startSXact` primitive with T_i as input in order to start a new transaction instance T_i^j which will be executed in speculative mode and will be added to the *ActivatedXacts* set.

On the other hand, upon TO-deliver of transaction T_i, SXM checks whether there exists an already completed transaction T_i^j that successfully passes the validation phase, which is meant to verify whether T_i^j accessed a snapshot consistent with the one produced by sequentially executing all the transactions that precede T_i in the final order (see Section 6.3.4 for details). If at least a transaction T_i^j is successfully validated, it gets committed, causing the abort of any of its sibling transactions, see Figure 6.5. Otherwise, whichever completed transaction T_i^j is aborted, causing the cascading abort of any other transaction

exhibiting (possibly indirectly) a read-from dependency (see Section 6.3.4 for further details).
Next, SXM checks whether there is no other transaction T_i^j currently active. In such a case, a new transaction T_i^k needs to be activated through either `startNonSXact` or `startSXact` depending on whether the transactions that precede T_i according to the final order have all been committed or not. In such a case, in fact, the freshly activated transaction can safely read the current committed snapshot. On the other hand, if there is any transaction preceding T_i in the final order that has not been committed yet, rather than waiting for the commitment, SXM activates T_i^k in speculative mode. This choice is a further expression of optimism about the absence of conflicts among T_i^k and any other not yet committed transaction preceding T_i^k in the final order.

6.3.4 Speculative Concurrency Control

To determine the set of speculative serialization orders according to which transactions need to be executed, SCC relies on a novel graph-based construct, which we call Speculative Polygraph (SP). SPs are inspired by Papadimitriou's polygraphs, introduced in [58] to test view-serializability of a non-speculative history $\mathcal{H}$ and whose definition we briefly recall in the following.

Polygraphs. A polygraph $P = (N, A, B)$ is a directed graph (N, A), whose nodes are defined by the set N and whose arcs are defined by the set A, augmented with a set B of so called *bipaths*. Each bipath is a pair of arcs $< (T'' \to T'), (T' \to T) >$ (where $(T \to T'') \in A$), not necessarily present in A. De-facto, a polygraph is a compact representation of a family of directed graphs (digraphs) $\mathcal{D}(N, A, B)$. A digraph (N, A') is in $\mathcal{D}(N, A, B)$ if and only if $A \subseteq A'$, and, for each bipath $(a_1, a_2) \in B$, A' contains at least one of the arcs a_1 and a_2.

Polygraphs capture partial order relations in a history of transactions, and the polygraph $P(\mathcal{H})$ associated with a history $\mathcal{H}$ is constructed according to the following two rules:

(i) whenever a transaction T reads some data item X from transaction T', the arc $(T' \to T)$ is added in A;

(ii) if a third transaction T'' also writes X, then the bipath $< (T \to T''), (T'' \to T') >$ is added to B.

In other words, each arc (T', T) in A keeps track of the direct read-from relation between transactions T and T', whereas a bipath $< (T \to T''), (T'' \to T') >$ means that since also T'' writes X, it can not be between T' and T, but must either precede T' or follow T.

Based on the above definition of polygraph, Papadimitriou defines a polygraph as acyclic iff there is at least an acyclic digraph in $\mathcal{D}(N, A, B)$ and proves that a history $\mathcal{H}$ is view-serializable iff its polygraph $P(\mathcal{H})$ is acyclic [58].
We show in Figure 6.2.a and 6.2.b the polygraphs associated with, respectively, the following two sequential transaction histories $\mathcal{H}_1$=$\{T_1, T_2, T_3^0\}$ and $\mathcal{H}_2$=$\{T_2, T_1, T_3^1\}$, where T_1 writes on data items X and Y, T_2 writes on data items Y and Z, whereas T_3^0 and T_3^1 read the same data items, namely X, Y and Z (returning different values for Y given that $\mathcal{H}_1$ and $\mathcal{H}_2$ serialize in a different order transactions T_1 and T_2).
As in [58], in Figure 6.2 a bipath is denoted by adding a circular edge on its common node. It can be easily verified that the only acyclic digraph associated with the polygraph in Figure 6.2.a is obtained by selecting the edge $(T_1 \rightarrow T_2)$ of the bipath centered on T_1, whereas the only acyclic digraph associated with the polygraph in Figure 6.2.b is obtained by selecting the edge $(T_2 \rightarrow T_1)$ of the bipath centered on T_2.

Speculative Polygraphs. The unfeasibility of conventional polygraphs to reason on view-serializability of a speculative transaction history appears manifest if one considers that the simultaneous coexistence within a polygraph of two sibling transactions, representative of irreconcilable serialization orders (such as transactions T_3^0 and T_3^1 in Figure 6.2), can corrupt the polygraph by introducing cycles that might render it useless. An example of such a problem is shown in Figure 6.2.c, which shows the polygraph obtained by merging the polygraphs in Figure 6.2.a and 6.2.b. It is straightforward to verify that every directed graph associated with this polygraph is cyclic. This is of no surprise considering that the polygraph keeps track of every partial order relation in a history that contains transactions that assume opposite serialization orders for T_1 and T_2.

Speculative Polygraphs address exactly these problems. Unlike Papadimitriou's polygraphs, which are representative of a *whole*, and non-speculative, history, SPs are designed to keep into account the history of execution as perceived by each speculative transaction. Roughly speaking, the SP of a

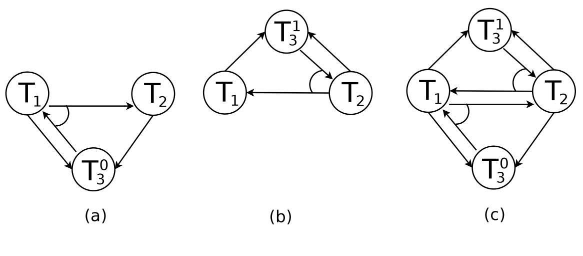


Figure 6.2: Polygraphs Associated with Histories $\mathcal{H}_1$ and $\mathcal{H}_2$.

transaction T_i^j is dynamically generated by selectively merging only the polygraphs of those speculative transactions T^* that i) conflict, either directly or indirectly, with T_i^j, and ii) such that exists at least a (non-speculative) serialization order which allows both T^* and T_i^j to coexist.
More formally, we define the speculative polygraph associated with transaction T_i^j, denoted as SP(T_i^j), as a triple (N, A, B) where:

- N is a set of nodes, each one representative of some speculative transaction.
- A is a set of, so called, *merging edges* denoted as $(T_r^s \circledast \rightarrow T_i^j)$, with $T_r^s, T_i^j \in$N, and where the notation $T_r^s \circledast$ means that we are not just adding an edge between T_r^s and T_i^j, but also merging SP(T_r^s) with SP(T_i^j), and linking them via a (plain) edge from T_r^s to T_i^j.
- B is a set of, so called, *asymmetric bipaths*, denoted as $< (T_u^v \circledast \rightarrow T_r^s), (T_i^j \rightarrow T_u^v) >$, with $T_r^s, T_i^j \in$N, where the first of the two arcs is a merging edge linking SP(T_u^v) with SP(T_i^j) through the plain edge $(T_u^v \rightarrow T_r^s)$, and the second one, namely $(T_i^j \rightarrow T_u^v)$, is a plain edge between the nodes T_i^j and T_u^v.

A speculative polygraph SP(T_i^j)=(N, A, B) generates a family of directed graphs $\mathcal{D}(SP(T_i^j))$, where each directed graph $\delta \in \mathcal{D}(SP(T_i^j))$ is obtained by (1) recursively replacing any merging arc, say $(T_r^s \circledast \rightarrow T_t^u)$, of A and B with the speculative polygraph SP(T_r^s)$\cup$($T_r^s \rightarrow T_t^u$), and (2) for each asymmetric bypath $< a_1, b_1 >$ present after the previous “merging phase”, selecting either a_1 or b_1.

A speculative polygraph SP(T_i^j) is initialized, at transaction creation time (see Figure 6.3), by serializing T_i^j after the most recently committed transaction (according to the TO-deliver order), say T_c^d, through a merging edge. This has the effect of setting a barrier, in terms of minimum logical time, for the visibility of data item versions observable via read operations by T_i^j. The speculative polygraph associated with transaction T_i^j is then used to determine whether the k-th read operation of T_i^j can return a given version X^s (possibly created by a not yet committed transaction). Indeed, letting the k-th read of T_i^j return a specific version X^s, rather than any other available version, corresponds to speculating on a set of possible serialization orders for T_i^j. In order to ensure the consistency of the k-th read by a transaction with its current execution history, it is however necessary that at least one of the speculative serialization orders associated with the reading of version X^s results “compatible” with those already determined by having executed the preceding k-1 reads. In this case we say that X^s is speculatively visible to T_i^j.

To determine if T_i^j may speculatively view a data item version X^s based on its current execution history, its Speculative Polygraph, $\mathrm{SP}(T_i^j)$, is updated by:

(R.1) adding a merging edge from the creator of version X^s to T_i^j, namely $(X^s.creator \circledast \rightarrow T_i^j)$

(R.2) adding, for each other available version $X^{s'}$, an asymmetric bipath: $< (X^{s'}.creator \circledast \rightarrow X^s.creator), (T_i^j \rightarrow X^{s'}.creator) >$.

Version X^s is considered speculatively visible iff there exists at least one directed graph $\delta \in \mathcal{D}(\mathrm{SP}(T_i^j))$ such that:

(C.1) δ is acyclic;

(C.2) δ does not contain two sibling transactions T_a^r, T_a^q both serialized before T_i^j, or, formally, $\nexists T_a^r, T_a^q \in \delta$ such that $(T_a^r \rightarrow T_i^j) \in \delta^*$ and $(T_a^r \rightarrow T_i^j) \in \delta^*$, where with δ^* we denote the transitive closure of δ.

If for a $\delta \in \mathcal{D}(\mathrm{SP}(T_i^j))$ both conditions C.1 and C.2 hold, we also say that δ is *valid*. The rationale underlying the above rules is to ensure that, whenever a transaction T_p^q is serialized before T_i^j, the speculative polygraphs of both transactions are recursively merged. This ensures that the resulting $\mathrm{SP}(T_i^j)$ keeps a complete track of any conflict relation among the transactions that generated the snapshot seen by T_i^j. On the other hand, whenever T_i^j issues a read operation on a data item for which exists a version created by a transaction T_t^u, whose $\mathrm{SP}(T_u^t)$ cannot be merged with $\mathrm{SP}(T_j^i)$ without generating cycles in any $\delta \in \mathcal{D}(\mathrm{SP}(T_j^i))$([1]), rule R.2 allows to serialize T_t^u after T_i^j through a plain edge *without* requiring to merge the polygraph of T_t^u. This prevents corrupting $\mathrm{SP}(T_i^j)$ by blindly incorporating into it the history of transactions associated with incompatible speculative serialization orders, and whose writes shall never result visible to T_i^j. On the other hand, by serializing T_t^u after T_i^j through the plain edge of an asymmetric bipath, we can still detect cyclic dependencies involving transactions not serializable before T_i^j in $\mathrm{SP}(T_i^j)$. Finally, condition C.2 avoids reading inconsistent snapshots generated by an execution history which serializes (at least) a pair of different sibling transactions T_a^r, T_a^q before T_i^j, as clearly, in any serial history a transaction T_a can be committed in a single serialization order.

Transaction's Data Structures. Each speculative transaction T_i^j is associated with an instance of the Transaction class which keeps track of the following state information (see Figure 6.3):

[1]This may happen for instance if the polygraphs have two transactions in common, but ordered in an opposite manner.

(i) *id* and *specId*, which are set, respectively, to the values i and j when the Transaction object associated with T_i^j is created;

(ii) SP, which stores the speculative polygraph of T_i^j;

(iii) *RS*, namely T_i^j's read set, which is organized as an array whose n-th entry records the identity X of the data item read during the n-th read operation, the version of X read, and a copy of T_i^j's Speculative Polygraph right *before* the read took place;

(iv) WS, T_i^j's write set;

(v) two boolean variables, *speculative* and *respawned*, reflecting, respectively, if the transaction was activated speculatively, and whether it was respawned;

(vi) $readOpCounter$, namely a counter that keeps track of how many read operations have been issued up to date by the transaction;

(vii) *generatingRead*, which stores a value different from 0 only if T_i^j has been activated through the fork/spawn of some sibling transaction T_i^j, in which case it keeps track of the T_i^k's read operation that has triggered the forking/spawing of T_i^j.

