

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

**INSTITUTO SUPERIOR TÉCNICO
UNIVERSIDADE DE LISBOA**

UNDER ERASMUS MUNDUS PROGRAMME

**Speculation in Partially-Replicated Transactional
Data Stores**

Zhongmiao Li

Supervisor: Doctor Peter Van Roy
Co-Supervisor: Doctor Paolo Romano

**Thesis approved in public session to obtain the PhD Degree in
Engineering Science**

Jury final classification: Pass with Distinction

2020

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
INSTITUTO SUPERIOR TÉCNICO
UNIVERSIDADE DE LISBOA

UNDER ERASMUS MUNDUS PROGRAMME

**Speculation in Partially-Replicated Transactional
Data Stores**

Zhongmiao Li

Supervisor: Doctor Peter Van Roy
Co-Supervisor: Doctor Paolo Romano

**Thesis approved in public session to obtain the PhD Degree in
Engineering Science**
Jury final classification: Pass with Distinction

Jury

Doctor Charles Pecheur, Université catholique de Louvain, Belgium

Doctor Peter Van Roy, Université catholique de Louvain, Belgium

Doctor Paolo Romano, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Doctor Bruno Martins, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Doctor Fernando Pedone, Università della Svizzera italiana, Switzerland

Doctor Etienne Rivière, Université catholique de Louvain, Belgium

Doctor Danny Hughes, Katholieke Universiteit Leuven, Belgium

Funding Institutions

European Union
Fundação para a Ciência e Tecnologia

2020



Co-funded by the
Erasmus+ Programme
of the European Union



Speculation in Partially-Replicated Transactional Data Stores

ZHONGMIAO LI

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium 2020
and
Instituto Superior Técnico
Universidade de Lisboa
Lisbon, Portugal 2020

Abstract

The last few decades have witnessed the unprecedented growth of large-scale online services. Distributed data storage systems, which are the fundamental building blocks of large-scale online services, are faced with a number of challenging, and often antagonistic, requirements. On the one hand, many distributed data storage systems have shifted away from weak consistency and embraced strong, transactional, semantics in order to tame the ever growing complexity of modern applications. On the other hand, the need for storing sheer amount of data and serving geo-dispersed clients with low latency has driven modern data storage systems to adopt partial replication techniques, often applied to geo-distributed infrastructures. Unfortunately, when employed in the geo-distributed and/or partial replicated settings, state of the art approaches to enforce transactional consistency suffer from severe bottlenecks that strongly hinder their efficiency.

This dissertation investigates the use of *speculative* techniques to enhance performance of partially replicated transactional data stores, with a focus on geo-distributed platforms. With the term speculation, in this dissertation, we refer to the possibility of exposing the updates produced by uncommitted transactions to other transactions and/or to external clients in order to enhance performance. We apply speculation techniques to two fundamental approaches to develop replicated transactional data stores, namely Deferred Update Replication (DUR) and State Machine Replication (SMR).

In DUR-based systems, transactions are firstly executed in a node and then propagated to other nodes for a global verification phase, during which pre-commit locks have to be held on data items updated by transactions. The global verification phase can throttle system throughput, especially when there is high conflict. We tackle this problem by introducing Speculative Transaction Replication (STR), a DUR protocol that exploits *speculative reads* to enhance performance of geo-distributed, partially replicated transactional data stores. The use of speculative reads greatly reduces the ‘effective duration’ of pre-commit locks, thus removing one of the key bottlenecks of DUR-based protocols. However, the indiscriminate use of speculative reads can expose applications to concurrency anomalies that can compromise their correctness in subtle ways. We tackle this issue by introducing Speculative Snapshot Isolation (SPSI), an extension of Snapshot Isolation (SI), which specifies desirable atomicity and isolation guarantees that must hold when using speculative processing techniques. In a nutshell, SPSI guarantees that, applications designed to operate using SI can safely execute atop STR, sheltering programmers from complex concurrency anomalies and source code modification. Our experimental study shows that STR, thanks to the use of speculative reads, yields up to $11\times$ throughput improvements over state-of-the-art approaches that do not adopt speculative techniques.

In SMR-based systems, transactions first undergo an ordering phase, then replicas have to guarantee that the result of transaction execution is equivalent to a serial execution according to the produced order from the ordering phase. To ensure this guarantee, existing approaches use a single-thread to execute or serialize transactions, which severely limits throughput especially given the current architectural trend towards massively parallel multi-core processors. This limitation is tackled through the introduction of SPARKLE. SPARKLE is an innovative deterministic concurrency control designed for Partially-Replicated State Machines (PRSMs). SPARKLE untaps the potential parallelism of modern multi-core systems through the use of speculative technique and by avoiding inherently non-scalable designs that rely on a single thread for either executing or scheduling transactions. The key contribution of SPARKLE is a set of techniques that can greatly minimize the frequency of misspeculations and the cost associated with correcting them. Our evaluation shows that SPARKLE achieves up to one order of magnitude throughput gains when compared to state of the art systems.

Résumé

Les dernières décennies ont vu une croissance sans précédent de services en ligne à grande échelle. Les systèmes de stockage de données distribués, sur lesquels reposent de tels services en ligne, doivent faire face à un certain nombre d'exigences complexes et souvent antagonistes. D'une part, de nombreux systèmes de stockage de données distribués se sont éloignés de la cohérence faible et ont adopté une sémantique forte et transactionnelle afin de pallier à la complexité toujours croissante des applications modernes. D'autre part, la nécessité de stocker une quantité considérable de données et de servir des clients géographiquement dispersés avec une faible latence a conduit les systèmes de stockage de données modernes à adopter des techniques de réplication partielle, souvent appliquées dans les infrastructures géo-distribuées. Néanmoins, les techniques de pointe permettant d'assurer la cohérence transactionnelle souffrent d'importants goulots d'étranglement dans un cadre géo-distribué ou avec réplication partielle, limitant alors sévèrement leur efficacité.

Cette thèse étudie l'utilisation de techniques *spéculatives* pour améliorer les performances des bases de données transactionnelles partiellement répliquées, en mettant l'accent sur les plateformes géo-distribuées. Par spéculation, nous désignons dans cette thèse la possibilité d'exposer les mises à jour produites par des transactions non validées (uncommitted) à d'autres transactions ou à des clients externes afin d'améliorer les performances. Nous appliquons des techniques de spéculation à deux approches fondamentales pour développer des bases de données transactionnelles répliquées : la Réplication de Mise à jour Différée (RMD) et la Réplication de Machine à État (RME).

Dans les systèmes basés sur RMD, les transactions sont d'abord exécutées dans un nœud, puis propagées à d'autres nœuds pour une phase de vérification globale, au cours de laquelle les sémaphores de pré-commit doivent être maintenus sur les données que les transactions modifient. La phase de vérification globale peut limiter le débit du système, en particulier en cas de conflits importants. Nous abordons ce problème en introduisant la réplication de transaction spéculative (*Speculative Transaction Replication, STR*), un protocole RMD qui exploite les lectures spéculatives pour améliorer les performances des bases de données transactionnelles géo-distribuées et partiellement répliquées. L'utilisation de lectures spéculatives réduit considérablement la 'durée effective' des sémaphores de pré-commit, supprimant ainsi l'un des principaux goulots d'étranglement des protocoles basés sur RMD. Cependant, l'usage sans distinction des lectures spéculatives peut exposer les applications à des anomalies de concurrence qui peuvent compromettre leur exactitude de manière subtile. Nous traitons ce problème en introduisant l'isolation spéculative d'instantané (*Speculative Snapshot Isolation, SPSI*), une extension de l'isolation d'instantané (*Snapshot Isolation, SI*), qui spécifie des garanties souhaitables d'atomicité et d'isolation qui doivent être respectées lors de l'utilisation de techniques de traitement spéculatif. En résumé, SPSI garantit que les applications conçues pour fonctionner avec SI peuvent s'exécuter en toute sécurité au-dessus de STR, mettant les programmeurs à l'abri d'anomalies de concurrence complexes et d'avoir à modifier leur code source. Notre étude expérimentale montre que STR, grâce à l'utilisation de lectures spéculatives, améliore jusqu'à onze fois le débit par rapport aux méthodes récentes qui n'utilisent pas de techniques spéculatives.

Dans les systèmes basés sur RME, les transactions subissent d'abord une phase d'ordonnancement, puis les répliques doivent garantir que le résultat de l'exécution de la transaction est équivalent à une exécution en série suivant l'ordre produit dans la phase d'ordonnancement. Pour assurer cette garantie, les méthodes existantes utilisent un unique thread pour exécuter ou sérialiser les transactions, ce qui limite considérablement le débit, en particulier compte tenu de l'évolution des processeurs vers des architectures multicœurs massivement parallèles. Cette limitation est résolue par l'introduction de SPARKLE. SPARKLE est un algorithme innovant de contrôle de concurrence déterministe conçu pour les machines à état partiellement répliquées. SPARKLE exploite le parallélisme dont sont capables les systèmes multicœurs modernes via l'utilisation de techniques spéculatives et en évitant intrinsèquement les implémentations qui n'utilisent

qu'un seul thread pour exécuter ou planifier des transactions. La principale contribution de SPARKLE est un ensemble de techniques qui peuvent considérablement minimiser la fréquence des erreurs de spéculations et le coût associé à leur correction. Notre évaluation montre que SPARKLE réalise des gains de débit allant jusqu'à un facteur dix par rapport aux systèmes actuels.

Resumo

Ao longo das últimas décadas testemunhou-se um crescimento sem precedentes dos serviços on-line em larga escala. Os sistemas de armazenamento de dados distribuídos, elementos fundamentais deste tipo de serviços, devem satisfazer um conjunto de requisitos desafiantes e muitas vezes antagônicos. Por um lado, devido à necessidade de domar a crescente complexidade das aplicações modernas, muitos destes sistemas abandonaram modelos de consistência fraca em prol de soluções que garantem semântica transacional na gestão de dados. Por outro lado, a necessidade de armazenar grandes quantidades de dados e atender, com baixa latência, a clientes geograficamente dispersos levou os sistemas modernos de armazenamento de dados a adotar técnicas de replicação parcial, frequentemente aplicadas a infraestruturas distribuídas geograficamente. Infelizmente, quando adotadas em sistemas replicados geográfica e/ou parcialmente, as abordagens de última geração para garantir a consistência transacional dos dados sofrem estrangulamentos significativos que as tornam severamente ineficientes.

Esta dissertação investiga o uso de técnicas especulativas para melhorar o desempenho de sistemas de armazenamentos de dados transacionais parcialmente replicados, focando-se em plataformas distribuídas geograficamente. Com o termo especulação, referimo-nos à possibilidade de expor as atualizações produzidas por transações não confirmadas a outras transações e/ou clientes externos, a fim de melhorar o seu desempenho. Aplicamos técnicas de especulação a duas abordagens fundamentais para desenvolver sistemas transacionais replicados, nomeadamente Replicação com Atualização Diferida (RAD) e Replicação de Máquinas de Estados (RME).

Nos sistemas baseados em RAD, as transações são executadas primeiro num nó e depois propagadas para outros nós para uma fase de verificação global. Durante esta fase é necessário manter os trincos de pré-confirmação associados aos dados atualizados pelas transações. A fase de verificação global pode reduzir o débito do sistema, especialmente quando há um elevado número de conflitos. Resolvemos este problema introduzindo “Speculative Transaction Replication” (STR), um protocolo RAD que explora leituras especulativas para melhorar o desempenho de sistemas transacionais parcialmente distribuídos e distribuídos geograficamente. O uso de leituras especulativas reduz significativamente a “duração efetiva” dos trincos de pré-confirmação, removendo assim um dos principais pontos de estrangulamento dos protocolos baseados em RAD. No entanto, o uso indiscriminado de leituras especulativas pode expor as aplicações a complexas anomalias de concorrência. Abordamos este problema introduzindo a “Speculative Snapshot Isolation” (SPSI), uma extensão da semântica de “Snapshot Isolation”, que especifica um conjunto de garantias de atomicidade e isolamento que devem ser respeitadas por técnicas de processamento especulativas. Em resumo, a semântica SPSI garante que as aplicações desenvolvidas para operar com a semântica SI possam executar corretamente usando STR, protegendo os programadores de complexas anomalias de concorrência e evitando a necessidade de qualquer modificação ao próprio código-fonte. O nosso estudo experimental mostra que, devido ao uso de leituras especulativas, o STR pode melhorar o débito até 11 vezes em comparação com abordagens que não adotam técnicas especulativas.

Nos sistemas baseados em RME, as transações passam primeiro por uma fase de ordenação; em seguida, as réplicas precisam de garantir que o resultado da execução da transação é equivalente a uma execução em série e que está de acordo com a ordem produzida na fase de ordenação. Para garantir o cumprimento da ordem, as abordagens existentes usam um único fio de execução para processar as transações, o que limita severamente o débito do sistema, especialmente tendo em conta a tendência atual de adotar arquiteturas de computação baseadas em processadores massivamente paralelos. Esta limitação é resolvida através da introdução do SPARKLE. O SPARKLE é um controlo de concorrência determinístico inovador desenvolvido para máquinas de estado parcialmente replicadas. O SPARKLE permite atingir o potencial de paralelismo disponibilizado pelas arquiteturas de elaboração modernas graças ao uso de técnicas

especulativas e evitando abordagens inerentemente não escaláveis que dependem de um único fio de execução para executar ou escalonar transações. A principal contribuição do SPARKLE consiste num conjunto de técnicas que podem minimizar bastante não só a frequência com a qual a computação especulativa falha, mas também o custo da recuperação de erros induzidos devido a especulações errôneas. A nossa avaliação mostra que o SPARKLE alcança ganhos de débito de até uma ordem de magnitude quando comparado com o atual estado da arte nesta área.

Acknowledgments

I would like to express my sincerest gratitude to many people that companied me along this long and winding journal of PhD.

First, I would like to thank my supervisors Peter Van Roy and Paolo Romano, who enlightened me at so many moments when I was discouraged and lost. Peter has always been supportive and optimistic; his ‘think big’ spirit motivates me to better understand my work and write better papers. Paolo’s critical thinking and high standard prompt me to delve into details of my work and deliver works with higher quality. Without them, I would not be able to finish this thesis.

Also, I would like to thank my parents and my sister. For long time in my PhD, I was highly stressed so I kept working during holidays and could not visit my family. Nevertheless, they were always supportive of my choices and never placed any pressure on me.

Then, to EMDC and EMJDDC programs, and SyncFree and LightKone projects. I am really grateful to be involved in these programs and projects, which allow me to meet so many amazing people, attend meetings in various countries and list to numerous interesting talks.

To the colleagues I met in UCL and INESC (IST). It was a great experience to work with these brilliant, hard-working and funny people of UCL and INESC. I especially miss the time in INESC, where we were like a big family and I had so many great memories of us discussing about research, eating out and going to events.

Finally, to all the friends that I have had great time with during these years. Thanks to Manuel and Alejandro, with who I have been friends for eight years and from whom I have learnt so much. Thanks to Igor for being my bouldering partner in LLN. Thanks to Paolo for your risotto and Italian wine. Thanks to Richard for the great collaboration in research and the Hillsong time. Thanks to Zygi and Julia for lots of help during my job search and hosting me in Zurich for a relaxing weekend. Thanks to Juliana for being my nicest Portuguese friend and taking me to laser tags. Thanks to Husnain for the billiard nights in LLN. Thanks to Ben for being a considerate friend and always willing to host me in Delft. Thanks to Martyna (Monica), Kasia (Kasha), Andrzej (Andrew) and Diana

(Daria) for your awesome Soplica recipes. Finally, thanks to Liza and golden Adrien for your genuine support during my final stage of thesis writing in Vancouver.

This work is partially funded by the LightKone project in the European Union Horizon 2020 Framework Programme under grant agreement 732505, by the Erasmus Mundus Doctorate Programme under grant agreement 2012-0030 and by FCT via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014.

Vancouver, Canada, June 2020

Zhongmiao Li

Contents

List of Figures

List of Algorithms

List of Tables

1	Introduction	1
1.1	Background and motivation	2
1.2	Thesis contributions	3
1.2.1	Summary of contributions	3
1.2.2	Publications	4
1.3	Outline	5
2	Background	7
2.1	Transaction abstraction and consistency models	7
2.1.1	Causal consistency	8
2.1.2	Snapshot Isolation	10
2.1.3	Serializability and Strict Serializability	11
2.1.4	Opacity	13
2.2	System model of replicated transactional data stores	14
2.3	Non-speculative replicated transactional data stores	15
2.3.1	DUR systems	15
2.3.2	SMR systems	20
2.3.3	Summary	23
2.3.4	Comparing DUR and SMR systems	24
2.4	Speculative replicated transactional data stores	25
2.4.1	Speculative DUR systems	26
2.4.2	Speculative SMR systems	28
3	Speculative Transaction Replication	33
3.1	Introduction	33
3.2	System and data model	36
3.3	The SPSI consistency model	38
3.4	The STR protocol	40

3.4.1	Base non-speculative protocol	41
3.4.2	Enabling SPSI-safe speculations	42
3.4.3	Promoting successful speculation via Precise Clocks	43
3.4.4	Pseudo-code of the STR Protocol	46
3.4.5	Dynamically tuning speculation	47
3.4.6	Fault tolerance	49
3.5	Correctness proof	50
3.5.1	Safety	50
3.5.2	Liveness	55
3.6	Ensuring real-time ordering and serializability	57
3.6.1	Real-time ordering	57
3.6.2	Serializability	59
3.7	Related Work	61
3.8	Evaluation	62
3.8.1	Synthetic workloads	64
3.8.2	Macro benchmarks	68
3.9	Summary	70
4	Sparkle	71
4.1	Introduction	71
4.2	System and transaction model	73
4.3	PRSM model	74
4.4	Single partition transactions	76
4.5	Multi Partition Transactions	78
4.6	Read-only Transactions	85
4.7	Correctness proof	85
4.7.1	Single-partition transactions	85
4.7.2	Multi-partition transactions of homogeneous batches.	86
4.7.3	Heterogeneous batches.	88
4.7.4	Real-time ordering	88
4.8	Related Work	89
4.9	Evaluation	90
4.9.1	Implementation and setup	91
4.9.2	Benchmarks	92
4.9.3	Single node deployment	92
4.9.4	Distributed deployment	94
4.10	Summary	98
5	Comparing STR and Sparkle	99
5.1	Comparison of base execution models	99
5.2	Comparison of speculation techniques	100
5.2.1	Internal and external speculation	100
5.2.2	Transparency	100
5.3	Summary	101

6	Conclusion	103
6.1	Limitations and Future work	103
6.1.1	Ensuring transparency for speculative reads across nodes	103
6.1.2	STR	104
6.1.3	SPARKLE	104
6.1.4	Combining STR and SPARKLE	106
6.2	Final remarks	106
	Bibliography	109

List of Figures

2.1	A general system model assumed by typical DUR and SMR systems.	15
3.1	Examples illustrating possible concurrency anomalies caused by speculative reads. $T1$, $T2$ and $T3$ are transactions; $N1$, $N2$ and $N3$ are three nodes that store data items A , B and C , respectively.	35
3.2	History exemplifying indirect conflicts between a local committed transaction, $T1$, and a final committed transaction originated at a different node, $T3$. If $T4$ included both $T1$ and $T3$ in its snapshot, it would violate SPSI property 3.	43
3.3	Exemplifying how replica failure may compromise SPSI-3. LR and PT denote last reader and prepare time, respectively.	50
3.4	Exemplifying how real-time ordering can be violated (a), and how commit delay can avoid the problem (b). $N1$ and $N2$ denote two nodes; $T1$ and $T2$ denote two transactions; arrowed lines denote the progress of time.	60
3.5	An example execution illustrating the <i>write skew</i> anomaly using SI.	61
3.6	The performance of different protocols for two synthetic workloads, representative of a favourable (Synth-A) and an unfavourable (Synth-B) scenario for internal speculation. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.	65
3.7	Normalized throughput with respect to the best performing static configuration. <i>No SR/SR</i> denote enabling/disabling statically speculative reads in STR; <i>Auto</i> denotes the use of the self-tuning technique presented in Section 3.4.5.	67
3.8	The performance of different protocols for three TPC-C workloads. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.	69
3.9	The performance of different protocols for RUBiS. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.	69
4.1	Example of a typical deployment scenario for Sparkle.	74
4.2	Exemplifying the execution of MPTs.	78
4.3	Single node deployment, TPC-C 90% update workload.	93

4.4	8 nodes cluster, synthetic benchmark generating workloads with varying contention level, percentage of MPTs and of dependent transaction. <i>MC</i> stands for medium contention and <i>LC</i> for low contention.	95
4.5	8 nodes cluster, TPC-C workloads.	95
4.6	Throughput of SPARKLE : <i>CC</i> , SPARKLE : <i>SC</i> and SPARKLE : <i>SC</i> + <i>schedule</i> , normalized to that of SPARKLE : <i>Cons</i> , while varying the percentage of MPTs. . . .	96
4.7	The performance of SPARKLE for three workloads varying the scale of the system. . .	97

List of Algorithms

1	Coordinator protocol	45
2	Partition protocol	48
3	Concurrency Control	79
4	Backend protocol	80
5	MPT execution at partition X	82

List of Tables

2.1	The taxonomy of discussed DUR systems.	19
2.2	The taxonomy of discussed SMR systems.	23
2.3	The taxonomy of discussed speculative DUR systems, and STR. We use 2PC to denote two-phase commit	28
2.4	The taxonomy of discussed speculative SMR systems, and SPARKLE. We use Strict SER to denote strict serializability.	31
3.1	Normalized throughput/abort rate of different techniques, varying a transaction's number of keys to update. <i>Physical/Precise</i> denotes the use of Physical Clocks/Precise Clocks; <i>SR</i> denotes that speculative reads are enabled. Throughputs reported in each column are normalized according to the throughput of 'Physical' in that column.	67
5.1	A table summarizing the pros and cons of STR and SPARKLE. ✓ and ✗ means a requirement can be supported by a system or not, respectively. ↑ and ↓ mean performance increases or decreases, respectively; more arrows indicates higher tendency for increasing/decreasing.	102

Chapter 1

Introduction

The last few decades have witnessed the unprecedented growth of large-scale online services: at the time of this writing, Facebook serves more than 2.3 billion active users every month, a number that has grown for 20 times over the last ten years [2]; Google receives and processes over two trillion search requests every year, 200 times more than in 2000 [115]. Consequently, distributed data storage systems, which are the fundamental building blocks of large-scale online services, are faced with a number of challenging, and often antagonistic, requirements.