Write/Read Operations. The logic for the management of write operations (see the `write` method in Figure 6.3) is extremely simple: the transaction simply logs the identity of the target data item, as well as its new value, into its local WS variable, which can be seen as the transaction's "private workspace".

Concerning read operations (see the `read` method in Figure 6.3), SCC first checks whether the transaction already wrote the same data item for which it is issuing the read. In the positive case, it just returns the corresponding value stored in its write set. Next, it verifies whether the transaction has been activated in non-speculative mode. As anticipated in Section 6.3.3, T_i^j is activated in non-speculative mode only if T_i has already been TO-delivered and all the transactions preceding T_i in the final order have already committed. Hence, any read executed by a non-speculative transaction can safely return the most recently committed version. On the other hand, if the read operation is issued by a speculative transaction, SCC determines, through the `getAllVisibleVersions` method, what subset of the available versions is speculatively visible to the reading transaction based on its execution history.

The `isVisible` method returns either a $\perp$ value if the target data item version is not speculatively visible. Otherwise, it returns the SP of the reading

```
class Transaction {
 int id, specId;
 Transaction $T_c^d$ =max{$T_a \in$ OAB.TO-deliveredMsgs() s.t. $\exists T_a^b \in$CommittedXacts}
 SpeculativePolygraph SP = ( $T_c^d \circledast \rightarrow T_{id}^{specId}$ )
 List<DataItemID, DataItemVersion, SpeculativePolygraph> RS;
 Set<DataItemID,Value> WS;
 boolean speculative, respawned;
 int readOpCounter=0, generatingRead=0;

 void write(DataItemID X,Value v)
  WS.store(X,v); // if X already exists in WS it gets overridden

 Value read(DataItemID X)
  readOpCounter++;
  if (< $X, v$ >∈WS) return $v$;
  if (¬speculative)
   return X.mostRecentCommitted();
  if (respawned ∧ readOpCounter≤generatingRead)
   return RS[readOpCounter].getValue();
  Set<DataItemVersion, SpeculativePolygraph> versions = getAllVisibleVersions(X);
  [$X^i, newSP$] = versions.pop();
  ∀ <DataItemVersion $X^j$, SpeculativePolygraph SP'>∈versions do
   forkSibling();
   if (I am the just forked transaction)
    RS[readOpCounter] = <X,$X^j$,SP>;
    SP = SP';
    generatingRead = readOpCounter;
    return $X^j$;
  RS[readOpCounter] = <X,$X^i$,SP>;
  SP = newSP;
  return $X^i$;
 . . .
}
```

Figure 6.3: Pseudo-code for the Speculative Concurrency Control (1).

transaction T_i^j updated to reflect the outcome of the read. Next, the transaction picks one of the selected visible versions, say X^s, to use it as the return value for the on-going read, logs it, together with its current SP, within its read set, and successively replaces its SP with the one updated by the `isVisible` method. Finally, before returning X^s, T_i^j forks, for each other visible version X^t, a sibling transaction whose execution will proceed in parallel with T_i^j, after returning X^t for the ongoing read (and after having correspondingly updated its read set and SP).

Transaction's Completion. When T_i^j completes its execution, it is added to the Completed set and the `Complete` method (see Figure 6.4) is invoked. Then, if T_i is already TO-delivered and exists some transaction T_p that precedes T_i in the final order, which has not yet been committed, the `completed` method simply ends, delegating the task of attempting to commit T_i^j to any T_p^s that will subsequently enter the commit phase. Conversely, if all the transactions T_p preceding T_i in the final order have already been committed, and T_i^j is either non-speculative or successfully passes the validation phase, T_i^j is committed

```
class Transaction {
...
void Complete() // invoked when the transaction logic terminates its execution
 if ($T_{id}$ $\notin$OAB.TO-DeliveredMsgs())
  TS.addSVersions($T_{id}^{specId}$);
  handleWriteAfterReadConflicts();
 else
  if ($\forall T_r$ s.t. ($T_r \rightarrow T_{id}$)$\in$OAB.TO-DeliveredMsgs() : $\exists T_r^s$ $\in$CommittedXacts)
   if ($\neg$speculative $\vee$ validateTransaction())
    commit();
    Transaction $T_j$ = min{$T_a \in$ OAB.TO-deliveredMsgs() s.t. ($T_{id} \rightarrow T_j$)}
    do
     if ($\exists T_j^k$ $\in$CompletedXacts s.t. ($\neg T_j^k$.speculative $\vee$ $T_j^k$.validateTransaction()))
      $T_j^k$.commit();
     else
      $\forall T_j^k$ $\in$CompletedXacts do $T_j^k$.abort();
      break;
    while($T_j = T_j.next_{\text{OAB.TO-deliveredMsgs()}}$)
   else
    abort();
    if ($\nexists T_{id}^{\cdot} \in$ ActivatedXacts) startNonSXact($T_{id}^{specId}$);

void handleWriteAfterReadConflicts()
 $\forall T_j^k \in$ ActivatedXacts s.t. ($T_j^k.RS \cap WS \neq \emptyset$) do
  let $r$ be the min index s.t. $T_j^k.RS[r].DataItemId \in WS$;
  $(X, \cdot, SP') = T_j^k.RS[r]$;
  let $X^i$ be the value of $X$ written by the current transaction;
  if (r > generatingRead)
   SpeculativePolygraph newSP = isVisible(X, $X^i$, $SP'$, $T_j^k$);
   if (newSP$\neq\perp$)
    $\forall T_j^k.RS[s] =< \cdot, \cdot, SP'' > s.t. s > r$ do
     $T_j^k.RS[s] =< \cdot, \cdot, SP'' \cup$ <($T_{id}^{specId} \circledast \rightarrow X^s$.creator), ($T_j^k \rightarrow T_{id}^{specId}$)>>;
    ReadSet newRS;
    $\forall$ $1 < t < r$ do newRS[t] = $T_j^k$.RS[t];
    newRS[r] = <X,$X^i$,newSP>;
    spawnSibling($T_j$,newSP,newRS,r);
...
}
```

Figure 6.4: Pseudo-code for the Speculative Concurrency Control (2).

and attempts to commit any completed transaction T_l^m, such that T_l follows T_i in the final order.

On the other hand, in case T_i has not yet been TO-delivered, T_i^j makes available its versions through `addSVersions`. Then it invokes the `handleWriteAfterReadConflicts` method to determine whether there is any transaction T_a^b that has read some data item also written by T_i^j, and whether the version created by T_i^j was speculatively visible for T_a^b at the time in which it executed the read.
To this end, the `isVisible` method is invoked passing as input parameter the SP (retrieved from T_a^b's read set) that T_a^b was storing at the earliest time in which T_a^b read a data item, say its $r-$th read, also written by T_i^j. Note that by retrieving the SP related to the *first* data item belonging to T_a^b's read set that

```
class Transaction {
...
Set<DataItemVersion,SpeculativePolygraph> getAllVisibleVersions(DataItem X)
 Set<DataItemVersion,SpeculativePolygraph> VisibleVersions;
 ∀$X^i \in X$.getAllVersions() do
   SpeculativePolygraph newSP = isVisible(X, $X^i$, SP, $T_{id}^{specId}$);
   if (newSP≠⊥) VisibleVersions = VisibleVersions ∪ $< X^i, newSG >$;
 return VisibleVersions;

SpeculativePolygraph isVisible(DataItem X,
    DataItemVersion $X^i$, SpeculativePolygraph currentSP, Transaction T)
 SpeculativePolygraph newSP = ($X^i$.creator⊛ →T)∪currentSP;
 ∀($X^j \neq X^i$) ∈ TS.getAllVersions(X) do
   newSP = newSP ∪ < ($X^j$.creator⊛ → $X^i$.creator) , (T→ $X^j$.creator)>;
 if ( ∃$\delta \in \mathcal{D}$(newSP) s.t. isValid($\delta$, T)) return newSP;
 else return ⊥;

void abort()
 TS.removeSVersions($T_{id}^{specId}$);
 ∀$T_l^m \in ActivatedXacts$ do
   ∀$\delta \in \mathcal{D}(T_l^m.SP)$ s.t. isValid($\delta$,$T_l^m$) do
     if (($T_{id} \to T_l^m$) ∈ $\delta^*$)
       $T_l^m$.abort();
 ActivatedXacts = ActivatedXacts \ {$T_{id}^{specId}$};
 CompletedXacts = CompletedXacts \ {$T_{id}^{specId}$};

void commit()
 TS.commitSVersions($T_{id}^{specId}$);
 ∀$T_{id}^m \in ActivatedXacts$ do $T_{id}^m$.abort(); // Abort sibling transactions

boolean isValid(DirectedGraph $\delta$, Transaction T)
 return ($\delta$ is acyclic ∧ ($\nexists$ ($T_i^j \to T$), ($T_i^k \to T$) ∈ $\delta^*$ with $j \neq k$));

boolean validateTransaction()
 ∀ <X,value,· > ∈ RS do
   if (value ≠ X.getCommittedVersion().getValue()) return false;
   else return true;
}
```

Figure 6.5: Pseudo-code for the Speculative Concurrency Control (3).

has also been written by T_i^j, we detect the earliest possible point in the execution trajectory of T_a^b that could have been affected by a write of T_i^j. Recall that in a snapshot deterministic model, if T_a^b had seen the version created by T_i^j, rather than the one returned in its execution, T_a^b might have not executed the same set of subsequent reads.

Hence, to avoid violating the Non-redundancy property, SCC respawns only a single sibling of T_a^b, say T_a^c, even if there are multiple data items in $T_i^j.WS \cap T_a^b.RS$. T_a^c will re-execute (see the `read` method in Figure 6.3) the same set of reads already performed by T_a^b, up to the read that caused its spawn. Such a read will return the value created by T_i^j and, henceforth, T_a^c will be free to evolve along its own execution trajectory. To enforce such a behavior, the first r-1 entries of the read set of the spawned transaction are set equal to the corresponding ones of T_a^b, and its r-th entry is set to reflect the read-from T_i^j. Note that T_i^j also updates, in the T_a^b's read set, every entry following the read that caused the spawning, reflecting the fact that T_a^b did not observe during the r-th read the data item version generated by T_i^j. This guarantees the completeness of the information stored by SP(T_a^b), which could be again queried in the future by the `handleWriteAfterReadConflicts` method.

In order not to incur in violations of the Non-redundancy property, we take one additional measure: if T_i^j detects that the conflict affects the $r-th$ read of transaction T_a^b, and T_a^b was activated due to a fork/respawn of some sibling transaction T_a^c occurred upon T_a^c's k-th read (i.e. the value of *generatingRead* for T_a^b is k), where $r < k$, then T_i^j avoids respawning T_a^b. In this case, in fact, since T_a^b and T_a^c exhibit the same behavior up to the $k-1$-th read, the sibling transactions spawned by T_a^b and T_a^c to deal with a conflict occurring at a read $r < k$ would observe the same snapshot. It is therefore sufficient to re-spawn exclusively T_a^c.

Commit/Abort/Validation. A transaction is considered successfully validated iff the values read by the transaction during its execution, and stored within its RS variable, coincide with the values currently stored by the committed version of the corresponding data item (see `validateTransaction` method of Figure 6.5).
The `commit` method marks the data item versions created by the committing transaction as the currently committed versions, and then triggers the abort of any of its sibling transactions through the `abort` method. When this latter method is invoked on transaction T_i^j, it first removes any data items' versions made available by T_i^j, and then triggers the cascading abort of any other transaction, say T_l^m, having a (possibly indirect) read-from dependency with the aborting transaction. This is verified by checking if for every valid $\delta \in \mathcal{D}(\text{SP}(T_l^m))$ there is a path from T_i^j to T_l^m.