On the one hand, while several works have proposed to give up consistency (i.e. strongly-consistent transactions) for latency and availability [31, 84], many distributed data storage systems have shifted away from weak consistency and embraced strong, transactional, semantics [26] in order to tame the ever growing complexity of modern applications. In fact, although the adoption of weak consistency models can lead to significant performance gains, it also exposes applications to subtle concurrency anomalies that place heavy burdens on application developers, which are "ultimately not worth the performance gains", as argued by Shute et al. [112].

On the other hand, the need of storing sheer amount of data and serving geo-dispersed clients with low latency has driven modern data storage systems to adopt partial replication techniques, often applied to geo-distributed infrastructures. As its name suggests, in a partial-replication setting, no system node is required to store the full data set. Compared to a full-replication setting, the scalability benefit of partial-replication is two-fold: 1) a system can store more data by simply using more nodes, whereas with full-replication the data set size is limited by the storage capacity of a single node; 2) update transactions can be, at least in theory, only executed by nodes that store data actually accessed during transaction execution — yielding opportunities of higher concurrency and thus higher throughput. Additionally, online services typically deploy system nodes in geographically distributed regions, in order to attain disaster tolerance and serve geo-distributed clients with low latency.

Unfortunately, though, when employed in the geo-distributed and/or partial replicated settings, state of the art approaches to enforce transactional consistency suffer from severe

bottlenecks that strongly hinder their efficiency. Consequently, state of the art transactional data stores can only attain limited throughput falling short in meeting the scalability challenges raised by today's large-scale online services.

The goal of this dissertation is to investigate the use of *speculative* transaction processing techniques to attain higher degrees of concurrency, and ultimately better performance, in partially replicated and geo-distributed transactional data stores. Speculation can be implemented in many different forms; in this dissertation, with the term speculation, we specifically refer to the action of *exposing the state of uncommitted transactions to other transactions in order to enhance performance*.

1.1 Background and motivation

A large body of work have investigated how to build strongly-consistent, yet highly scalable, transactional data stores. Existing systems can be coarsely classified based on whether they adopt the *deferred update replication* (DUR) [15] or the *state-machine replication* (SMR) [75] approaches.

In systems based on the DUR approach [26, 73], the processing of transactions follows an *execute and verify* approach: transactions are first locally executed at a single replica and then globally verified, via an agreement protocol based on consensus [66] and/or Two Phase Commit [50]. The final fate of the transaction depends on the outcome of the global verification phase: if replicas agree to commit the transaction, the updates of this transaction will be applied to all replicas; otherwise, the updates are discarded and the transaction is re-executed.

Systems based on the SMR approach [122, 38, 16] follow an *agree then execute* paradigm. Specifically, replicas first agree on the serialization order of transactions, using some form of consensus-based coordination scheme, which ensures that all correct replicas agree on the set and order of transactions. Then, each replica executes the set of transactions independently, using a deterministic concurrency control to ensure a common serialization order at all replicas. To enforce the common serialization order, state of the art solutions typically rely on a single thread to execute [38, 16] or serialize [122] transactions. As long as the transaction logic is deterministic (which is a prerequisite of SMR systems), all replicas will traverse the same sequence of state and reach the same final state.

Although both are widely used, the DUR and SMR approaches are essentially two alternative approaches to build consistent replicated data stores, which have different pros and cons and thus are suitable for different workloads. Before presenting a more comprehensive review of the most representative solutions adopting either the DUR or the SMR approach (see Chapter 2), we briefly list their pros and cons as follows:

- in DUR systems, pre-commit locks on data items have to be held during the whole duration of distributed validation phase, blocking other transactions from accessing these data items, i.e., the lock convoy effect [124]. In fact, the lock convoy effect can be dramatically amplified due to the presence of geo-distribution, as replicas can be remotely dispersed in geographical scale, incurring hundreds of milliseconds of

communication delay. Consequently, the maximum concurrency achievable by the system can be severely throttled, especially in contention prone workloads.

- the correctness of the SMR approach hinges on the assumption that the execution of transactions is deterministic, i.e., transactions observe the same state and produces the same updates/output at every replica. This assumption is not necessary for DUR, where a transaction is executed only at a single replica and its updates are then applied to the other replicas (provided that the transaction passes a distributed validation phase). The need to ensure deterministic transaction execution has led existing SMR approaches that support partial replication to adopt a single-threaded execution model [16, 86]. This approach precludes any non-deterministic effect due to the concurrent execution of transactions. However, it also precludes the possibility to take advantage of the parallelism potential of modern multi-core architectures.

1.2 Thesis contributions

1.2.1 Summary of contributions

This thesis makes two main contributions that aim at addressing the shortcomings of state of the art DUR and SMR solutions for partially-replicated transactional data stores:

1. For DUR systems, whose main bottleneck is long lock duration, this thesis proposes a, so called, *speculative read* mechanism, which allows transactions to observe the data item versions produced by pre-committed transactions, instead of blocking until these are committed or aborted. This leads to the first contribution of this thesis, namely Speculative Transaction Replication (STR) [82]. The use of speculative reads greatly reduce the ‘effective duration’ of pre-commit locks, thus removing what is arguably one of the key bottlenecks of DUR-based protocols, especially in geo-distributed settings. Furthermore, STR proposes Speculative Snapshot Isolation (SPSI), in order to tackle the concurrency anomalies that can arise due to the indiscriminate use of speculative reads. SPSI is a variant of Snapshot Isolation (SI) that provides clear and rigorous guarantees on the atomicity and isolation of the snapshots observed and produced by *executing transactions*. We present the correctness arguments which proves that STR does not violate SPSI properties, and also provide discussions on how to extend STR to ensure additional guarantees, e.g., serializability and real-time ordering. The experimental study shows that STR, thanks to the use of speculative reads, yields up to $11\times$ throughput improvements over state-of-the-art approaches that do not adopt speculative techniques.
2. For SMR systems, we apply speculation techniques to enhance the parallelism of the deterministic concurrency control. Specifically, transactions can be processed ‘out of order’, i.e., to be tentatively executed in a serialization order that may potentially differ from the established total order, but misspeculations (i.e. transactions indeed executed in a wrong order) are detected and fixed in an efficient fashion. This leads to second contribution of this thesis, SPARKLE, an innovative deterministic concurrency

control designed for Partially-Replicated State Machines (PRSMs) [79]. Thanks to the use of speculation, SPARKLE takes full advantage of modern multi-core microprocessors, by avoiding inherently non-scalable designs that rely on a single thread for either executing [16] or scheduling transactions [122]. The key contribution of SPARKLE is a set of techniques that can greatly minimize the frequency of misspeculations and the cost associated with correcting them, including speculative confirmation and scheduling techniques. We prove that SPARKLE’s concurrency control is correct, and show that SPARKLE achieves up to an order of magnitude throughput gain than state of the art systems.

1.2.2 Publications

The results presented in this thesis have led to several publications [82, 81, 79], which are listed as follows:

- **Zhongmiao Li**, Peter Van Roy, and Paolo Romano. Transparent speculation in geo-replicated transactional data stores. In *Proceedings of the 27th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC’18, pages 255-266. [Conference paper]
- **Zhongmiao Li**, Peter Van Roy, and Paolo Romano. Exploiting speculation in partially replicated transactional data stores. In *Proceedings of the 2017 ACM Symposium on Cloud Computing*, SOCC’17, pages 640-640. [Abstract]
- **Zhongmiao Li**, Paolo Romano and Peter Van Roy. Sparkle: Speculative Deterministic Concurrency Control for Partially Replicated Transactional Data Stores. In *Proceedings of the 49th IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN’19, pages 164-175. [Conference paper]
- **Zhongmiao Li**, Paolo Romano and Peter Van Roy. Transparent speculation in geo-replicated transactional data stores. *Journal of Parallel and Distributed Computing*. [Journal paper]

During my doctoral studies, I have collaborated with other researchers and worked on other projects, some of which led to academic papers published in international workshops or conferences. While these projects are not directly related with my thesis, they allowed me to explore relevant fields and deepen my knowledge in the area of distributed systems. These publications [19, 5, 80, 88, 18] are listed below:

- Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, **Zhongmiao Li**, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems*, ICDCS’16, pages 405-414. [Conference paper]

- Richard Gil Martinez, **Zhongmiao Li**, Antonia Lopes, and Luis Rodrigues. Augure: Proactive reconfiguration of cloud applications using heterogeneous resources. In *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications*, NCA'17, pages 73-80. [Conference paper]
- **Zhongmiao Li**, Peter Van Roy, and Paolo Romano. Enhancing throughput of partially replicated state machines via multi-partition operation scheduling. In *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications*, NCA'17, pages 261-270. [Conference paper]
- Iwan Briquemont, Manuel Bravo, **Zhongmiao Li**, and Peter Van Roy. Conflict-free partially replicated data types. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*, CloudCom'15, pages 282-289. [Conference paper]
- Manuel Bravo, **Zhongmiao Li**, Peter Van Roy, and Christopher Meiklejohn. Derflow: distributed deterministic dataflow programming for Erlang. In *Proceedings of the 13th ACM SIGPLAN workshop on Erlang*, pages 51-60. [Workshop paper]

1.3 Outline

The remainder of this dissertation is organized as follows.

Chapter 2 provides the background on partially-replicated, geo-distributed transactional data stores. We first introduce the notion of transactions and several relevant consistency criteria. Then, we present the main approaches to implement replicated transactional data stores, i.e., DUR and SMR approaches, and describe several state-of-the-art protocols of each approach which do not employ speculative techniques. Finally, we describe representative DUR/SMR protocols that employ speculative mechanisms.

Chapter 3 presents STR, a DUR transactional system that exploits speculation techniques to enhance performance of geo-distributed, partially replicated transactional data stores.

Chapter 4 presents SPARKLE, an innovative deterministic concurrency control that enhances the throughput of state of the art partially-replicated state machine systems by more than one order of magnitude, on standard benchmarks.

Chapter 5 strives to shed light on the different pros and cons of STR and SPARKLE, by highlighting their suitable workloads and applications.

Chapter 6 concludes this dissertation with our contribution, the limitations of the current work, possible future directions and final remarks.

Chapter 2

Background

This chapter covers the background knowledge on large-scale replicated transactional data stores. First, Section 2.1 introduces the notion of transactions and describe some major consistency criteria used by transactional applications, some of which will be also targeted by the contributions of this dissertation. Section 2.2 describes the general system model assumed by typical replicated transactional data stores, including DUR/SMR systems that we will discuss later. Section 2.3 and 2.4 are devoted to overviewing existing approaches of replicated transactional data stores. In both sections, existing solutions are classified as either instances of the DUR or of the SMR approach. However, Section 2.3 focuses on non-speculative systems, i.e., systems in which the effects of uncommitted transactions are not observable to neither other transactions nor external clients, whereas Section 2.4 analyzes replication schemes that adopt speculative transaction processing techniques.

2.1 Transaction abstraction and consistency models

Transaction was first developed in the context of database management systems as a paradigm to regulate concurrent access to data, which abstracts the internal complexity of database systems and maintains the consistency of data even in the presence of concurrent accesses. A transaction is a sequence of read and write operations on data items, preceded by a begin operation and followed by either a commit or an abort operation, and a partial order over this set of operations [15]. We say that two operations are conflicting if they access a common data item x and at least one of them is a write. If two operations are conflicting and both of them are write operations, we call this a *write-write conflict*. We capture what happens in an execution of a transactional system via the notion of *history*. A history $\mathcal{H}$ over a set of transactions $T_1, \dots, T_n$ is a partial order $\prec_{\mathcal{H}}$ defined over the set of operations $OP_{\mathcal{H}} = \bigcup_{i=1}^n OP_i$ such that $\prec_{\mathcal{H}}$ preserves the ordering of the operations of each transaction T_i ($\prec_{\mathcal{H}} \supseteq \bigcup_{i=1}^n OP_i$); and for any two conflicting operations $op_i, op_j \in \mathcal{H}$, either $op_i \prec_{\mathcal{H}} op_j$ or $op_j \prec_{\mathcal{H}} op_i$ [15].

Modern transactional data storage systems support various consistency levels to allow programmers to trade off consistency and ease of programming for a potential gain in per-

formance. One of the most widely adopted consistency criteria is *serializability* [15]¹. Serializability ensures that the outcome of transaction execution (i.e., data store state and values read by transactions) is equivalent to *some* serial execution (i.e., without overlapping in time) of the same set of transactions. Note that the order of transactions in the equivalent serial execution may not be follow the order transactions were submitted (and thus expected) by clients, which may, in some settings, lead to counter-intuitive results. For instance, a client’s transaction may not observe the updates of his previously committed transaction. Strict serializability [94] (also named external consistency [47] or linearizability [61]) is an extension to serializability, which precisely tackles this problem. Apart from ensuring that the outcome of transaction execution is equivalent to some serial execution, strict serializability additionally ensures that if a transaction commits at real time t , its effects (i.e., its updates) must be visible to any transaction starting after real time t (i.e., real-time order, which we introduce more formally later). Strict serializability essentially provides the illusion that a transaction is executed on a single-threaded, non-replicated system. Several weaker consistency criteria are also widely adopted, e.g., causal consistency [5, 35], read-committed [73] and snapshot isolation [36, 30]. Generally, the weaker a consistency criterion is, the higher performance it is expected to achieve, but also the larger set of concurrency anomalies it exposes.

The above discussed consistency criteria focus on providing certain guarantees to *committed transactions*, but do not provide consistency guarantees to provide to *abort transactions*. Nevertheless, this means aborted transactions may observe arbitrary snapshots, which can endanger application’s correctness, e.g., leading to divisions by zeros, infinite loops. *Opacity* [53] extends strict serializability and tackles these problems by ensuring that even aborted transactions will also only observe a state that could have been produced by a history ensuring serializability. Note that although opacity was defined by extending strict serializability, its spirit can be applied to other consistency criteria (as our work in Chapter 3), i.e., to ensure that these consistency criteria do not expose arbitrary snapshots even to aborted transactions.

In the following text, we overview several representative consistency criteria adopted by today’s replicated data storage systems, namely, *causal consistency*, *snapshot isolation*, *serializability*, *strict serializability* and *opacity*. Each consistency criterion is stronger than the previous one, thus exposing less anomalies. We introduce the detailed definitions of these consistency consistency criteria, and also discuss the anomalies they expose.

2.1.1 Causal consistency

Let us start with causal consistency, the strongest consistency criterion that is compatible with availability [8]. On one hand, causal consistency does not require replicas to undergo a blocking synchronization phase upon updates – instead, replicas asynchronously propagate updates and converge to the same state, which allows for high availability and low latency. On the other hand, causal consistency imposes ordering constraints among opera-

¹While there exist some variations of serializability [15], in this dissertation, the term ‘serializability’ specifically refers to ‘conflict serializability’ defined in the literature.

tions that have causal dependencies (to be defined more precisely soon), thus is still deemed to provide intuitive results to clients and sufficient for certain applications.

Causal consistency was first defined to model the causal relation between a set of individual operations [74]. Recently, Transactional Causal Consistency (TCC) was introduced [5, 126], which extended causal consistency to scenarios where operations are grouped within transactions. For the sake of self-containment, in the following, we first define the notion of causal consistency for individual operations, and then give the definition of transactional causal consistency.

Causal consistency is defined as follows. For two operations a and b , we say that b causally depends on a , i.e., $a \rightsquigarrow b$, if and only if at least one of the following conditions holds:

1. **Execution thread.** a and b are operations in a single thread of execution, and a happens before b ;
2. **Gets from.** a is a write operation, b is a read operation, and b reads the version written by a ;
3. **Transitivity.** There is some other operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$. Intuitively, this property ensures that if a client has seen the effects of b and $a \rightsquigarrow b$, then the client also sees the effects of a .

Before presenting the definition of TCC, let us clarify how to apply the above three conditions for establishing causal-dependency, which were defined for single-item operations, to transactions. For instance, if a transaction T_i reads from another transaction T_j (the *Gets from* rule), we consider T_i to be causally-dependent on T_j and write $T_i \rightsquigarrow T_j$. This is essentially equivalent to writing that $x_i \rightsquigarrow y_j$, where x_i and y_j are two data item versions created respectively by T_i and T_j .

TCC enforces the following two properties:

1. **Transactions read from causally-consistent snapshots.** A transaction T read from a causally-consistent snapshot, which represents a logical copy of the data store that includes the effects of all transactions that causally precede T [37, 84]. Specifically, for any two item versions, x_i and y_k , if $x_i \rightsquigarrow y_k$ and both x_i and y_k belong to the same causal snapshot, then there is no version of x , denoted as x_j , such that $x_i \rightsquigarrow x_j \rightsquigarrow y_k$.
2. **Transaction's updates are atomic.** Either all items written by a transaction are visible to other transactions, or none is. Namely, if a transaction writes x and y , then any snapshot visible to other transactions either includes both x and y or neither one of them.

TCC exposes a number of anomalies, including the well-known ‘lost update’ anomaly [12]. We explain how this anomaly could happen using the UPLOAD_PICTURE

shown in Listing 2.1. This transaction adds a picture to an album, by appending the URL of the new picture to the list of URLs of existing pictures. As already mentioned, causal consistency (and also TCC) does not require replicas to synchronize upon updates, thus two clients can concurrently execute the UPLOAD_PICTURE transaction and both commit. Nevertheless, as these two transactions are concurrent, only the update of one transaction can be preserved but the other transaction’s update will be lost.

```
// A transaction that uploads a picture to a user's album.
Bool upload_picture(Album album, URL picture_url) {
    List<String> existing_pics = read(album);
    existing_pics.append(picture_url);
    write(album, existing_pics);
    return committed;
}
```

Listing 2.1 – An upload picture transaction that may cause a ‘lost update’ anomaly if concurrently executed.

2.1.2 Snapshot Isolation

Snapshot Isolation [12] (SI) is a popular consistency level provided by many commercial database systems, such as Oracle and SQLServer. SI was first defined in a rather informal fashion [12], which triggered a myriad of work to study the properties of SI and propose new definitions (including some variants) [41, 36, 70, 7]. While the original definition of SI requires transactions to read from the latest snapshot when transactions start (i.e., real-time order as we will discuss later), many recent proposals relaxed this guarantee, as ensuring such a property can impose large costs in distributed settings [41]. Here, we present the definition of SI adopted by Clock-SI [36], a recent protocol that provides SI for partitioned data stores.

More precisely, SI is a consistency criterion defined via three properties:

1. Each transaction reads from a consistent snapshot, taken at the start of the transaction and identified by a snapshot timestamp. A consistent snapshot includes all writes of transactions committed before the snapshot timestamp, and does not include any writes of aborted transactions or transactions committed after the snapshot timestamp.
2. Update transactions commit in a total order. Every commit produces a new database snapshot, identified by the commit timestamp.
3. An update transaction aborts if it introduces a write-write conflict with a concurrent committed transaction. Transaction T_i is concurrent with committed update transaction T_j , if T_i took its snapshot before T_j committed and T_i tries to commit after T_j committed.

By enforcing Property 3 we defined above, SI avoids the lost update anomaly that is exposed by causal consistency and exemplified in Listing 2.1. In practice, this is typically implemented by requiring transactions to undergo some form of certification check, which

only allows a transaction to commit if it does not have write-write conflict with concurrent committed transactions.

Nevertheless, SI is known to suffer from an anomaly named as write skew [12]. Considering the two transactions in Listing 2.2: both transactions try to withdraw money from a user's credit or saving account, provided that the total balance of the user's credit and saving accounts are above the amount to be withdrawn. If these two transactions are concurrently executed, since they do not have write-write conflicts, they can both commit, causing the person's balance to become negative. This execution would never be produced had the two transactions been executed serially (in any order).

```
// A transaction that withdraws money from a user's credit account.
Bool credit_withdrawal(Key user_credit, Key user_saving, Int amount) {
    Int credit_balance = read(user_credit);
    Int saving_balance = read(user_saving);
    if (credit_balance + saving_balance >= amount) {
        write(user_credit, credit_balance - amount);
        return committed;
    }
    return aborted;
}

// A transaction that withdraws money from a user's saving account.
Bool saving_withdrawal(Key user_credit, Key user_saving, Int amount) {
    Int credit_balance = read(user_credit);
    Int saving_balance = read(user_saving);
    if (credit_balance + saving_balance >= amount) {
        write(user_saving, saving_balance - amount);
        return committed;
    }
    return aborted;
}
```

Listing 2.2 – Exemplifying transactions that may generate non-serializable execution under SI.