Example Scenario. Let us consider an example scenario of execution of SCC associated with the following history:
$\mathcal{H}_3$= $\{ B_{T_1^0}, R_{T_1^0}(X), W_{T_1^0}(X), C_{T_1^0}, B_{T_2^0}, R_{T_2^0}(X), F_{T_2^1}, W_{T_2^0}(X), R_{T_2^1}(X), W_{T_2^1}(X)\}$
where we use the notation $B_{T_i^j}$, $R_{T_i^j}$, $W_{T_i^j}$, $C_{T_i^j}$ and $F_{T_i^j}$, to denote, respectively, begin, read, write, complete (hence not commit) and fork of a transaction T_i^j. We shall assume that the only version for each data item present in memory is the committed version, and denote with C the identifier of the last committed transaction. In Figure 6.6.a we show the state of SP(T_1^0) right after the execution of the read operation on X. Note that, in order to refer to a merging edge, and distinguish it from a plain edge, we use a dashed arrow, which in Figure 6.6.a reflects the read-from dependency of T_1^0 from C. Further,

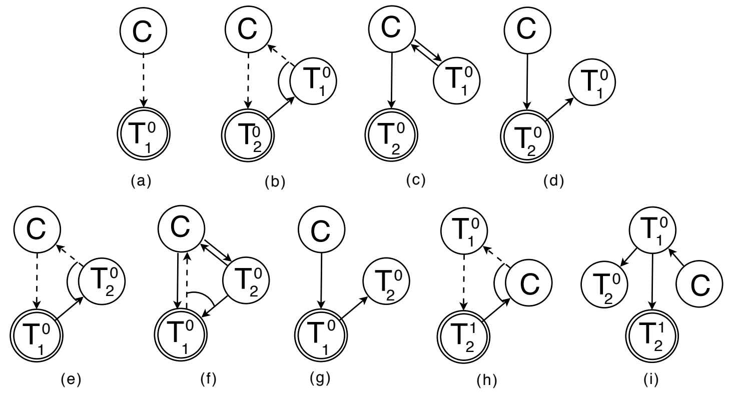


Figure 6.6: SCC Execution for History $\mathcal{H}_3$.

within the speculative polygraph SP(T_i^j), we use the convention of drawing a double circle to refer to transaction T_i^j.

Figure 6.6.b shows the state of SP(T_2^0) after the execution of the read on X, assuming that T_2^0 reads the committed version. Since T_1^0 has already completed executing (but has not been committed yet) at the time in which T_2^0 performs the read, T_2^0 finds also available the versions created by T_1^0. Thus, SP(T_2^0) contains also the asymmetric bipath $<(T_1^0\circledast \rightarrow C),(T_2^0 \rightarrow T_1^0)>$, which we denoted by adding a circular arc on the common node. Figure 6.6.c and 6.6.d shows the two directed graphs $\delta, \delta' \in \mathcal{D}(\text{SP}(T_2^0))$. As it can be seen, the directed graph in Figure 6.6.c, associated with the merging edge $(T_1^0\circledast \rightarrow C)$ exhibits a cycle, but since the directed graph in Figure 6.6.d is acyclic the committed version of X results speculatively visible to T_2^0.

In Figure 6.6.e we provide the speculative polygraph of T_1^0 after the update performed during the completion phase of T_2^0 (i.e. within the method `handleWriteAfterReadConflicts`, that adds the asymmetric bipath $b_1 =<(T_2^0\circledast \rightarrow C),(T_1^0 \rightarrow T_2^0)>$ to SP(T_1^0). In Figure 6.6.g, we show the only valid directed graph in $\mathcal{D}(\text{SP}(T_1^0))$, which serializes T_1^0 before T_2^0. In fact, as shown in Figure 6.6.f, by considering the merging edge $(T_2^0\circledast \rightarrow C)$ of b_1, and merging SP(T_2^0) with SP(T_1^0), the resulting speculative polygraph is necessarily cyclic.

Finally, we show in Figure 6.6.h the speculative polygraph of T_2^1, namely the transaction forked by T_2^0 upon the read of data item X, and which returns the version of X written by T_1^0. Figure 6.6.i shows the only directed graph in $\mathcal{D}(\text{SP}(T_2^1))$ to be acyclic that permits the speculative visibility of the version written by T_1^0.

6.3.5 Correctness and Optimality Proof

Theorem 1 *The history of committed transactions generated by the STR protocol is 1-copy serializabile.*

Proof In STR a process $p_k \in \Pi$ commits a transaction T_i^j only if the corresponding T_i has been already TO-delivered, and all the preceding transactions within the TO-deliver order have been committed. Hence, by the Global Agreement and Global Order properties of Optimistic Atomic Broadcast, it follows that every correct process $p_k \in \Pi$ commits transactions according to the same ordered sequence T_k=$\{T_{i^1}^{j^k}, \ldots, T_{i^l}^{j^k}, \ldots, T_{i^n}^{j^k}\}$, where with the notation $T_{i^l}^{j^k}$ we denote that the l-th transaction along the sequence of two distinct processes $p_k, p_{k'} \in \Pi$, must have the same id, namely i^l, but may have different specIds, respectively j^k and $j^{k'}$. Also, any process f that crashes can only commit a prefix T_f of T_k. Hence, 1-copy serializability follows from that any transaction $T_{i^l}^{j^k}$ committed by a process $p_k \in \Pi$ is deterministically validated to ensure that every read it issued returned the most recently committed version with respect to the execution of a prefix of length l of the sequence T_k.
Lemma 1 *Let* $\mathcal{R}_{T_i^j} = \{r_1(X_1), \ldots, r_{k-1}(X_{k-1})\}$ *be the set of k-1 read operations already performed by a speculative transaction* T_i^j. *The* `isVisible` *method determines that a version* X_k^s *of data item* X_k *is visible by the k-th read of* T_i^j *if and only if there exists a sequential history* $\mathcal{H}$ *in which* T_i *executes the same sequence of reads in* $\mathcal{R}_{T_i^j}$, *returning the same sequence of values seen by* T_i^j, *and* X_k^s *at its k-th read.*

Proof We proceed by showing that the `isVisible` method returns all and only the versions that would have been visible by the considered transaction if this had been executed in a speculative, but view-serializable, history. To this end we prove the equivalence between the speculative polygraph of T_i^j, namely $\mathrm{SP}(T_i^j)$, and the polygraph $P(\mathcal{H})$ representative of the non-speculative history that generates the snapshot read by T_i^j up to the current execution phase. A speculative polygraph $\mathrm{SP}(T_i^j)$ is equivalent to a polygraph $P(\mathcal{H})$, iff for each valid directed graph $\delta = (N, A) \in \mathcal{D}(\mathrm{SP}(T_i^j))$ there exists an acyclic directed graph $\delta' = (N', A') \in \mathcal{D}(P(\mathcal{H}))$.

The rules R.1 and R.2 used to construct $\mathrm{SP}(T_i^j)$ ensure that any transaction that is serialized before T_i^j (either directly, or indirectly, e.g., through the merging edge of a speculative bipath) in every directed graph $\delta \in \mathcal{D}(\mathrm{SP}(T_i^j))$ must have merged its speculative polygraph with the one of T_i^j in δ. This implies that for any $\delta \in \mathcal{D}(\mathrm{SP}(T_i^j))$ there exists a one-to-one correspondence with a directed graph $\delta' \in \mathcal{D}(\mathrm{P}(\mathcal{H}))$ where P is the polygraph associated with the history $\mathcal{H}$ obtained considering all and only the transactions that are serialized (possibly indirectly) before T_i^j based on the snapshot observed by T_i^j up to the current stage of execution. Also, since the `isVisible` method considers a data item version X^s speculatively visible to transaction T_i^j only if, by condition C.2, there are no two sibling transactions T_a^r, T_a^q preceding T_i^j, the above history $\mathcal{H}$ is guaranteed to contain at most one instance of a given transaction. Hence for any δ for which condition C.2 holds, the corresponding history $\mathcal{H}$ constructed as discussed above, is representative of a non-speculative history. Note however that the directed graph $\delta \in \mathcal{D}(\mathrm{SP}(T_i^j))$ may contain a set S of vertexes not present in its corresponding $\delta' \in \mathcal{D}(\mathrm{P}(\mathcal{H}))$, each vertex being associated with a speculative transactions T' such that:

– it created a version X^a of some data item X for which a read $r(X)$ operation was issued either by T_i^j, or by any transaction T_s^r preceding $T_i^j \in \delta$, and such that $r(X)$ returned a different version X^b, and

– its speculative polygraph $\mathrm{SP}(T')$ prevents to serialize T' before T_i^j (via a plain merging edge or through the merging edge of a speculative bipath) because, in the resulting speculative polygraph SP*, every directed graph $\delta \in \mathcal{D}(SP*)$ would not be valid.

Recalling that, by rule R.2, each time we serialize a transaction that creates a data item version X^b after a transaction that reads a data item version different from X^b, we do so via a simple (i.e. non-expanding) edge of a speculative bipath, it follows that all the vertexes in S must be sink nodes (i.e. they have no outgoing edges). However, since S is composed exclusively of sink vertexes, their presence in $\mathrm{SP}(T_i^j)$ is not influential for the generation of cycles in any directed graph $\delta \in \mathcal{D}(\mathrm{SP}(T_i^j)$. This ensures the equivalence between $\mathrm{SP}(T_i^j)$ and the polygraph $P(\mathcal{H})$ representative of the non-speculative history that generates the snapshot read by T_i^j up to the current execution phase. Also, recalling that a history $\mathcal{H}$ is view-serializable if and only if $\mathrm{P}(\mathcal{H})$ is acyclic, it follows that the `isVisible` method returns all and only the versions that would have been visible by the considered transaction if this had been executed in a speculative, but view-serializable, history.

Theorem 2 (Consistency) *The history of execution of each speculative transaction in Σ' is view-serializable.*

Proof Derives directly by Lemma 1 and from that a read operation of a speculative transaction returns a data item's version only if this is deemed visible by the `isVisible` method.

Theorem 3 (Non-redundancy) *No two sibling transactions in Σ' observe* *the same snapshot.*

Proof In this proof we denote with generatorOf(T_a^b) the transaction T_l^m, if any, that caused the activation (via either a fork or a respawn) of the speculative transaction T_a^b, and with generatorOf* the transitive closure of the generatorOf relation.

Let us now consider any two sibling transactions T_i^j and T_i^k. The following three cases are possible:

1. generatorOf(T_i^k)= T_i^j (or alternatively generatorOf(T_i^j)=T_i^k). In this case, T_i^j forked T_i^k while executing a read operation on some data item X, for which there was a set $S = \{X^1, \ldots, X^n\}$ of speculatively visible data item versions. In this case, T_i^j and T_i^k necessarily returned two different versions of data item X, therefore the two transactions necessarily observe distinct snapshots. Clearly, the same considerations apply if generatorOf(T_i^j)=T_i^k. Hence the claim follows.

2. Neither generatorOf(T_i^k)=T_i^j, nor generatorOf(T_i^j)=T_i^k, but $T_i^k \in$ generatorOf*(T_i^j) (or alternatively $T_i^j \in$ generatorOf*(T_i^k)). In this case, it is possible to determine a chain of sibling transactions (T_i^j,$T_i^{j^1}$,...,$T_i^{j^n}$,T_i^k), where:

   T_i^j =generatorOf($T_i^{j^1}$), ..., $T_i^{j^{n-1}}$ =generatorOf($T_i^{j^n}$), $T_i^{j^n}$ =generatorOf(T_i^k)

   such that each transaction in the n-th position of the chain forks the $(n+1)$-th sibling transaction of the chain. It follows that the snapshots seen by T_i^k and T_i^j differ at least for one read data item, namely the one which caused T_i^j to fork $T_i^{j^1}$. The same considerations apply by switching the roles of T_i^k and T_i^j. Hence the claim follows.

3. $T_i^k \notin$ generatorOf*(T_i^j) and $T_i^j \notin$ generatorOf*(T_i^k). With no loss of generality let us focus on T_i^j, and let T_a^b be the transaction that caused the activation of T_i^j, i.e. generatorOf(T_i^j)=T_a^b where $a \neq i$. This is only possible if T_a^b completed its execution, which encompassed the issuing of a write on data item X, after some transaction T_i^s carried out the read of X, say, as its $t-$th operation. Also, during the execution of the `completed` method by transaction T_a^b, the `handleWriteAfterReadConflicts` method must have detected that the version written by T_a^b could have been speculatively visible by T_i^s at the time it issued the read on X. This has caused the spawning of T_i^j.

   Now assume by contradiction that there exists a transaction T_i^k which sees the same snapshot of T_i^j. This implies that the first t reads of

T_i^k must return the same sequence of data items' versions returned by the first t reads of T_i^j. However, T_i^j won't fork any sibling transaction during the execution of its first t reads, being these simply replayed from T_i^j's read set. Also, no transaction T^* could cause the re-spawning of T_i^j by making available a new version of a data item read by T_i^j during any of its first t reads (this is avoided by an explicit guard in the `handleWriteAfterReadConflicts`). Hence, for T_i^k to return, during its first t reads, the same sequence of data items' versions returned by T_i^j, it means that T_i^k was forked by T_i^j during its t'-th read, where $t < t'$. In such a scenario we should have fallen in case 1 above, hence the assumption is contradicted and the claim follows.