Note that in spite of allowing the write skew anomaly, certain applications can execute under SI without any serialization anomalies [42]. As such, SI is deemed to be sufficient for a wide range of applications, and is the strongest consistency level provided by some database systems [102]. Moreover, recent works have proposed several approaches to spare applications from experiencing such anomaly when being executed in SI-based systems, ranging from modifying applications [42] to extending the certification check [102].

2.1.3 Serializability and Strict Serializability

As already described, serializability ensures that the outcome of transaction execution is equivalent to the outcome of some serial execution. More formally, a history represents a serializable execution if it is equivalent to a serial history. Thus, we first define conditions under which a history is serial, then define conditions under which two histories are equivalent [15].

A history $\mathcal{H}$ is serial, if for every two transactions T_i and T_j that appear in $\mathcal{H}$, either all operations of T_i appear before all operations of T_j , or vice versa. Basically, a serial history represents an execution in which operations of different transactions do not interleave.

Next, two histories $\mathcal{H}$ and $\mathcal{H}'$ are equivalent, if:

1. They are defined over the same set of transactions and have the same operations.
2. They order conflicting operations of non-aborted transactions in the same way. Namely, for any conflicting operations op_i and op_j belonging to transactions T_i and T_j , respectively, if $op_i \prec_{\mathcal{H}} op_j$ then $op_i \prec_{\mathcal{H}'} op_j$.

A system providing serializability does not expose the write-skew anomaly described in Section 2.1.2, nor the lost update anomaly mentioned in Section 2.1.1. It is easy to see that any execution that exposes these anomalies can not be equivalent to a serial execution history, thus would not be produced by a serializable system.

Yet, as already mentioned, serializability can still suffer from anomalies since it allows committed transactions to be re-ordered with respect to real-time, i.e., it may violate real-time order. We provide an example illustrating this type of anomalies in Listing 2.3, which models a simple photo managing system. Imagine the following scenario: a user first issues MODIFY_PRIVACY to make one of his album private, and then uploads a private picture to the album by calling UPLOAD_PICTURE; meanwhile, another user tries to browse pictures of the album using the FETCH_PICTURES transaction. As serializability does not guarantee that UPLOAD_PICTURE is ordered after MODIFY_PRIVACY even if the former is submitted after the latter is committed, FETCH_PICTURES may fetch the user's private picture due to not observing the effect of MODIFY_PRIVACY.

Here, we define Real-Time Order (RTO) as follows. RTO is a partial order defined over a transaction history $\mathcal{H}$, noted $\prec_{\mathcal{H}}^{RTO}$, which reflects the happen-before relations among transactions in a history. A transaction T_i is ordered before T_j in RTO, $T_i \prec_{\mathcal{H}}^{RTO} T_j$, if the commit operation c_i of T_i precedes the begin operation b_j of T_j in real-time.

```
// A transaction that updates a user album's privacy setting.
Bool modify_privacy(PrivacySetting album_privacy, Int privacy_level) {
    write(album_privacy, privacy_level);
    return committed;
}

// A transaction that uploads a picture to a user's album.
Bool upload_picture(Album album, URL picture_url) {
    List<String> existing_pics = read(album);
    existing_pics.append(picture_url);
    write(album, existing_pics);
    return committed;
}

// A transaction that fetch pictures from a user's album.
List<String> fetch_pictures(Album album, PrivacySetting album_privacy) {
    Int privacy_level = read(album_privacy);
    if (privacy_level == PUBLIC) {
        return read(album);
    } else {
        return "";
    }
}
```

Listing 2.3 – Exemplifying transaction execution that may produce unexpected results due to real-time order violations.

Strict serializability tackles precisely this issue, by extending serializability in order to enforce real-time order and avoid the type of anomalies exemplified above. A history $\mathcal{H}$

enforces strict serializability, if it satisfies an additional constrain with respect to serializability. Basically, let $\mathcal{H}'$ be the serial history to which $\mathcal{H}$ is equivalent. For any pair of transactions T_i and T_j , if the commit operation c_i of T_i happens before the begin operation b_j of T_j in *real time*, then T_i must be ordered before T_j in the serial history $\mathcal{H}'$. This avoids the above anomaly, because if `UPLOAD_PICTURE` begins after `MODIFY_PRIVACY` commits in real time, then `UPLOAD_PICTURE` must be ordered after `MODIFY_PRIVACY`. Consequently, `UPLOAD_PICTURE` can never solely observe the effect of `UPLOAD_PICTURE` without observing the effect of `MODIFY_PRIVACY`.

2.1.4 Opacity

Opacity is a safety property that captures the intuitive requirements that (1) all operations performed by every *committed* transaction appear as if they happened instantaneously at some point during the transaction lifetime, (2) no operation performed by any *aborted* transaction is ever visible to other transactions (include live ones), and (3) every transaction always sees a *consistent* state of the system.

In order to introduce a formal definition of opacity, it is necessary to first introduce some preliminary notions.

First, a history $\mathcal{H}$ is said to be complete, if $\mathcal{H}$ does not contain any transaction that has not completed (i.e., committed or aborted). If a history $\mathcal{H}$ is not complete, then we can transform it to a complete history $\mathcal{H}'$ by aborting or committing the uncompleted transactions in $\mathcal{H}$. We denote the set of complete histories that can be obtained from an incomplete history $\mathcal{H}$ as $Complete(\mathcal{H})$.

Next, let us introduce the concept of a *sequential specification* to describe the semantics of shared objects, as in [61, 123]. Intuitively, a sequential specification of a shared object *ob* lists all sequences of operation executions on *ob* that are considered correct when executed outside any transaction context in a single-threaded application.

Finally, let $\mathcal{S}$ be any complete serial history. For each transaction T_i in $\mathcal{S}$, let $visible_{\mathcal{S}}(T_i)$ denote the largest subsequence $\mathcal{S}'$ of $\mathcal{S}$, such that, for every transaction $T_k \in \mathcal{S}'$, either (1) $k = i$, or (2) T_k is committed and $T_k \prec_{\mathcal{S}} T_i$. Then, we say that a transaction $T_i \in \mathcal{S}$ is legal in $\mathcal{S}$, if history $visible_{\mathcal{S}}(T_i)$ is legal. Note that we say history $visible_{\mathcal{S}}(T_i)$ is legal, if $visible_{\mathcal{S}}(T_i)$ respects the sequential specifications of all the shared objects, operations on which are performed in $visible_{\mathcal{S}}(T_i)$.

We consider a history to $\mathcal{H}$ be *opaque*, if there exists a serial history $\mathcal{S}$ equivalent to some history in set $Complete(\mathcal{H})$, such that:

1. $\mathcal{S}$ preserves the real-time order of $\mathcal{H}$, and
2. Every transaction $T_i \in \mathcal{S}$ is legal in $\mathcal{S}$.

2.2 System model of replicated transactional data stores

This section describes the general system model assumed by typical DUR and SMR systems. For the sake of generality, we assume a partially-replicated model, as full replication can be viewed as a special case of partial replication.

In fact, although we have briefly mentioned that partial replication can generally achieve better scalability than full replication, both full and partial replication are widely adopted by various systems and no model is better than the other in all workloads/scenarios. Here we highlight the major differences between these two models. As already discussed in Chapter 1, full replication incurs clear scalability limitations related with data capacity and the handling of update workloads. Basically, in fully-replicated transactional systems, all replicas have to store the full data set and to apply all update operations. Thus, adding more replicas to the system does not permit a system to store a larger data set or sustain higher rates of update transactions. While partial replication addresses these issues, it can also bring additional overheads. Essentially, the fact that data are scattered across multiple nodes means that transactions may have to access data stored in multiple partitions. This reflects into an increase of the transaction execution and certification time with respect to full replicated systems.

As show in Figure 2.1, we assume that the full data set is sharded into multiple disjoint data *partitions*. Overall, a data partitioning scheme strives to achieve the following goals: i) maximizing data locality by storing data in the node that access them more frequently, in order to reduce the chance of accessing remote objects, and ii) maximizing lookup speed, by ensuring that data can be located as quickly as possible, and iii) balancing loads across partitions, in terms of both storage capacity and access rate to data items. On one hand, classical approaches, e.g., partitioning data in a round-robin approach or using hash functions [34], can provide optimal lookup speed (as the location of an object can usually be deterministically calculated without querying intermediate nodes), but can not guarantee good data locality. On the other hand, some systems [25, 22] rely on a logical directory service (implemented, e.g., using a central server or fully replicating the directory across each node), and nodes have to query the directory for object location. While these approaches can achieve better data locality, the use of the directory service can introduce additional overheads, e.g., additional round-trip delays to access a centralized server or keeping the directory's state updated at every node if the directory is fully replicated. Recently, several works propose solutions that provide fast data lookup while also optimizing data locality [29, 89]. In this dissertation, we consider data partitioning an orthogonal problem and assume workloads present reasonable good data locality. In practice, as we will show in Section 3.8 and Section 4.9, using simple data partitioning schemes is sufficient to achieve good data locality for some applications.

The data of each partition is replicated across multiple nodes, which is referred to as the *replication group* of that partition. In the figure, we show that each replication group consists of two nodes; nevertheless, in principle, different replication groups can consist of different numbers of nodes, depending on various factors, such as the required level of fault-tolerance and the load pressure for the replication group [26].

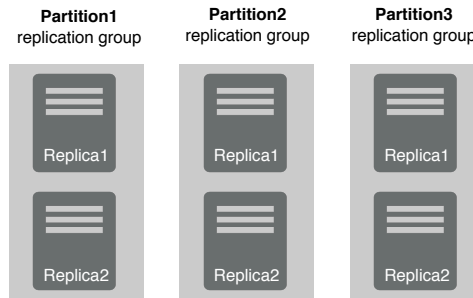


Figure 2.1 – A general system model assumed by typical DUR and SMR systems.

2.3 Non-speculative replicated transactional data stores

This section serves to overview the state of the art on DUR and SMR systems, which can be seen as two complementary approaches to implement replicated transactional systems. Note that in this section we focus on *non-speculative* systems; speculative DUR/SMR systems (i.e., system that, based on the definition provided in Chapter 1, expose the state produced by uncommitted transactions to other transactions and/or to external clients) are discussed in Section 2.4. We provide an overview of DUR/SMR systems in Section 2.3.1 and Section 2.3.2, respectively.

2.3.1 DUR systems

In this sub-section, we first describe the general transaction execution model of a DUR system, which follows the *execute-then-verify* model. Next, we discuss several representative instances of the DUR approach.

General execution model

1. **Execute:** a transaction is first executed in one system node. In this state, the updates produced by transactions are applied to a private buffer, invisible to any other transaction. Read operations are first redirected to the transaction's private buffer, in order to read the transaction's previous updates. If the transaction has not updated the data item, the read is sent to some (possibly local) replica of the required data item. The selection of the replica to be contacted depends on the specific protocol being employed, e.g., selecting the replica with the lowest latency [118] or always selecting a special node, typically designated as master replica of the partition's replication group [30].
2. **Verify:** When the transaction finishes its execution, the transaction's metadata are propagated to remote replicas for verification/certification. The verification phase should ensure that all involved partitions, as well as their replicas, agree to either commit or abort the transaction. A typical approach to ensure this property is to first execute a two-phase commit protocol among all involved partitions, and then rely on

each partition to replicate its received transaction metadata and certification decision in its replication group.

Though, note that prior to the verification phase, the originating node typically already employs some local concurrency control mechanisms, e.g., two-phase-locking [51] or timestamp-based concurrency control [14], such that only transactions not conflicting with other locally originated transactions undergo the (global) verification phase.

Depending on the consistency guarantees to be enforced, different metadata of the transaction, e.g., write-set and read-set, need to be propagated. For instance, if the system enforces snapshot isolation, transmitting the read-set of transactions is normally omitted, as the verification phase only needs to check for write-write conflicts among concurrent transactions. Supporting serializability, conversely, typically requires transmitting both the read-sets and write-sets of transactions.

Examples of Representative DUR systems

To better understand the inherent commonalities as well as the differences between existing DUR systems, we classify and present them according to the following three dimensions:

1. The *replication model* they target, i.e., whether it supports full or partial replication. The replication mode (full versus partial) can have a strong impact on the scalability of the transactional platform, as discussed in Chapter 1 and Section 2.2.
2. The *consistency criterion* they support. The consistency criterion is important, as it has crucial implication both on the ease of development and on performance – enforcing more stringent consistency level typically simplifies development at the cost of introducing additional overheads (e.g., in terms of transactional aborts).
3. The *certification method* they employ, which has a strong impact on performance, as, for instance, it can determine how long pre-commit locks have to be held and also directly affects latency (as discussed in Section 1.1).

Next, we first discuss several DUR systems that adopt the *full replication* model, and then discuss systems that adopt partial replication. While discussing each system, we further highlight their adopted consistency criteria and certification methods.

Fully-replicated DUR systems. In the following, we overview several representative DUR systems that target full replication. Specifically, the solutions we review either adopt serializability, SI without RTO or SI with RTO as consistency criteria. In terms of certification methods, some of them employ atomic broadcast and some simply rely on a centralized process to certify transactions.

Pedone et al. [97] propose a DUR approach, which, according to the above classification, targets full replication, adopts serializability as its consistency criterion and employs a certification method based on atomic broadcast. The system relies on atomic broadcast to

certify transactions [70]. Atomic broadcast protocols, in a nutshell, ensure that all correct replicas deliver the same set of messages in the same order, which greatly simplifies the design of the certification logic as we will show. In this work, each replica locally allows multiple threads to execute transactions, and relies on two-phase locking to synchronize the concurrent execution of transactions. When a transaction finishes execution, if it is a read-only transaction, it can be immediately committed. Otherwise, the transaction's read-set and write-set are propagated to other replicas using atomic broadcast protocols. Each replica relies on a single certification thread to certify whether the transaction has read-write conflicts (as this solution enforces serializability) with concurrent committed transactions, and, in the absence of conflicts, the transaction's write-set is applied. Since all replicas are guaranteed to deliver the certification requests of transactions in the same order (as these are disseminated via atomic broadcast), all replicas can certify each transaction unilaterally without further communication. In fact, replicas are guaranteed to reach the same state, as the certification phase is made deterministic due to the use of a single certification thread.

Besides serializability, SI has also been targeted by several DUR systems adopting full replication. An instance of such a type of DUR protocols is SI-rep [83]. The protocol operates as follows. First, a transaction is executed at the replica where it was originated, where it observes the snapshot locally stored by the replica at the moment in which the transaction started. Upon certification, the transaction's write-set is propagated to other transactions using atomic broadcast, and is then certified by each replica independently. Note that the updates produced by a transaction that was (recently) committed at node N , can be missed by a transaction that starts later in time at a different node N' . This can happen in case N' is lagging behind in applying the stream of (totally ordered) commits disseminated via the atomic broadcast channel. As such, this approach does not guarantee real-time order.

Elnikety et al. [41] discuss the tradeoffs associated with enforcing real-time order for SI in replicated settings, and propose two protocols for enforcing SI targeting full replication, a decentralized protocol and a centralized one. The decentralized protocol is similar in spirit to the one in [83], thus here we only describe the centralized protocol. This solution still allows transaction to be executed at any replica of the system, but relies on a master replica to certify transactions. The protocol can switch between providing real-time order or not, depending on the techniques used by transactions to establish their snapshots. If a transaction establishes its snapshot at the master replica, then the transaction is guaranteed to observe the globally latest snapshot and, thus, enjoy real-time order; otherwise, if a transaction takes its snapshot at the replica where the transaction is submitted to, it is only guaranteed to observe the last snapshot locally committed at the time in which it starts. In either case, the transaction's metadata are propagated to the master replica for certification and the certification decision is sent back to the originating replica.

Partially-replicated DUR systems. In the following, we overview several DUR systems that adopt partial replication. These systems target SI, serializability and strict serializability as consistency criteria; as for the certification methods, the reviewed solutions employ atomic broadcast, atomic multicast and two-phase commit with Paxos as certification meth-

ods.

Serrano et al. [111] propose a partially-replicated DUR system providing SI. Similar to several aforementioned protocols for full replication, this protocol leverages atomic broadcast primitives for certifying transactions, i.e., after a transaction is locally executed, the transaction’s metadata is propagated to all nodes (i.e., *all* replicas of *all* partitions) in the system. Note that although the system adopts a partial-replication model, i.e., each node only hosts a subset of the full data set, broadcasting transaction metadata to all system nodes ensures that all nodes are aware of all transactions. Thus, each node can decide to commit a transaction or not using solely local information without requiring extra communication. As such, the certification of transactions only requires the communication delay of one atomic broadcast. However, this approach has a fundamental scalability limitation: since update transactions have to be sent to all system nodes for certification, the cost of certification increases linearly with the scale of the system. This is at odds with one of the main motivations of partial replication, which strives to improve system throughput by allowing nodes to store a subset of the full data and evenly distribute load among nodes.

P-Store [108] tackles the scalability limitation of the above system by introducing the notion of *genuine partial replication* (which extends the notion of *genuine atomic multicast* originally introduced by Guerraoui and Schiper [55]). Specifically, a genuine partially-replicated data store must ensure that for any submitted transaction T , only nodes that replicate data items read or written by T exchange messages to certify T . In addition to ensuring this property, P-Store provides serializability. As the above protocol proposed by Serrano et al., a transaction is first executed in a replica, and then both its read-set and write-set are propagated to other replicas for certification. However, there are two major differences with respect to the previously described protocol: 1) transaction’s metadata are only propagated to the replicas of the partitions it involve using *atomic multicast* primitives, instead of to all nodes using atomic broadcast, and 2) after receiving the transaction, each node only certifies the updates of the transaction to local keys; then, there is a voting phase, during which each replica sends its voting decision to replicas of other involved partitions. A replica can only commit a transaction, if it receives a ‘commit’ vote from at least one replica of each involved partition. Note that having a voting phase increases the communication steps of the certification phase compared with the solution of Serrano et al. [111]. However, P-Store’s certification phase replaces atomic broadcast (which involves *all* nodes in the system) with a more scalable atomic multicast primitive (which only involves the replicas storing data accessed by the transaction).

Finally, Spanner [26] is a DUR database that is the first one known to implement strict serializability in a geo-distributed setting. Spanner employs a classical scheme to certify transactions that combines two-phase commit and Paxos [75]: after the execution of a transaction, the transaction’s read-set and write-set are sent to the partitions it involves. Each partition checks whether the transaction has read-write conflicts with other concurrent transactions; each partition employs Paxos [75] to replicate transactional metadata, which ensures that replicas of a partition agree on the transactional metadata even in the presence of replica failures. Given the leader-based nature of Paxos, one of the partitions’ replicas acts as the leader, whereas the rest of the partitions act as slave replicas: the leader replica

	Replication mode	Consistency guarantee	Certification method
Pedone et al. [97]	Full	Serializability	Atomic broadcast
SI-Rep[83]	Full	SI without RTO	Atomic broadcast
Elnikety et al.[41]	Full	SI with RTO	Centralized server
Serrano et al.[111]	Partial	SI without RTO	Atomic broadcast
P-Store[108]	Partial	Serializability	Atomic multicast + One voting step
Spanner[26]	Partial	Strict Serializability	Two-phase commit + Paxos

Table 2.1 – The taxonomy of discussed DUR systems.

certifies the transaction, replicates the decision to slave replicas and then sends back a vote (commit/abort) to the transaction coordinator; the transaction is committed if the leaders of all involved partitions decide to commit the transaction, and is otherwise aborted. Spanner enforces strict serializability (i.e., serializability with real-time order) as follows. Basically, after determining the outcome of a transaction T , T 's coordinator disseminates the corresponding decision (only to the nodes involved by T) after waiting for a duration equivalent to an estimated maximal clock drift value between all system nodes. This mechanism ensures that any transaction T' started in real time after the commit of T and that could possibly ‘miss’ (i.e., not observe) T 's updates due to a clock skew is made concurrent with T by conflating T . This frees T' from the obligation of observing T 's updates, according to the specification of RTO. Note that Spanner leverages special hardware to provide tight bounds on the clock skew across geo-distributed nodes (as small as 7 milliseconds) [26], thus using the above mentioned mechanism to enforce real-time order is highly efficient.

Summary

Table 2.1 summarizes the DUR systems reviewed so far, classifying them according to the three dimensions we defined at the beginning of this section, namely: replication mode, consistency criterion and certification mechanism.

Overall, one key observation that we draw from the analysis of these solutions is that the systems that support partial replication achieve higher scalability at the cost of a higher complexity of the transaction's certification phase. Specifically, the certification phase of (genuine) partially replicated transactional data stores incur a larger number of communication steps with respect to the case of fully replicated systems. This has detrimental effects on throughput, especially in conflict prone workloads, since transactions undergoing certification typically have to hold pre-commit locks for the whole certification duration. In the context of geo-distributed and/or partially-replicated data stores (where a communication step may have latency up to hundreds of milliseconds), this is likely to cause severe lock convoy effect and hinder performance.