Theorem 4 (Completeness) *If the system is quiescent then for every permutation $\sigma \in \pi(\Sigma)$ and for every transaction $T_i \in \Sigma$, there eventually exists a speculative transaction $T_i^j \in \Sigma'$ that executes on (i.e. observes) the same snapshot that would have been produced by sequentially executing all the transactions preceding T_i in σ.*

Proof Let us denote with T_c the last committed transaction by the STR protocol over time. Given that by the protocol structure no transaction can ever be committed before it is TO-delivered, and given that the system is quiescent, eventually, T_c does not vary over time. Also, by the definition of Σ and by the system's quiescence, $T_c \notin \Sigma$.

Let us now consider whichever permutation σ of the transactions belonging to Σ. The proof is carried out by induction on the number n of transactions preceding T_i in σ.
(n=0) In this case no transaction belonging to Σ precedes T_i according to a sequential execution, and T_i sees the snapshot produced up to the commit of transaction T_c. Hence, we have to show that, eventually, a speculative transaction T_i^j completes its execution reading the snapshot produced by T_c and by the transactions which committed before T_c. By the structure of the SRT protocol, such a snapshot corresponds to the snapshot upon the completion of T_c. Recall that, always by the structure of the STR protocol, at least one transaction T_i^j eventually exists since T_i has been Opt-delivered (given that it belongs to Σ). There are two cases:

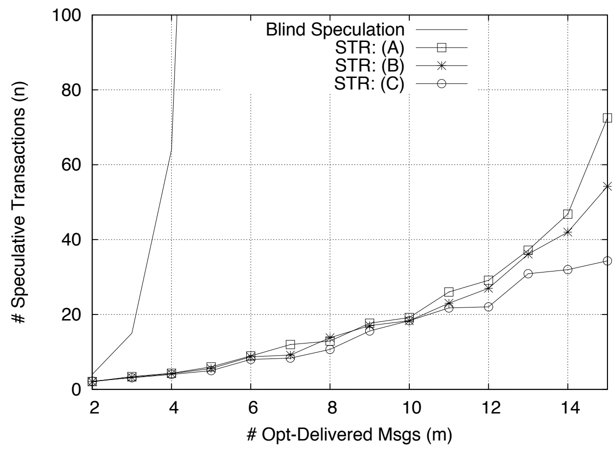
1. The transaction T_i^j that is activated upon the Opt-deliver of T_i performs its first read after the commit of T_c and of every transaction T^* preceding T_c in the final TO-deliver order. We recall that at this time there exists for each data item a version created by T_c or by any transaction that committed before T_c. Given that the history in which T_i^j reads

exclusively the committed versions of the data items is clearly serializable, upon the first read operation by T_i^j on whichever data item X, by Lemma 1 the `isVisible` method makes the committed version of X visible. Assume, without loss of generality, that T_i^j exactly returns the committed version of X, while other uncommitted versions of X, if any, are returned by sibling transactions T_i^k forked upon such a read operation. Since the committed version of a data item always exists, we can apply the same reasoning upon the second, and any other subsequent read operation on whichever data item by T_i^j. Hence, T_i^j actually sees the committed snapshot while performing all its read operations. This corresponds to the snapshot observed by T_i, hence the claim follows.

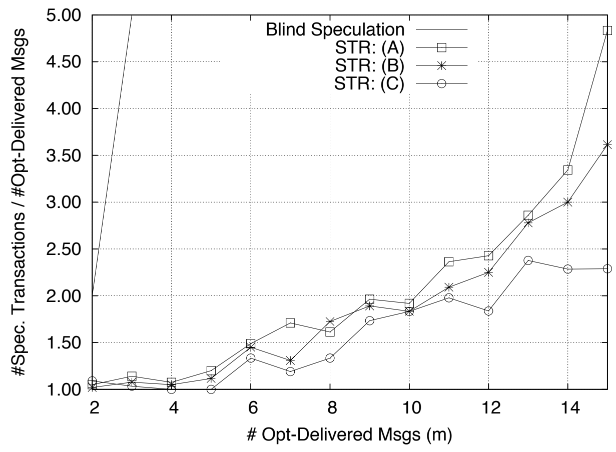
2. The transaction T_i^j that is activated upon the Opt-deliver of T_i performs its first read before T_c, or some transaction T^* preceding T_c in the final TO-deliver order has committed. In this case, if T_i^j does not read any of the data items written by T_c or by T^*, by the same reasoning of the above case, it will complete observing a committed snapshot equivalent to the one generated by executing sequentially all the transactions up to T_c according to the final TO-deliver order. If, conversely, T_i^j reads any of the data items written by T_c or by T^*, by the STR protocol structure, T_i^j will be eventually aborted upon T_c's or T^*'s commit, and a new speculative transaction instance T_i^k will be restarted. Hence, eventually a transaction T_i^k will be activated after the commit of all the transactions that precede T_i in the final TO-deliver order, thus leading us to fall in case 1 above. Hence, in any of the discussed cases, the claim follows.

(n=m) Let us consider the subsequence of m transactions that precede T_i in σ. Let us denote with $T_1,\ T_2,\ \ldots,\ T_m$ such a subsequence of transactions. By the inductive assumption, it follows that there is a time t after which, $\forall j \in [1 \ldots m]$, a completed transaction $T_j^{j_k}$ exists that has executed observing the snapshot generated by the execution of the sequential transaction history $T_c,\ T_1,\ T_2,\ \ldots,\ T_{j-1}$. We now prove that for T_i, being it the $(m+1)^{th}$ transaction in σ, there eventually exists a transaction T_i^p that completes its execution observing the snapshot generated by the sequential execution of the transaction history $\mathcal{H} = \{T_c, T_1^{j_1}, T_2^{j_2}, \ldots, T_m^{j_m}\}$.

Let t be the earliest time in which a speculative transaction T_i^p performs its first read operation and assume, with no loss of generality, that the read operation is on data item X. If at time t, all the transactions belonging to the history $\mathcal{H}$ are already completed, by Lemma 1 T_i^p will be able to see the version of X created by the last transaction in $\mathcal{H}$ that wrote X. On the other hand, if at time t not all the transactions in $\mathcal{H}$ have completed, it might happen that a transaction $T_k^{j_k} \in \mathcal{H}$ generates a new version X^k of the data item X, and makes it available with TS only after T_i^p issues the read on X. In this case, by the structure of the STR protocol, $T_k^{j_k}$ will eventually respawn a sibling transaction of T_i^p that will read X^k. In both cases, there exists a transaction T_i^q that eventually reads the data item version created by the most recent transaction in $\mathcal{H}$ that wrote X. By re-applying the same reasoning to the subsequent reads of T_i^q, we have that eventually there is a transaction T_i^w that completes by observing the snapshot produced by the execution of the sequential history $\mathcal{H}$. Hence the claim follows.



(a)



(b)

Figure 6.7: #Speculative Transactions vs #Opt-delivered Msgs.

6.3.6 Dealing with Read-Only Transactions

As in other OAB-based transactional replication solutions, when employing the STR protocol, read only transactions can be processed locally by each replica.

To this end, a simple approach would consist in running read-only transactions in a purely optimistic fashion, letting them always read the most recently commit data items' versions and relying either on an a-posteriori (at commit time), rather than eager (at each read) validation. This solution requires maintaining at any time only a single committed version of the data items, albeit at the cost of incurring in aborts of read-only transactions.

An alternative solution, exhibiting specular benefits and drawbacks, would consist in ensuring that a read-only transaction is always able to observe the snapshot created by the transactions already committed when it started. This would shelter read-only transactions from the chance of aborting, though forcing TS to maintain multiple committed versions of each data item.

6.4 Simulation Results

The optimality of an STR protocol, as expressed in Section 6.2, ensures the ability of the protocol to explore the minimum set of distinct serialization orders that suffices to guarantee the "coverage" of any arbitrary final delivery order established by the OAB service. Clearly, given a number m of optimistically delivered transactions, the optimal number n of speculative transactions that have to be spawned to ensure complete coverage is a function of the actual conflict patterns among transactions. Also, the value of n has a direct impact on the computational resources that need to be employed both to carry out the speculative execution and to handle any house-keeping data structure, whose size may vary as a function of n. By this consideration, if for realistic transaction data access patterns, the growth of n (with respect to m) happens to be moderate, we can argue that an implementation of a protocol providing optimality is likely prone to incur in a relatively bounded overhead (e.g. due to bounded size of house-keeping data structures), while exhibiting the potential for real performance improvements (e.g. via latency reductions in delivering the results of transactional data manipulations to the application layer).

The simulation study presented in this section addresses effectiveness issues exactly according to the above perspective, hence in a manner independent of any specific technology or design style for the internals characterizing the transactional system. In particular, our target is to gain insights on the practical relevance of our proposal by experimentally determining, the actual rate of growth of the optimal number of speculative transactions as a function of the number of optimistically delivered transactions, when considering realistic settings for both the size of the data set and the transaction logic (e.g. in terms of data access pattern). As a final preliminary note, the simulation study is framed within the context of Software Transactional Memory systems.

To this end we have developed a simulator which can be fed with:

(i) a number m of optimistically delivered transactions, treated as an independent parameter in our study;

(ii) traces of those m transactions accessing a specific data set.

Once started up, the simulator runs our STR protocol by activating the whole (and optimal) set of n speculative transactions necessary to ensure the complete coverage of the (conflict dependent) speculative serialization orders related to the different delivery orders associated with the m optimistically delivered messages.
Data access patterns of the transactions running within the simulation is determined by using traces of the popular RedBlack Tree benchmark for Transactional Memories introduced in [42].

Since read-only transactions can be handled within the replicated system via solutions aside of speculation schemes proper of our protocol (see Section 6.3.6), in our simulation experiments we consider a mix containing exclusively update transactions performing, with equal probability, either an insertion or a delete of an entry of the RedBlack Tree. Given that the aim of the simulation study is to determine the rate of growth of n vs m by observing the protocol evolution in a phase which mimics quiescence, the timing of actions from the traces becomes not relevant. Hence, each read/write operation is modeled as having a fixed fictitious cost of one simulation time unit.

As for the data set, the RedBlack Tree benchmark is characterized by two parameters, namely the initial size, *init*, and the maximum size, *max*, of the tree, variations of which allow to vary the probability of conflict among transactions (the larger is the size of the tree, the least is the conflict probability among transactions accessing the tree). To evaluate the STR protocol in scenarios representative of different conflict probabilities, we consider three different settings for these two parameters:

(A) *init*=32, *max*=1024;

(B) *init*=128, *max*=2048;

(C) *init*=1280, *max*=20480.

The plots in Figure 6.7.a contrast the number, n, of speculative transactions activated by our STR protocol, with the number of transactions that would have been activated by a naive speculative approach that does not take into

account the actual conflict relations developed at run-time by the transactions. As recalled in Section 6.2, such a number of speculative transactions is given by the expression $\sum_{i=1...m} \frac{m!}{(m-i)!}$.
In the plots, we report the value of n while varying the number m of optimistically delivered messages between 2 and 15. The plots clearly highlight the effectiveness of the optimal STR protocol in reducing the number of serialization orders that need to be speculatively explored in every considered scenario.
When m is equal to 10, for instance, n is equal to around 10 millions for the blind speculative approach, whereas it varies between 15.6 and 17.7 for the optimal STR protocol. Figure 6.7.b allows us to visualize these data from a different perspective. By reporting the ratio between n and m, we can quantify the amount of speculative transactions that the system should process in order to ensure completeness, with respect to a non-speculative system which exactly processes a single transaction for every Opt-delivered message (e.g. by processing it only after the corresponding TO-deliver).
The plots show that, as long as m is less than 5, such a ratio is flat around the value 1 for the STR protocol. On the other hand, if m grows up to 10, with the STR protocol it suffices, on average, to execute each transaction in no more than two different serialization orders. Finally, with $m = 15$, depending on the considered contention scenario, it would be necessary to explore, on average, between 2.5 and 5 different serialization orders per transaction.

As we have pointed out, non-speculatively replicated transactional systems wait for the completion of the distributed coordination scheme before transaction processing activities are carried out. Hence, beyond impacting the latency of transaction completion (and related result delivery) they may also suffer from CPU under-utilization (as shown in Section 4.2), especially in fine grain transactional contexts. The latter phenomenon could even become more evident according to the trend of having machines based on the many-core architectural paradigm.

By the above simulation results, the proposed speculative replication approach can lead to exploit such idle computational resources in a fruitful manner since the reduced amount of speculatively explored serialization orders (compared to any blind scheme not accounting for actual conflicts among transactions) would allow avoiding thrashing. In other words, it is our belief that the above data strongly support the practical relevance of our optimal STR framework/protocol, thus establishing it as a reference for the design and development of replicated OAB-based transactional systems relying on the systematic exploitation of transaction speculation.

Chapter 7

Changing the Perspective: Speculating According to an Opportunistic Paradigm

Clearly, serializing the optimistically delivered transactions according to the optimistic message delivery order does not pay-off in case of non-minimal likelihood of mismatch between optimistic and final message ordering. In such a case, in fact, optimistically processed transactions may have to be aborted and restarted right after OAB completion, thus nullifying any performance gain associated with their early activation. Unfortunately, ordering mismatch is more likely to occur at high load, which leads the aforementioned approaches to lose their effectiveness precisely when the increased workload would have instead demanded an increased efficiency in order to ensure adequate performance. Experimental data provided in [51] show in fact that, even in case of latency-predicable networks like LANs, the spontaneous order property typically holds only for normal load periods, but it is quite unlikely to hold when the load tends to become heavy.

In order to cope with this issue, the approach described in Chapter 6 proposes the idea to speculatively explore the *entire* set of serialization orders in which optimistically delivered transactions would observe different snapshots (i.e. states) of the underlying transactional system. Such a complete-exploration approach ensures the ability to eventually guess *any* actual order established by the OAB service, provided the availability of sufficient time and computational resources.

In this chapter we change our view point and presented a novel active replication protocol for transactional systems based on an *opportunistic* paradigm, which we name OSARE, namely Opportunistic Speculation in Active REplication. Similarly to the one presented in Chapter 5, OSARE maximizes the overlap between replica coordination and transaction execution phases by propagating, in a speculative fashion, the (uncommitted) post-images of completely processed, but not yet finally delivered, transactions across chains of conflicting transactions. However, the rules according to which the speculated serialization orders are selected in OSARE are completely different from any existing literature solution, and also from the approach presented in Chapter 6. To maximize concurrency, OSARE activates the processing of a transaction, say T, as soon as it is optimistically delivered and attempts to serialize it after any previously optimistically delivered transaction, say T'. Clearly, this attempt may fail, given that, at the time in which the processing of T is activated, T' may still be running. Therefore, the read operations issued by T may miss the values generated by T', an event that we refer to as "snapshot-miss" in the following. In such a case, OSARE does not abort and restart T to ensure that its serialization order is aligned with the optimistic message delivery order.

Conversely, OSARE takes advantage of the occurrence of snapshot-miss events in an *opportunistic* fashion, by exploring additional serialization orders not only for T, but also for any transaction originally serialized after T whose execution may be affected, possibly transitively, by the snapshot-miss involving T. Also, OSARE biases the speculative exploration towards the serialization order aligned with the optimistic message delivery order, by triggering the (re-)activation of a new instance of T (and, recursively, of the transactions having developed a read-from dependency from T) as soon as it detects T's snapshot-miss.

The OSARE respawning logic guarantees that freshly activated transactions are correctly serialized after T', thus avoiding redundant executions of multiple instances of the same transaction observing identical snapshots. In addition, OSARE is designed to guarantee that, for each transactional request delivered by the OAB service, there eventually exists one speculative transaction instance whose view of the serialization order is aligned with the optimistic delivery order.

It is worthy to highlight that the likelihood of snapshot-miss events is higher in high concurrency scenarios, namely when the inter-arrival time of optimistic deliveries is relatively short compared to transaction processing latency. Interestingly, these scenarios are precisely those in which the probability of mismatches between the optimistic and final message delivery orders, and consequently the added value of exploring additional speculative serialization orders, is higher. The ability of OSARE to adjust adaptively its degree of speculation on the basis of the current level of concurrency in the system represents a unique, innovative feature, which, to the best of our knowledge, does not appear in any literature result in the field of actively replicated transactional systems. Also, such an ability points out a tradeoff between OSARE and the STR protocol provided in Chapter 6, where OSARE is designed according to a perspective that surely looks more pragmatical.

We assess the performance of OSARE via an extensive simulation study. Our simulation model relies on traces collected by running well known benchmarks for Software Transactional Memory systems (STM) [15, 42], and the APPIA [55] group communication toolkit. Our experimental study highlights the effectiveness of the opportunistic speculative approach pursued by OSARE, showing that, when the probability of mismatch between optimistic and final delivery in non-minimal, it can provide up to 160% response-time speedup compared to the protocol described in Chapter 5, which entails speculation limitedly to the serialization order associated with the optimistic delivery sequence.

7.1 System Model

The OSARE system model is compliant with one presented in Chapter 3. However, in order to ease the protocol presentation and simplify its understanding, we introduced some additional features.

Firstly, we add to the functionalities of the Speculative Concurrency Control (SCC), the function Complete(), used to explicitly inform the Speculative Transaction Manager (SXM) about the completion of the execution of a transaction.

Further, the Transactional Store (TS) has been enriched with two primitives to manipulate data item versions that are:

- `setComplete(`X^T`,`T`)`, which marks a data item version X^T written by transaction T as *complete*;
- `unsetComplete(`X^T`,`T`)`, which is used for removing a complete data item version X^T originally exposed by transaction T.

7.2 The OSARE Protocol

In this section is provided the description of the OSARE protocol, by first introducing some key notations and data structures and then discussing the protocol pseudo-code.

7.2.1 Protocol Notations and Data Structures

The OAB service delivers transactions, each of which is denoted as T_i in our protocol. OAB delivered transactions are however never directly executed by SCC, which, conversely only executes speculative transaction instances, denoted using the notation T_i^j.

Each transaction T_i^j speculatively executed by OSARE keeps locally track of its own *serialization view*, defined as the totally ordered sequence of transactions that are expected to be serialized before T_i^j. The construction of the per-transaction view of the serialization order relies on two main data structures:

– OptDelivered: a global list of speculative transaction identifiers, accessible by all the transactional threads;

– T_i^j.SpeculativeOrder: a local list of speculative transaction identifiers which is associated with the transactional thread currently handling transaction T_i^j.

The sequence of speculative transactions recorded within T_i^j.SpeculativeOrder expresses, on the basis of the view by T_i^j, the order according to which speculative transactions preceding T_i^j should be serialized. This determines a history of speculative transactions whose snapshots may be visible by T_i^j's read operations.

We use the notation $T_k^h \overset{T_i^j}{\to} T_s^t$ to indicate that T_k^h precedes T_s^t within the ordered list T_i^j.SpeculativeOrder. This expresses that, according to the view of T_i^j:

1. T_k^h and T_s^t belong to the same speculative history of transactions;
2. T_k^h and T_s^t are both expected to be serialized before T_i^j;
3. T_k^h is expected to be serialized before T_s^t.

By convention, the special transaction identifier T_α^ω represents the minimum element for the $\overset{T_i^j}{\to}$ relation for whichever transaction T_i^j. This notation is used to encapsulate the history of already committed transactions that, according to T_i^j's view of speculative serialization expressed via the relation $\overset{T_i^j}{\to}$, must necessarily be serialized before T_i^j and before any transaction belonging to T_i^j.SpeculativeOrder.

Always by convention, T_i^j represents the maximum element of the $\overset{T_i^j}{\to}$ relation. Overall, denoting with $(T_{k_1}^{h_1}, \dots, T_{k_n}^{h_n})$ the sequence of transactions recorded within T_i^j.SpeculativeOrder, we have: $T_\alpha^\omega \overset{T_i^j}{\to} T_{k_1}^{h_1} \overset{T_i^j}{\to} \dots \overset{T_i^j}{\to} T_{k_n}^{h_n} \overset{T_i^j}{\to} T_i^j$.

The global list OptDelivered maintains the transaction identifiers whose speculative serialization view is aligned with the order of optimistic deliveries generated by the local OAB service. An addition to this list occurs upon the Opt-delivery of a transaction T_i by appending, to the tail of the list, the identifier of the first instance of speculative transaction associated with T_i, which by convention we denote as T_i^0. Before such an update occurs, T_i^0.SpeculativeOrder is populated with the current content of OptDelivered. In this way, T_i^0 is provided with a view of the speculative serialization order which is (at least upon its activation) aligned to the one associated with the current sequence of optimistically delivered transactions.

The global list OptDelivered is also updated when a commit occurs for an already TO-delivered transaction. In this case one element is eliminated from the list, reflecting the fact that the corresponding speculatively executed transaction has been finalized. If the element is not the top standing one, it means that some TO-delivery has subverted the Opt-delivery order.

A third type of update of OptDelivered occurs when a transaction T_i^j currently recorded within this list is discovered to have been actually executed along a speculative serialization order that diverges from the one currently expressed by the list content. As we will see, in this case, a new instance of speculative transaction T_i^k is spawned, substituting T_i^j within that list (possibly recursively causing other spawns and substitutions). This is done in order to guarantee the existence of a speculative transaction instance that is being processed in a serialization order compliant with the order expressed by the currently optimistically delivered transactions.

7.2.2 Protocol Logic

The protocol pseudo-code is shown in Figures 7.1 and 7.2, and is discussed in the following.

Optimistic delivery of transactions.

Upon the Opt-deliver event of a transaction T_i, the SXM instantiates a speculative transaction T_i^0, and then sets up the serialization order to be seen by this new transaction instance by copying the current content of OptDelivered into T_i^0.SpeculativeOrder. Next, the SXM appends T_i^0's identifier within the global list OptDelivered to reflect that at least one instance of speculative transaction associated with T_i exists, and that it should be serialized at the tail of the sequence of speculative transactions currently recorded within the OptDelivered list. Finally, the SXM passes the control to SCC for processing T_i^0 transaction activities by invoking the ActivateSpeculativeTransaction function with T_i^0 as input parameter. The latter function also adds T_i^0 to the set of active transactions ActiveXacts.

Handling of read and write operations.

```
OrderedList<Transaction> TODelivered, OptDelivered;
Set<Transaction> ActiveXacts;

upon Opt-deliver(Transaction $T_i$) do
  $T_i^0$ =$T_i$.createNewSpecXact();
  $T_i^0$.SpeculativeOrder = copy(OptDelivered);
  OptDelivered.enqueue($T_i^0$);
  ActivateSpeculativeTransaction($T_i^0$);

void ActivateSpeculativeTransaction(Transaction $T_i^s$)
  ActiveXacts.add($T_i^s$);
  start processing thread;

DataItemValue Read(Transaction $T_i^s$, DataItem $X$)
  if ($X \in T_i^s$.WriteSet) return $T_i^s$.WriteSet.get(X).value;
  select version of $X$ completed or committed by $T_j^t = max\{T_j^f \overset{T_i^s}{\rightarrow} T_i^s\}$;
  // the committed version is written by $T_\alpha^\omega$ by definition
  $T_i^s$.ReadSet.add($X$);
  $T_i^s$.ReadFrom.add($T_j^t$);
  return $T_i^s$.ReadSet.get($X$).value

void Write(Transaction $T_i^s$, DataItem $X$, Value $v$)
  if ($X \in T_i^s$.WriteSet) $T_i^s$.WriteSet.update($X, v$) ;
  else $T_i^s$.WriteSet.add($X, v$);

void Complete(Transaction $T_i^s$)
  atomically do
    $T_i^s$.isCompleted = TRUE
    $\forall X \in T_i^s$.WriteSet do setComplete($X, T_i^s$);
    $\forall$ $T_j^t$ s.t. ($\exists X \in T_j^t$.ReadSet: ($X \in T_i^s$.WriteSet  and
        $T_i^s = \max\{T_l^f : T_l^f \overset{T_j^t}{\rightarrow} T_j^t$ exposing a complete version of $X\}$) do
      $T_j^{xId}$ =$T_j$.createNewSpecXact();
      $T_j^{xId}$.SpeculativeOrder = copy($T_j^t$.SpeculativeOrder);
      $T_j^t$.SpeculativeOrder.remove($T_i^s$); / reflects the snapshot-miss of $T_j^t$
      if ($T_j^t \in OptDelivered$) OptDelivered.replace($T_j^t, T_j^{xId}$);
      wave($T_j^t$,$T_j^{xId}$,$T_i^s$);
  wait until TODelivered.topStanding == $T_i$;
  if ($\forall X \in T_i^s$.ReadSet: $X$.version == LatestCommitted) $T_i^s$.RaiseEvent(Commit);
  else $T_i^s$.RaiseEvent(Abort);

void wave(Transaction $T_j^t$, Transaction $T_j^{xId}$, Transaction $T_i^s$)
  $\forall T_l^f$ s.t. $T_j^t \in T_l^f$.ReadFrom do
    $T_l^{xId'}$ =$T_l$.createNewSpecXact();
    $T_l^{xId'}$.SpeculativeOrder = copy($T_l^f$.SpeculativeOrder);
    $T_l^{xId'}$.SpeculativeOrder.replace($T_j^t$,$T_j^{xId}$);
    $T_l^f$.SpeculativeOrder.remove($T_i^s$);
    if ($T_l^f \in OptDelivered$) OptDelivered.replace($T_l^f, T_l^{xId'}$);
    wave($T_l^f$,$T_l^{xId'}$,$T_i^s$)
  $\forall T_l^g$ s.t. ($T_j^t \in T_l^g$.SpeculativeOrder and $T_j^t \notin T_l^g$.ReadFrom) do
    $T_l^g$.SpeculativeOrder.replace($T_j^t$,$T_j^{xId}$);
  ActivateSpeculativeTransaction($T_j^{xId}$);
```