2.3.2 SMR systems

This section is devoted to overviews of state of the art transactional data stores that adopt the SMR approach. We start by describing the general execution model used in these systems, which follows the *order-then-execute* approach. Next, we discuss in more detail several representative instances of SMR systems.

General execution model

1. **Order:** In a nutshell, the goal of the ordering phase is to establish a common transaction serialization, which shall be globally enforced by the various replicas during the execution phase. In the special case of full replication, systems typically employ atomic broadcast [58] or consensus [75] primitives to establish a total order over the whole set of transactions. While these approaches are also amenable for partially-replicated systems, they may incur large overheads, as in a partially-replicated setting not all transactions involve data in all partitions and, thus, they do not need to be delivered to all nodes. Several works have proposed optimized ordering protocols [122, 87] for partially-replicated systems. These protocols aim at maximizing scalability by establishing only a partial order among transactions executing at different partitions and ensure that only the nodes involved in the processing of a transaction participate in its ordering phase. Note that, independently of whether the ordering phase establishes a total or partial ordering over the set of transactions executing at different partitions, the ordering phase establishes a total order over the set of transactions to be executed at each node/partition.
2. **Execute:** In the ordering phase, transactions need to be executed in a deterministic way at all the replicas they involve. This implies, first of all, that the transaction's application-level logic should be deterministic. This requires removing any source of non-determinism, such as generation of random numbers or interaction with local hardware devices (e.g., as physical clocks). This is a necessary assumption to ensure that the independent execution of transactions at different replicas yield the same results.

In addition to ensuring the determinism of transactions, the SMR model needs to ensure that the concurrency control employed at each replica guarantees that transactions are serialized in the same order, i.e., the order established during the ordering phase. Several classical approaches [16, 109] employ a simple single-threaded sequential execution model to ensure such guarantee. This approach, however, prevents any concurrency among the transactions being processed at each node (or partition, in the case of partially replicated data stores) and is, as such, unfit for modern multi-core processors, which represent de facto standard in today's computing architectures.

Therefore, some works strive to improve the concurrency of execution in each partition by different techniques, e.g., using speculation techniques [91, 90], or serializing lock requests of transactions using a single thread and then execute these transactions concurrently with multiple threads [122].

Examples of Representative SMR systems

Next, we overview several representative SMR systems. We classify and discuss these systems focusing on the following dimensions:

1. The *replication model* they target. As already discussed, the replication mode can have a strong impact on the scalability of the transactional platform.
2. Whether the protocol allows *concurrently executing* transactions within a partition, i.e., intra-partition concurrency, which directly affects the maximum throughput achievable during the transaction processing phase in modern multi-core machines.
3. Whether the protocol relies on *additional assumptions on the transactional logic*, which limit the general applicability of the solution. Examples of such assumptions include: a priori knowledge of read-set and write-set of transactions [122] or of the partitions that they involve [16]; only supporting multi-partition transactions which can execute independently (i.e., without exchanging information) at all of the partitions that they involve [85].

Analogously to Section 2.3.1, we first discuss systems that adopt a full replication model and then systems that consider partial replication.

Fully-replicated SMR systems. Santos et al. [107] argue that while the bottleneck of SMR systems is typically the single execution thread, a considerable amount of computation capacity of the execution thread of typical SMR systems are actually spent at dealing with tasks such as ordering, retransmission, snapshotting etc. In order to allow SMR systems to fully exploit the computation capacity of multi-core architecture, they advocate a new approach to implement SMR systems by decoupling most of the mentioned tasks from transaction execution and distributing these tasks to other available cores of a multi-core machine. This approach imposes no assumptions/restrictions on the transaction logic, besides excluding non-deterministic behaviors — which, as mentioned, is a common assumption for all SMR systems. Despite alleviating the load on the execution thread, this technique relies on a single thread for executing transactions. As such, it fails to take full advantage of modern multi-core machines.

P-SMR [85] seeks to enhance the parallelism of SMR by leveraging transaction semantics. P-SMR is a fully-replicated SMR and each replica is organized into multiple execution threads. P-SMR uses a deterministic approach to map transactions to execution threads. For instance, P-SMR can split data into multiple logical partitions and assign each partition to a single thread. Assuming that the system has approximation about the partitions involved by a transaction, transactions can be delivered to the partitions/threads they involve. As such, if a thread receives a transaction that only involves its managed partition, it can execute it independently, i.e., without communicating with other threads. Otherwise, for transactions delivered to multiple threads, all involved threads have to synchronize and then only one of the threads can execute the transaction. As conflicting transactions are guaranteed to be delivered to some common threads, this synchronization mechanism among threads ensures

that conflicting transactions are not concurrently executed. To ensure that transactions involving multiple threads are received by these threads in the same order, P-SMR employs atomic multicast protocol to deliver transactions.

Rex [56] is a fully-replicated SMR system that proposes a novel *execute-agree-follow* framework to achieve parallelism in SMR systems. The execute-agree-follow paradigm works as follows. First, Rex designates a primary replica to execute operations before any secondary replica does so; while executing operations concurrently, the primary replica also records the order of lock acquisition and release. Then, the primary periodically proposes the up-to-date recorded trace as the value for the next Paxos instance. Finally, after replicas have reached consensus on a trace, they follow the execution based on the agreed trace. More specifically, each replica concurrently applies the trace using multiple threads and only synchronize threads in order to respect event orders specified in the trace. For instance, if e in thread t has to wait for the execution of an event e' in thread t' , Rex pauses thread t before executing e and registers with t' to have it signal t after it executes e' . This approach exploits parallelism and also does not impose any requirement to transactions, but the recording and transmission of traces can cause significant computation and network overhead.

Partially-replicated SMR systems. High Performance State Machine Replication [86] (which we refer to as HP-SMR in the following text) and S-SMR [16] are two prominent examples of partially-replicated SMRs. Both systems propose to partition application states to enhance system scalability and rely on atomic multicast primitives to deliver transactions to involved partitions – thus assuming a priori knowledge of the (possibly over-approximated) set of partitions to be accessed by transactions. Each partition employs a single thread to execute transactions in order to ensure determinism. While S-SMR [16] assumes a fully generic transaction logic, HP-SMR [86] assumes transactions to be ‘one-shot’, i.e., for transactions that involve multiple partitions, each of its operations has to be able to be executed in one partition without requiring information from other partitions.

To the best of our knowledge, Calvin [122] is the first partially-replicated SMR system that exploits intra-partition parallelism. Calvin achieves intra-partition parallelism as follows: each partition’s replica employs a single *sequencer* thread that acquires the locks required by transactions prior to their execution in a deterministic fashion; transactions are then executed using multiple execution threads, based on the (deterministic) schedule determined by the sequencer thread. For the sequencer thread to be able to acquire locks for transactions prior to their execution, transaction’s read and write set have to be known in advance. If that is not the case, Calvin has to execute a ‘reconnaissance phase’ prior to transaction execution, which estimates the transaction’s read/write set based on the current state of the system. In case the predicted read/write set turns to be incorrect during actual execution (i.e., as other transactions modified the data store state), the transaction has to be aborted and re-executed.

Eris [77] considers the case of partially replicated transactional data stores, but focuses on the case in which the data store is deployed within the same datacenter. While traditional transaction processing systems are typically network-oblivious, Eris relies on datacenter

	Replication mode	Intra-partition parallelism	Restrictions/assumptions on transactional logic
Santos et al.[107]	Full	No	None
P-SMR[85]	Full	Yes	Estimating involved threads
Rex[56]	Full	Yes	None
HP-SMR[86]	Partial	No	Estimating involved partitions & ‘one-shot’ transactions
S-SMR[16]	Partial	No	Estimating involved partitions
Calvin[122]	Partial	Yes	Knowing transaction’s read-/write set
Eris[77]	Partial	No	Knowing transaction’s read-/write set

Table 2.2 – The taxonomy of discussed SMR systems.

network to efficiently establish a linearizable order of transactions. Basically, in Eris, all packets containing transactions involving multiple partitions are routed to a network-level sequencer, which adds sequence numbers to packets that allow involved partitions of a transaction to consistently order transactions. The sequencer can be implemented in several ways, e.g., a middlebox implemented using a network processor or directly implementing the functionality in a switch. Each partition employs a single thread to execute these transactions – as such, Eris does not provide intra-partition parallelism. One relevant limitation of Eris is that, due to the design of its ordering protocol, the only type of transactions that it can execute efficiently are transactions whose read-set and write-set are known in advance. The transactions that do not meet these assumptions need to undergo additional expensive mechanisms (for instance, a reconnaissance phase similar in spirit to the one employed by Calvin [122]).

2.3.3 Summary

Table 2.2 provides an overview of the discussed SMR systems, classified according to three dimensions, namely: replication mode, intra-partition parallelism and assumptions/restrictions on transactional logic. As can be seen in the table, among the overviewed systems, only Rex [56] achieves intra-partition parallelism, while supporting a fully generic (although deterministic) transactional logic. However, the scalability of Rex is inherently limited by the fact of assuming a fully replicated system model. Also, we can see that all partially-replicated systems require estimating the partitions to be accessed by transactions – this is a fundamental requirement for any partially-replicated SMR systems to have scalability, as otherwise, transactions would always have to be addressed to all partitions. Overall, we can see that enhancing the parallelism of SMR systems is not trivial: state of the art solutions that do achieve this goal either rely on the ability to accurately predict transaction’s read-set and write-set, or are efficient only when processing specific type of transactions (e.g, one-shot transactions).

2.3.4 Comparing DUR and SMR systems

DUR and SMR approaches are both widely adopted approaches to build scalable and fault-tolerant systems. However, due to their different execution models, the two approaches exhibit different trade-offs and achieve best performance in the presence of different types of workloads. In the following text, we discuss several key aspects that differentiate these two approaches.

Restriction on the behaviour of transactions

In SMR systems, the execution of a transaction is replicated at different replicas, after it has been submitted and ordered. Essentially, this means that in SMR systems, 1) the transaction code has to be decided prior to execution, thus interactive transactions, i.e., a transaction where clients can issue new operations after it has been started, can not be supported easily, and 2) transactions have to be deterministic, otherwise even if replicas execute the transaction in the same order, they may end up in different states. Conversely, DUR systems do not suffer from any limitation regarding the supported transactional logic and can be, in this sense, considered amenable for a wider spectrum of applications.