Figure 7.1: Behavior of SCC (Part A).

```
upon TO-Deliver(Transaction $T_i$) do
  TODelivered.enqueue($T_i$);

upon Abort(Transaction $T_i^s$) do atomically
  $\forall X \in T_i^s$.WriteSet do unsetComplete($X, T_i^s$);
  $\forall T_j^h \in$ActiveXacts $s.t.$ $j \neq i$ and $T_i^s \in T_j^h$.SpeculativeOrder do
    $T_j^h$.SpeculativeOrder.remove($T_i^s$);
  $\forall T_j^h$ s.t. $T_i^s \in T_j^h$.ReadFrom do $T_j^h$.RaiseEvent(Abort);
  ActiveXacts.remove($T_i^s$);

upon Commit(Transaction $T_i^k$) do atomically
  ActiveXacts.Remove($T_i^k$);
  $\forall X \in T_i^k$.WriteSet do $T_i^k$.WriteSet.Commit($X$);
  TODelivered.Dequeue($T_i$);
  OptDelivered.Remove($T_i^*$);
  $\forall T_i^h \in$ActiveXacts $s.t.$ $h \neq k$ do $T_i^h$.RaiseEvent(Abort);
  $\forall T_j^h \in$ActiveXacts $s.t.$ $j \neq i$ and $T_i^k \in T_j^h$.SpeculativeOrder do
    $T_j^h$.SpeculativeOrder.remove($T_i^k$);
  $\forall T_j^h \in$ActiveXacts $s.t.$ $T_i^k \in T_j^h$.ReadFrom do $T_j^h$.RaiseEvent(Validate);

upon Validate(Transaction $T_i^k$) do
  $\forall X \in T_i^k$.ReadSet do
    compute $T_j^h = max\{T_l^f : T_l^f \overset{T_i^k}{\rightarrow} T_i^k$ and $X \in T_l^f$.WriteSet$\}$;
    if ($T_i^k$.ReadSet.get($X$).Creator $\neq T_j^h$)
      $T_i^k$.RaiseEvent(AbortRetry);
      break;

upon AbortRetry(Transaction $T_i^s$) do atomically
  $\forall X \in T_i^s$.WriteSet do unsetComplete($X, T_i^s$);
  $\forall T_j^h$ s.t. $T_i^s \in T_j^h$.ReadFrom do $T_j^h$.RaiseEvent(AbortRetry);
  restart transaction $T_i^s$;
```

Figure 7.2: Behavior of SCC (Part B).

When a transaction T_i^s issues a read operation on a data item X, the SCC verifies whether a version of X belongs to the write set of the reading transaction. In the positive case, the written value is returned (ensuring that the read operation returns the version of X associated with the last write issued by T_i^s during its execution). On the other hand, if T_i^s has not previously issued a write on X, the precedence relation associated with T_i^s.SpeculativeOrder is used in order to determine which version of X that should be made visible to T_i^s. To this end it is used a simple rule that allows identifying the most recent version exposed by a completed or committed transaction according to the serialization view of T_i^s.
Specifically, it is determined the maximum speculative transaction T_j^t preceding T_i^s according to the $\overset{T_i^s}{\rightarrow}$ relation, which has:

(i) written X, and

(ii) already completed its execution.

The logic associated with a write operation of a transaction T_i^s on a data item X is also very simple: the SCC simply stores the updated value of X into the write-set of the writing transaction. As we will see, the data item versions generated by a transaction T_i^s are in fact made all atomically visible to the remaining speculative transactions only once that T_i^s reaches its completion phase.

Completion of speculative transactions.

The core of the OSARE protocol is represented by the logic handling the completion of a speculative transaction. Specifically, when the Complete method is executed by the SCC for transaction T_i^s, each data item version created (i.e. written) by T_i^s is made speculatively visible by setting its state to the complete value. Before making the snapshot produced by T_i^s visible, however, it is first checked whether every transaction T_j^t that, according to its serialization view, is serialized after T_i^s, is still correctly executing along that order, or whether it has instead missed the snapshot generated by the execution of T_i^s.
More in detail, a snapshot miss event is detected in case:

(i) T_i^s wrote some data item X for which T_j^t has already issued a read operation;

(ii) T_i^s is the last speculative transaction to have written X among those in T_j^t's speculative view.

In this case, in fact, T_j^t has observed a different version of X, despite, according to its serialization view, it should have observed the version of X generated by T_i^s. A snapshot is detected if T_j^t should have seen the version that T_i^s wrote, but it has observed some different version of X . In this case we are in the presence of a *snapshot-miss* event, since the current serialization view of T_j^t has not been respected, given that it did not observe the data item version that it should have read from T_i^s.
The following three actions are taken to handle a snapshot-miss event.

1. A new speculative instance T_j^{xId} is activated, setting its serialization view to be equal to the one currently associated with T_j^t. Given that T_i^s has now reached completion, the new instance T_j^{xId}, during its execution, is guaranteed not to miss the snapshot produced by T_i^s.

2. The serialization order of T_j^t is then updated by removing T_i^s (namely the transaction whose write operation T_j^t missed) from T_j^t.SpeculativeOrder. Next, if T_j^t was originally recorded within OptDelivered, it is replaced by T_j^{xId} within this list. This reflects the fact that T_j^t is known not to be any longer in a serialization order compliant with that of the optimistic message delivery order, and that there is now a new incarnation of T_j, namely T_j^{xId}, which will be activated (in the wave, see the next point) in order to pursue this serialization order.

3. The snapshot-miss event is recursively propagated via the wave() method (described shortly afterwards) across chains of transactions that were transitively serialized (according to their own serialization view) after the transaction T_j^t involved in the snapshot-miss event.

After having handled all the snapshot-miss events detected upon its completion, T_i^s simply remains waiting for the corresponding transaction T_i to be TO-delivered, and to become the top standing element within the TODelivered queue. As it be clearer in the following, this means that for any transaction T_j, which was TO-delivered before T_i, there exists a corresponding speculatively executed transaction T_j^* that has been already committed.
Hence T_i^s can now be safely validated (by verifying whether it has read data items belonging to the latest committed snapshot) and, depending on the validation's outcome, a commit, or an abort, event is raised to finalize the commit, or the abort, of the speculative transaction.

Recursive propagation of snapshot-miss events.

As we have just explained, the completion of a transaction T_i^s can trigger a series of snapshot-miss events involving transactions T_j^t, which, according to their serialization view, should have observed the value of a data item updated by T_i^s when executing a read operation and that have instead observed a version of that data item created by a different transaction. In this case T_j^t's speculative view is updated (removing T_i^s from it) to reflect the occurrence of the snapshot-miss event, and a new speculative instance of transaction T_j, namely T_j^{xId} is activated that will be guaranteed not to miss the snapshot created by T_i^s. In order to pursue, on one hand, the opportunistic exploration of additional serialization orders, and, on the other hand, the completion of a sequence of transactions serialized in an order compliant with the optimistic message delivery order, OSARE transitively propagates the handling of the snapshot-miss event via the wave method. The transaction T_j^t, in fact, may have already completed its execution and exposed its snapshot to a different speculative transaction, say T_l^f. In this case, even if T_l^f had not missed the snapshot generated by T_i^s, it is still transitively involved by the snapshot-miss event affecting T_j^t. Analogously to T_j^t, therefore, T_i^s needs to be removed by the speculative view of T_l^f. Further, in order to pursue the exploration of a serialization order compliant with the optimistic message delivery order, a new speculative instance of T_l, namely $T_l^{xId'}$, needs to be activated, which should now include in its serialization view T_j^{xId}.

Finally, just like in the Complete method, it is verified if T_l^f was considered to be serialized in an order compliant with the optimistic delivery order (by checking whether it is included in OptDelivered). In the positive case, the OptDelivered sequence needs to be updated, replacing T_l^f with $T_l^{xId'}$, reflecting the fact that it is now the latter one to be expected to be serialized according to the optimistic delivery order. Note that the wave method relies on an elegant recursion technique to ensure the complete propagation of the snapshot-miss across the whole set of transactions that have established a transitive read-from relationship from T_j^t. Upon returning from the recursive call, the SCC substitutes T_j^t with T_j^{xId} from the speculative view of every transaction T_l^g that:

(i) contained T_j^t in its speculative view, and

(ii) did not develop a read-from dependency from T_j^t.

This is necessary in case T_l^g is still active, in order to ensure that during its subsequent reads, it will be able to observe the snapshot generated by T_j^{xId}, thus correctly realigning T_l^g's speculative view towards the serialization order compliant with the optimistic message delivery oder.
Finally, activation of processing activities for the spawned transaction T_j^{xId} takes place right before returning from wave().

Final delivery of transactions.

As for the handling of final delivery events, the associated logic only entails the enqueuing of the delivered transaction within TODelivered. This ensures that the corresponding placeholder is sequentialized after all the already TO-delivered ones and regulates that all the replicas validate (and ultimately commit) transactions in the same total order.

Abort and Commit Events.

The handling of the abort event simply removes the aborting transaction from the set ActiveXact and from any speculative order currently recording the transaction identifier. It also propagates the abort event towards all the transactions having read-from dependency from the currently aborting transaction.

A bit more sophisticated is the handling of the commit event. In this case, the committing transaction identifier T_i^k is removed from ActiveXacts, and every data item it wrote is marked as committed. Then, the corresponding transaction T_i is dequeued from the TODelivered list. This can cause another TO-delivered transaction to become the top standing transaction of this list, eventually enabling the commit of one of its corresponding speculative instances.
Next, whichever transaction instance T_i^* currently present within OptDelivered is removed from this list, in order to ensure that instances of T_i are no longer to be considered as belonging to the speculative portion of the serialization order associated with the sequence of optimistically delivered transactions. Further, the abort event is raised for all the transactions different from T_i^s that are instances of T_i. This leads to the abort of all the speculative transactions that had developed a, possibly transitive, read-from dependency from a a instance of T_i different from T_i^s. T_i^s is then removed from any serialization view that is currently recording it (again because it is logically passed to the committed transaction history).

Finally, it is necessary to verify whether transactions having (direct or transitive) read-from dependencies from the committing transaction are still valid. This is required since, as hinted, T_i^s is moved to the committed history. Therefore we need to verify whether the transactions exhibiting dependencies on the snapshot produced by T_i^s are still executing along a consistent speculative serialization path. This check is performed via the Validate function, which simply verifies whether the items read by those transactions still correspond to those produced by the transactions representing the maximum elements exposing these items as complete along the corresponding serialization orders.

In the negative case it means that the transaction (directly or transitively) reading from the committing transaction T_i^k needs to be restarted by speculating along the modified path where T_i^s has been moved to the committed history (see the AbortRetry module).

7.3 Speculative trajectories explored by OSARE

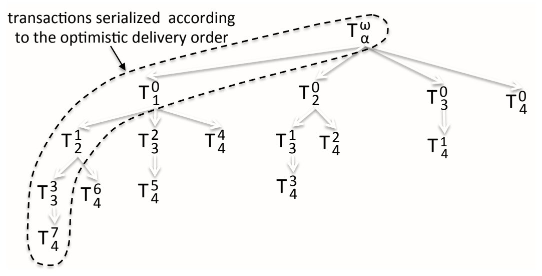


Figure 7.3: Largest Set of Speculative Serialization Orders Explored by OSARE.

Rather than attempting a complete speculative exploration, OSARE biases speculation exploration towards the optimistic delivery order. More precisely, OSARE explores exclusively serialization orders obtained by extracting subsequences of variable length of the optimistic delivery order, see Figure 7.3. This choice allows, on one hand, to favor the opportunistic exploration of serialization orders ”close” to the optimistic delivery order. On the other hand, it allows to rely on a simple and lightweight logic (when compared to the protocol presented in Chapter 6, which is oriented to speculation completeness) to identify the set of speculative serialization orders to be explored.

Given that OSARE activates additional speculative transactions in an opportunistic fashion (i.e. only upon the occurrence of snapshot-miss events), the actual number of speculative transactions spawned by OSARE depends on the level of concurrency and conflict among the set of optimistically delivered transactions.

Nevertheless, it is still possible to determine analytycally an upper bound on the number of speculative transactions activated in OSARE as a function $\theta(n)$ of the length n of the sequence σ of optimistically, but not yet finally, delivered transactions. Figure 7.3 illustrates the scenario in which OSARE explores the maximum number of alternative serialization orders for a sequence σ of length 4. The number of nodes of such a (partial) permutation tree can be, in general, enumerated using the following expression:

$$\theta(n) = \sum_{i=1\ldots n} \delta(i,n)$$

where $\delta(i,n)$ denotes the number of distinct subsequences of σ obtainable after discarding from σ the first $i-1$ messages, and is computable recursively as:

$$\delta(i,n) = \begin{cases} 1 + \sum_{j=i+1,n} \delta(j,n) & i \neq n \\ 1 & i = n \end{cases}$$

By using standard unfolding techniques, it is possible to show that $\theta(n) = 2^n - 1$.

7.4 Protocol Properties

7.4.1 Opacity

As hinted in Section 5.3, the opacity property guarantees that (O.1) committed transactions should appear as if they were executed sequentially, in an order that agrees with their real-time ordering, (O.2) no transaction should ever observe the modifications to shared state done by aborted or live transactions, and (O.3) all transactions, including aborted and live ones, should always observe a consistent state of the system.

In each replica, OSARE ensures property (O.1) by committing transactions only after a validation phase that would detect any unserializable behavior. It ensures (O.2) because read operations can only return either a committed value, or the value generated by a transaction whose execution has already reached the complete phase (and hence is neither live nor aborted at the time of the read). It ensures (O.3) since the read of a transaction T_i^s always returns the value generated by the latest complete transaction that precedes T_i^s according to its own view of the speculative serialization order. This mechanism clearly excludes the possibility of incurring in any anomaly in case all the reads executed by T_i^s take place i) after that all the transactions preceding it according to its speculative order have already completed, and ii) if the speculative order of T_i^s is never altered.
Let us start by analyzing the scenario in which a transaction T_l^f, which is serialized before T_i^s according to its speculative order, completes after T_i^s has already issued at least a read operation. Assume, with no loss of generality, that this read is on a data item X and returned a value written by a transaction T_j^t such that $T_j^t \overset{T_i^s}{\rightarrow} T_i^s$. In this case, the only possible anomaly that could affect T_i^s would arise if also T_l^f had written X and if $T_j^t \overset{T_i^s}{\rightarrow} T_l^f \overset{T_i^s}{\rightarrow} T_i^s$. In this

case, if T_i^s were ever to read, along its execution, the value of any data item written by T_l^f, it would observe an inconsistent state. In fact, having read X from T_j^t, T_i^s has already serialized itself before T_l^f. This would therefore lead to a violation of property (O.3). On the other hand, this scenario is avoided in OSARE since, as soon as T_l^f completes (namely, before making its data items speculatively visible), it detects that T_i^s has missed its write on X and removes itself from the speculative order of T_i^s, preventing T_i^s from ever reading values written from T_l^f.
To complete the reasoning on the absence of non-opaque schedules it remains to assess the scenarios in which the speculative order of a transaction is altered during its execution. This can happen in three situations. The first one, whose correctness has just been discussed, is associated with a snapshot-miss event. The remaining two cases are associated, respectively, with the Commit and Abort events of a transaction T_i^k. Both events, in fact, trigger the removal of T_i^k from the speculative order of every transaction T_j^h that is supposed to serialize itself after T_i^k. In the case of a Commit event, T_j^h is immediately aborted (and restarted) if it is detected that the previously executed read operations would have observed different values if re-executed according to the updated speculative order. On the other hand, in the case of an Abort event, T_j^h is immediately aborted if it developed any (direct or transitive) read-from dependency from T_i^s. In both cases, therefore, it is guaranteed that T_j^h, were it still be running at the time in which its speculative order is altered, will not observe inconsistent snapshots when continuing its execution.

7.4.2 1-Copy Serializability

1-Copy Serializability [8] is ensured since transactions are committed at every site only upon a deterministic validation that is executed by all replicas in the same total order, i.e., the final delivery order of the OAB service.

7.4.3 Non-redundant speculation

This property ensures that no two speculative instances T_j^t, T_j^{xId} of the same transaction T_j observe the same snapshot. This follows by observing that OSARE activates a new speculative transaction T_j^{xId} only if it detects that a transaction T_j^t has missed a value written by a transaction T_i^s serialized before T_j^t according to its speculative order. In this case, T_j^{xId} will observe at least a data item version different from those observed by T_j^t since T_j^{xId} will not miss the value written by T_i^s, having T_i^s already competed and made visible its snapshot. The same is true for all the transactions that are (recursively) spawned due to an (direct or transitive) read-from dependency on T_i^s. They will execute along a serialization order where the snapshot by T_j^t is not visible since it is replaced by the one provided by T_j^{xId}.

7.4.4 Lock-freedom

Lock-freedom [36] guarantees that there is always at least a thread to make progress, thus ruling out deadlock and livelock scenarios. In OSARE, this is a direct consequence of the fact that the transaction currently representing the top standing element within the TODelivered queue always experiences an abort free (re)run.

7.5 Simulation Study

7.5.1 Simulation environment

In order to assess the performance of OSARE we developed detailed simulation models for the following protocols: OSARE, AGGRO (presented in Chapter 5 and Opt [51], all relying on OAB, and traditional State Machine (SM), relying on AB. AGGRO and Opt are baseline protocols for the evaluation of OSARE since they entail some form of speculation, either limited to conflicting transaction chains of length one, or entailing multiple conflicting transactions along the chain. SM acts as a reference for assessing the performance of speculative vs non-speculative protocols.
In order to accurately model transactional execution dynamics we collected traces using a number of heterogenous STM benchmarks, namely:

(i) three micro-benchmarks, that have been adopted in a number of performance evaluation studies of STM systems [15, 24, 42]:

  - Red Black Tree;
  - List;
  - SkipList;

(ii) two benchmarks of the STAMP suite [17]:

  - Yada;
  - Labyrinth++;

The micro-benchmarks are already described in Section 4.1. Conversely, Yada++ is a parallel implementation of the Ruppert's algorithm for Delaunay mesh refinement, which uses transactions to concurrently modify a shared graph. Labyrinth, instead, implements a variant of the Lee's algorithm to lay in parallel the junctions of an electrical circuit. The above benchmarks were configured not to generate any read-only transaction. This choice depends on the fact that, in all the protocols considered in this study, read-only transactions can be executed locally, without the need for distributed coordination. By only considering update transactions, we can therefore precisely assess the impact of distributed coordination on the performance of the replicated system, as well as the performance gains achievable by OSARE.

The machine used for the tracing process is equipped with an Intel Core 2 Duo 2.53 GHz processor and 4GB of RAM, running Mac OS X 10.6.2 and JVSTM [15].

The simulation model of the replicated system comprises a set of 4 replicated STM processes, each hosted by a machine equipped with 32-cores processing transactions at the same speed as in the above architecture.

The transactions' arrival process via optimistic and final message deliveries is also trace-driven. Specifically we use traces generated by running a sequencer based (O)AB implementation available in the Appia GCS Toolkit [55] on a cluster of 4 quad-core machines (2.40GHz - 8GB RAM) connected via a Gigabit Ethernet and using TCP at the transport layer. We injected in the system messages of 512 bytes (largely sufficient to encode the parameters of the transactional methods exposed by the considered STM benchmarks) with an exponentially distributed arrival rate having mean λ. We treat λ as the independent parameter of our study, letting it vary in the range [1000,4000] messages per second, thus expressing from low/moderate up to high load to be sustained by the group communication service. As expected, the mismatch between optimistic and final delivery orders (or message reordering for the sake of brevity) increases along with the message arrival rate, ranging from 16%, at 1000 msgs/sec, up to 48%, at 4000 msgs/sec.

7.5.2 Analysis of the Results

The plots in Figure 7.4 report the speed-up achieved by OSARE vs the other protocols, evaluated as the percentage of additional latency for executing a transaction (being the latency the average time since the TO-broadcast of a transaction till its commitment) in any of these protocols with respect to OSARE.

The data highlight striking performance gains by OSARE compared to AGGRO, which increase (up to around 160%) as the load and the message reordering grow. This is due to the fact that, by opportunistically processing a transaction in multiple serialization orders, OSARE overlaps more effectively processing and communication. On the other hand, the gains over Opt (which unlike OSARE and AGGRO does not speculate along chains of conflicting transactions) and SM are even larger, being on the order of up to 350/360%.

Comparing more closely OSARE and AGGRO, which both speculate along

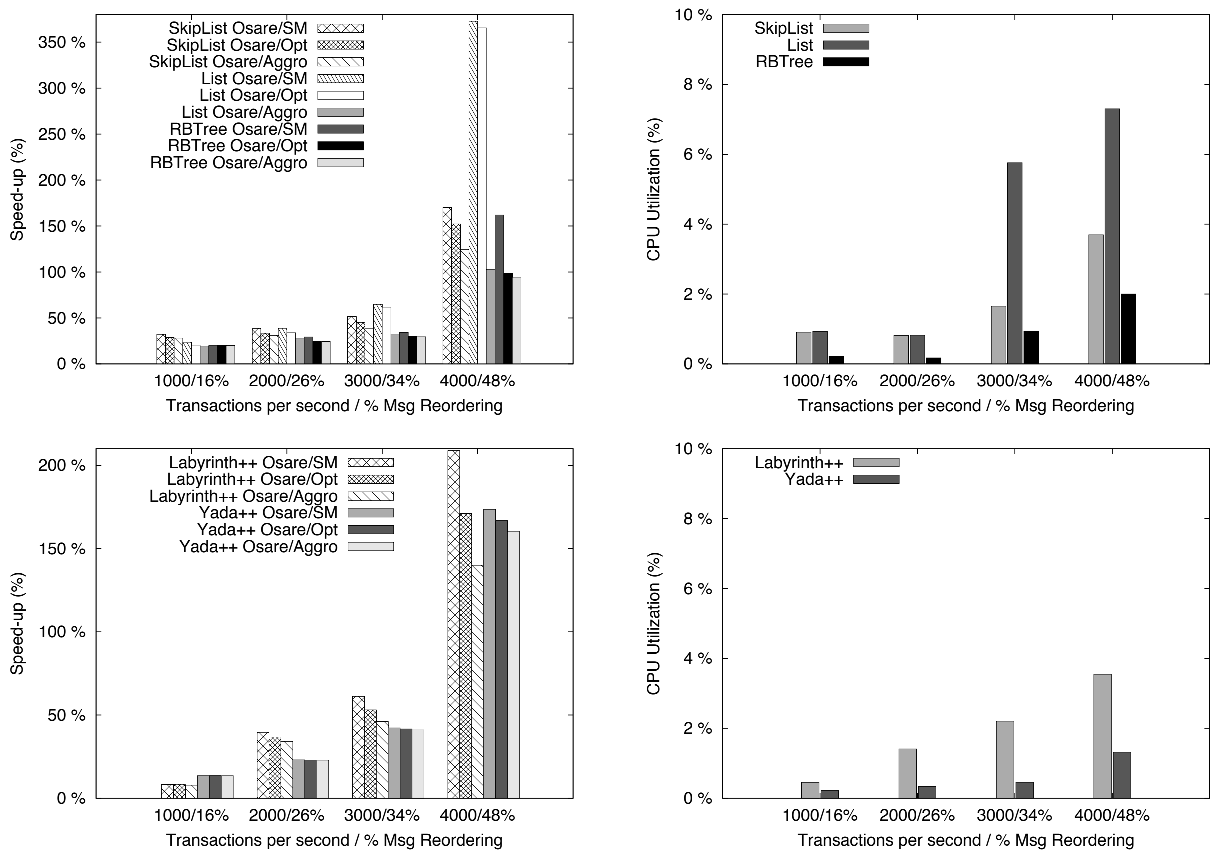


Figure 7.4: Speed-up by OSARE.

chains of conflicting transactions, the number of transactions that have already started (or completed) along a serialization order compliant with the OAB final delivery sequence is up to 50% higher in OSARE than in AGGRO. Also, at high load, the number of transactions aborted in AGGRO is around 4x larger than in OSARE. This depends on that AGGRO uses an aggressive rollback-retry mechanism which re-activate transactions as soon as they are detected not to be serialized according to the optimistic delivery order. This policy pays off at negligible levels of message reordering. On the other hand, as soon the probability of message reordering becomes non-minimal, AGGRO incurs in a significant waste of computation, which is conversely fruitfully exploitable by OSARE thanks to its opportunistic speculative approach.

Finally, interesting conclusions can be drawn by analyzing the statistics on the average and maximum number of speculative transactions generated in OSARE. At 4000 transactions per second (exhibiting about 48% of message reordering), the average number of speculative instances activated by OSARE for a given transaction (across all the evaluated benchmarks) is 2.7. In other words, beyond the serialization order associated with the final delivery order, only 1.7 additional serialization orders are explored for each transaction.

Also, our experimental data show that, on average, the corresponding CPU utilization is less than 8% with OSARE on the simulated hardware architecture. Overall, we can deduce that the speculative approach provided by OSARE is perfectly sustainable by off-the-shelf multi-core and many-core architectures, at least when considering scenarios resembling the simulated settings.

Chapter 8

Concluding Remarks

In this dissertation I have presented some innovative research results addressing the issue of active replication of transactional systems. The basic idea underlying all these results has been to maximize the overlap between replica-coordination and local transactions processing activities. This has been done in order to effectively cope with challenging scenarios where (A) the ratio between transaction granularity and coordination latency gets reduced due to differentiated (technological) trends, such as the usage of SSD technology or the reliance on the Software Transactional Memory (STM) paradigm, and (B) the level of real parallelism while processing transactions may be largely scaled up thanks to the widespread diffusion of massively parallel architectures, such as many-core machines.

All the provided approaches have been based on the exploitation of an optimized group communication primitive, such as Optimistic Atomic Broadcast (OAB) and their performance have been evaluated by selecting STM applications as the reference test-bed. In the following I summarize the key aspects of each proposed approach.

- AGGRO *(Boosting STM Replication via Aggressively Optimistic Transaction Processing)* is an active replication protocol that relies on an Optimistic Atomic Broadcast service to determine a global transaction serialization order across all replicas, and on an innovative concurrency control scheme that allows for immediately processing optimistically delivered transactions (according to the guessed serialization order) while the broadcast service is being finalized. AGGRO is designed for networks ensuring the spontaneous order.

- STR *(Speculative Transactional Replication)* is a speculative framework for the replication of transactional systems. STR exploits an Optimistic Atomic Broadcast service and maximizes the overlapping between communication and local computation by processing transactions in different speculative serialization orders, corresponding to distinct permutations of the set of optimistically delivered messages. The challenge is to avoid the exhaustive exploration of the whole set of permutations of the optimistically delivered messages, because this is both infeasible and non-productive considering that the scenario where every transaction conflicts with every other can result highly unlikely. Hence, a (possibly large) portion of the set of the permutations would actually generate identical, redundant, computations.

– OSARE *(Opportunistic Speculation in Active REplication)* is an active replication protocol for transactional systems that combines the usage of an Optimistic Atomic Broadcast service with a speculative (opportunistic) concurrency control mechanism. OSARE relies on a lock-free algorithm to bias the speculative serialization of transactions towards an order aligned with the optimistic message delivery order. Due to the lock-free nature of the concurrency control algorithm adopted by OSARE, at high concurrency levels, namely when the probability of mismatches between the optimistic and final delivery orders is higher, the chances of exploring alternative (i.e. not equivalent to the optimistic message delivery order) transaction serialization orders correspondingly increase. By adaptively adjusting its degree of speculation on the basis of the current level of concurrency in the system, OSARE achieves robust performance also in scenarios characterized by non-minimal likelihood of mismatches between the optimistic and final delivery orders.

Bibliography

[1] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Proc. of Euro-Par*, pages 496–503, London, UK, 1997. Springer-Verlag.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[3] Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 209–218, New York, NY, USA, 2000. ACM.

[4] Akamai. Akamai white paper: Internet bottleneck.

[5] Peter Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE*, pages 562–570, 1976.

[6] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2000.

[7] Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25:171–220, June 1993.

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[10] Azer Bestavros and Spyridon Braoudakis. Value-cognizant speculative concurrency control. In *Proc. of VLDB*, pages 122–133, 1995.

[11] Nadya Bliss. Addressing the multicore trend with automatic, 2007.

[12] Andrey Brito, Christof Fetzer, and Pascal Felber. Multithreading-enabled active replication for event stream processing operators. In *Proc. of SRDS*, pages 22–31, Washington, DC, USA, 2009.

[13] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transactional memory. In *Proc. of DEBS*, July 2008.

[14] Joao Cachopo. *Development of Rich Domain Models with Atomic Actions.* PhD thesis, Technical University of Lisbon, 2007.

[15] Joao Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[16] Lásaro J. Camargos, Fernando Pedone, and Rodrigo Schmidt. A primary-backup protocol for in-memory database replication. In *NCA*, pages 204–211, 2006.

[17] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc.of ISWC*, 2008.

[18] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server dbms architectures. *SIGMOD Rec.*, 20:357–366, April 1991.

[19] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Scert: Speculative certification in replicated software transactional memories. In *SYSTOR*, page 10, 2011.

[20] Bernadette Charron-bost, Rachid Guerraoui, and André Schiper. Synchronous system and perfect failure detector: solvability and efficiency issues. In *In International Conference on Dependable Systems and Networks (IEEE Computer Society*, pages 523–532. IEEE, 2000.

[21] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 14–25, New York, NY, USA, 2001. ACM.

[22] Oracle Corporation. Oracle8i advanced replication, November 1998.

[23] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luis Rodrigues. D2stm: Dependable distributed software transactional memory. Technical Report 30/2009, INESC-ID, May 2009.

[24] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luis Rodrigues. D^{2}STM: Dependable Distributed Software Transactional Memory. In *Proc. of PRDC*. IEEE Computer Society Press, 2009.

[25] Maria Couceiro, Paolo Romano, and Luis Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. *Self-Adaptive and Self-Organizing Systems, IEEE International Conference on*, 0:184–193, 2010.

[26] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[27] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34:56–78, February 1991.

[28] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real-Time Systems*, 2:195–212, 1990.

[29] Flaviu Cristian, Richard De Beijer, and Shivakant Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1:177–201, 1994.

[30] Flaviu Cristian, Shivakant Mishra, and Guillermo A. Alvarez. High-performance asynchronous atomic broadcast. *Distributed Systems Engineering*, 4(2):109–, 1997.

[31] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[32] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[33] Richard Ekwall and André Schiper. Modeling and validating the performance of atomic broadcast algorithms in high-latency networks. In *Proc. of Euro-Par*, pages 574–586. Springer, 2007.

[34] Richard Ekwall and André Schiper. Modeling and validating the performance of atomic broadcast algorithms in high latency networks. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 574–586. Springer, 2007.

[35] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Trans. Database Syst.*, 22(3):315–363, 1997.

[36] Keir Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003.

[37] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the SIGMOD*, pages 173–182. ACM, 1996.

[38] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[39] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proc. of PPOPP*, 2008.

[40] Rachid Guerraoui and Luis Rodrigues. *Introduction to Reliable Distributed Programming.* Springer, 2006.

[41] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[42] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

[43] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. of the Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, New York, NY, USA, 2003. ACM.

[44] Markus; Leland R. Beaumont: Hofmann. Content networking: Architecture, protocols, and practice.

[45] IBM. Db2: Replication guide and reference, June 1999.

[46] Rebecca Ingram, Patrick Shields, Jennifer E. Walter, and Jennifer L. Welch. An asynchronous leader election algorithm for dynamic networks. In *IPDPS*, pages 1–12, 2009.

[47] Khushboo Kanjani, Hyunyoung Lee, Whitney L. Maguffee, and Jennifer L. Welch. A simple byzantine-fault-tolerant algorithm for a multi-writer regular register. *IJPEDS*, 25(5):423–435, 2010.

[48] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, page 424, Washington, DC, USA, 1999. IEEE Computer Society.

[49] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the The 18th International Conference on Distributed Computing Systems*, page 156, Washington, DC, USA, 1998. IEEE Computer Society.

[50] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[51] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Andre Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TKDE*, 15(4):1018–1032, 2003.

[52] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[53] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Computers*, 48(9):866–880, 1999.

[54] University of Newcastle upon Tyne Mark Little. Javasim 0.3 ga, http://javasim.codehaus.org/, 2008.

[55] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of ICDCS*, pages 707–710, Phoenix, Arizona, April 2001. IEEE.

[56] J. Mocito, A. Respicio, and L. Rodrigues. On statistically estimated optimistic delivery in large-scale total order protocols. In *Proc. of PRDC*, page (accepted for publication), University of California, Riverside, USA, December 2006. IEEE.

[57] Ngoc Thanh Nguyen. Consensus system for solving conflicts in distributed systems. *Inf. Sci. Inf. Comput. Sci.*, 147:91–122, October 2002.

[58] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[59] Marta Patino-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[60] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[61] Fernando Pedone and André Schiper. Optimistic atomic broadcast. In *DISC*, pages 318–332, 1998.

[62] Fernando Pedone and André Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theor. Comput. Sci.*, 291(1):79–101, 2003.

[63] Fernando Pedone, Matthias Wiesmann, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474, 2000.

[64] Francisco Perez-Sorrosal, Marta Pati no-Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. Consistent and scalable cache replication for multi-tier j2ee applications. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 328–347, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[65] Francesco Quaglia. Software diversity-based active replication as an approach for enhancing the performance of advanced simulation systems. *Int. J. Found. Comput. Sci.*, 18(3):495–515, 2007.

[66] T. Ragunathan and P. Krishna Reddy. Improving the performance of read-only transactions through speculation. In *Proceedings of the 5th international conference on Databases in networked information systems*, DNIS'07, pages 203–221, Berlin, Heidelberg, 2007. Springer-Verlag.

[67] P. Krishna Reddy and Masaru Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE TKDE*, 16(2):154–169, 2004.

[68] Luís Rodrigues, Nuno Carvalho, and Emili Miedes. Supporting linearizable semantics in replicated databases. In *NCA*, pages 263–266, 2008.

[69] P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. In *Proc. of the Workshop on Large-Scale Distributed Systems and Middleware*, September 2008.

[70] Paolo Romano, Diego Rughetti, Francesco Quaglia, and Bruno Ciciani. Apart: Low cost active replication for multi-tier data acquisition systems. In *NCA*, pages 1–8, 2008.

[71] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 37–, Washington, DC, USA, 2001. IEEE Computer Society.

[72] Peter G. Sassone and D. Scott Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 7–17, Washington, DC, USA, 2004. IEEE Computer Society.

[73] O. T. Satyanarayanan and Divyakant Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. *IEEE Trans. Knowl. Data Eng.*, 5(5):859–871, 1993.

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

[75] Fred B. Schneider. *Replication management using the state-machine approach.* ACM Press/Addison-Wesley Publishing Co., 1993.

[76] Gurindar S. Sohi and Amir Roth. Speculative multithreaded processors. *IEEE Computer*, 34(4):66–71, 2001.

[77] Scott D. Stoller. Leader election in asynchronous distributed systems. *IEEE Trans. Computers*, 49(3):283–284, 2000.

[78] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25:8–17, June 1992.

[79] Péter Urbán, Ilya Shnayderman, and André Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *DSN*, pages 645–654. IEEE Computer Society, 2003.

[80] Li Wang and Wanlei Zhou. Primary-backup object replications in java. In *In Proc. of TOOLS Asia'98*, pages 78–82. IEEE Computer Society Press, 1998.

[81] Matthias Wiesmann and Andre Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. on Knowl. and Data Eng.*, 17:551–566, April 2005.

[82] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowl. Data Eng.*, 17(4):551–566, 2005.

[83] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *SRDS*, pages 206–215, 2000.