Contention rate

High contention rate can have a significant negative impact on the throughput of both DUR and SMR systems. In both types of systems, in general, during the lifetime of a transaction, locks have to be acquired on some items for a certain period, preventing concurrent access by other transactions². Therefore, when a workload presents high contention, transactions may often be blocked due to lock contention, limiting the achievable parallelism (and, thus, throughput) of the system. Nevertheless, the negative effect of high contention on throughput tends to be more severe for DUR systems than SMR systems [124, 71]. On one hand, recall that in a DUR system, locks have to be held during the whole certification phase, which can take up to hundreds of milliseconds in case of geo-distribution (which is a common type of deployment for today's online applications to achieve disaster fault-tolerance). On the other hand, in a typical deployment of SMR systems, where the whole set of partitions is replicated at each data center, the execution of transactions does not involve any wide-area communication and, thus, is usually very fast. Since the lock duration in SMR systems is much shorter than in DUR systems, a DUR system is likely to experience much longer blocking time than a SMR system under the same contention level, thus achieving lower throughput.

Update ratio

The relative performance of DUR and SMR systems can also be significantly affected by the ratio of update versus read-only transactions. In fact, it is a common optimization for DUR and SMR systems to only execute read-only transactions at one replica. Thus, we

²Of course, this does not apply to single-threaded SMR/PRSM systems. The performance of these systems is not affected by contention rate.

consider read-only transactions impose a similar execution cost in both DUR and SMR systems. However, the cost of executing update transactions can be significantly different in systems adopting these two models. Recall that in a DUR system, update transactions are only executed at one replica and certified at all replicas (and certifying transactions is typically much more lightweight than executing them). Conversely, in SMR systems, update transactions need to be fully executed at all the replicas (of the partition they involve). In this sense, DUR systems have been considered as more scalable than SMR ones in update intensive workloads [66].

2.4 Speculative replicated transactional data stores

After overviewing representative DUR/SMR systems in the above section, in this section we describe DUR and SMR systems that employ speculation to improve system performance. As already discussed in Chapter 1, speculation denotes allowing transactions to observe the updates produced by uncommitted transactions. As we will see, speculation techniques can mitigate the limitations faced by the described non-speculative systems. Note that in the above section, we classify DUR/SMR systems using different dimensions, in order to better understand the intrinsic differences between these systems. In this section, in addition to the above mentioned features, we use several additional parameters relevant to speculation, described as follows.

Speculation mode Speculative results can be exploited by systems in different ways: 1) speculative results can be exposed by other transactions, to avoid other transactions from being blocked, and 2) speculative results can be exposed to external clients, in order to reduce perceived latency and allowing clients to activate optimistic operations. We name the former *internal speculation* and the latter *external speculation*.

Internal speculation is considered ‘safe’ to users, as users can never observe the effects of uncommitted transactions. In contrast, external speculation is more risky, since it can possibly expose misspeculated results (i.e., speculated results that turn out to be incorrect) to clients, it may require developing non-trivial/application-dependent compensating actions [93, 54]. However, it also offers strong advantages performance-wise. In fact, in internal speculation, clients/threads need to block till the final outcome of the transactions they submitted is determined. Conversely, with external speculation, clients/threads can inject new transactions in the system as soon as the system notifies that previously submitted transactions are speculatively (although not yet final) committed, removing the execution of the transaction’s commit phase from the critical path of execution (at least in case speculation is successful). This can lead to significant speed-ups especially in applications in which the workload is injected by a relatively small population of clients/threads (sometimes referred to as closed arrival model [110])

Nevertheless, in fact, depending on how internal/external speculation are implemented, they may expose arbitrary snapshots to the application logic or external clients, which may endanger the application’s correctness or expose clients unexpected results, respectively

(recall our discussion in Section 2.1.4). The next section extends the discussion on the correctness of internal/external speculation.

Transparency As we have discussed, internal/external speculations may lead to exposing arbitrary snapshots to application logic and/or external clients, which is not observable in a non-speculative system. On one hand, the former may endanger the application’s correctness, e.g., leading to divisions by zeros, infinite loops, etc³. On the other hand, the latter may hinder user experience due to exposing users anomalous application state.

We name speculative systems that only expose states compliant with the original consistency criteria as *transparent* speculative systems, or non-transparent otherwise. This correctness criteria is inspired by similar criteria proposed in the domain of transactional memory, namely opacity [53] (see Section 2.1.4). Roughly, opacity can be viewed as an extension of the classical database serializability property with the additional requirement that even transactions that are eventually aborted should still observe a state that could have been produced by *some* sequential history (although not coinciding with the sequential history associated with the history of committed transactions) [53]. While opacity was only defined for serializability, similar concepts can be expanded to other consistency levels, such as snapshot isolation. Transparency is a useful property for speculative systems, as it ensures that applications developed targeting a specific consistency criterion can be executed in systems targeting the same consistency criterion and employing internal and external speculation, without the risk of crashing or exhibiting arbitrary behaviors. This spares programmers from handling complicated concurrency bugs that could otherwise be introduced by speculation techniques.

2.4.1 Speculative DUR systems

As already discussed, the certification phase of non-speculative DUR systems can take hundreds of milliseconds in geo-distributed, partially-replicated setting, which can cause high latency and throttle system through when there is high contention. Several systems strive to mitigate these problems through the use of internal and/or external speculation.

Several systems solely employ the use of external speculation. PLANET [93] is a transactional programming model that aims to help programmers to cope with uncertainty and high latency. The programming model can be layered on top of different transactional data store, and it is equipped with various programming callbacks to facilitate externalizing speculative results to clients. For instance, programmers can specify to externalize speculative results to clients after a specific timeout and/or the calculated commit rate of a transaction is higher than a threshold. Also, programmers are provided with convenient programmatic interfaces to define appropriate compensation logic, which will be automatically triggered in case of misspeculation. Incremental Consistency Guarantees (ICG) [54] exploits the same idea as PLANET of exposing preliminary, possibly incorrect, results with the objective to enhance user-perceived performance. ICG proposes the notion of *Correctables*, which is an interface that allows for exposing multiple, preliminary results

³We illustrate this problem with concrete examples in Chapter 3.3.

of an operation in an incremental way, before its final outcome (which guarantees strong consistency) becomes eventually available. Additionally, as is demonstrated in [54], these preliminary results can often be used by many applications, which do not require strongly-consistent results all the time. In these scenarios, applications can enjoy the low latency of speculative results without the need/risk of compensating misspeculation. Both systems only exploit external speculation, but they rely on conventional data management platforms that do not make use of internal speculation techniques.

Next, we describe systems that employ internal speculation. SPECULA [99] is a fully-replicated transactional memory system that exploits both internal speculation and external speculation. In this system, after a transaction has finished execution, it first undergoes a local certification phase in the local node, which ensures that the transaction has no conflicts with other local transactions. Then, the transaction starts its global certification phase (which relies on total order broadcast primitive [55]) and is speculatively committed, which allows other transactions to speculatively observe its updates. Additionally, the speculative commit event is also externalized to clients, which allows clients to optimistically issue new transactions. SPECULA provides *opacity* consistency guarantee, i.e., transparent speculation for serializability, through the use of several techniques, such as atomic removal of updates of aborted transactions and cascading aborts of transactions that have speculatively read from aborted transactions.

Internal speculation has also been employed in partially-replicated DUR systems [49, 96, 59], though, to the best of our knowledge, none of these systems ensure transparent speculation. In [49, 96, 59], they all employ the classical two-phase commit protocol to certify distributed transactions. To tackle the lock convoy effect, these approaches exploit internal speculation mechanism. Specifically, after a transaction has finished execution in a partition (but before it commits), the transaction's updates are made visible to other transactions. Transactions that have speculatively read from uncommitted transactions can not commit before its depending transaction commits. Nevertheless, this approach can easily expose non-atomic/non-isolated speculative data to transactions – for instance, if a transaction T_1 finishes execution in a partition P_1 but is aborted in partition P_2 , another transaction T_2 may read T_1 's updates only in P_1 but misses T_1 's updates in P_2 , thus observing only parts of the updates produced by T_1 - which would never be possible, had transactions been processed in a non-speculative fashion.

Summary Table 2.3 lists the above described speculative DUR systems and also STR, namely one of the contributions of this dissertation which will be presented in Chapter 3. All these systems employ either internal speculation or external speculation, or both of them, to improve system performance. Though, we can see that among the systems that employ internal speculation, most of them do not provide transparency, except for SPECULA, which however considers the (simpler) case of full replication. In fact, to the best of our knowledge, none of the existing DUR systems that support partial replication employ transparent speculation techniques. This, as will be clearer in Chapter 3, is due to the inherent difficulty to maintain transparency of speculation in partially-replicated, which is also our key motivation to propose STR.

	Replication mode	Consistency guarantee	Certification method	Speculation mode	Transparency
PLANET[93]	Partial	N/A	2PC	External	N/A
ICG[54]	Partial	N/A	2PC	External	N/A
SPECULA[99]	Full	Serializability	Total-order broadcast	Internal & External	Yes
[49]	Partial	Serializability	2PC	Internal	No
[96]	Partial	Serializability	2PC	Internal	No
PROMPT[59]	Partial	Serializability	2PC	Internal	No
STR	Partial	Snapshot Isolation	2PC	Internal	Yes

Table 2.3 – The taxonomy of discussed speculative DUR systems, and STR. We use 2PC to denote two-phase commit

2.4.2 Speculative SMR systems

A large body of work has also investigated the use of speculative techniques in the context of SMR systems, either focusing on reducing the latency of the (consensus-based) ordering layer [70, 76] and/or enhancing the throughput at which transactions can be processed [90, 62]. As the focus of our work is to enhance the throughput of the transaction processing layer, in the following, we focus on overviews solutions that apply speculative techniques to transaction processing. Yet, to understand some of the works to be reviewed next, it is important to first briefly explain how speculation can be applied at the ordering layer. Recall that the SMR approach relies on a preliminary ordering phase, typically implemented via a consensus-based protocol, that ensures that all replicas agree on a common set of transactions and on their serialization order. In many consensus protocols (e.g. [75, 70]), though, it is possible to estimate with high probability what the final order will be, before the ordering layer has completed executing the full protocol. This estimated order is often referred to ‘optimistic’ order. This optimistic order can be used to reduce the effective latency of the ordering layer as follows: a transaction T is first speculatively executed in an order that matches the optimistic order, as soon as this is available. If the optimistic and final order coincide, speculation is successful and T can be committed; else, a misspeculation is detected and T has to be aborted.

Eve [69] proposes a new execution paradigm for SMR systems, namely *execute-then-verify*, that exploits parallelism of SMR systems. Basically, each replica first speculatively executes a batch of delivered transactions concurrently, and then these replicas verify if they have reached the same state; if their state have diverged, all replicas have to rollback their state prior to executing this batch of transactions and re-execute these transactions sequentially. To minimize the chance of state divergence, Eve relies on the use of a ‘mixer’, which applies application-specific information on transactions’ conflict patterns (typically provided by developer) to only execute non-conflicting transactions in parallel. Nevertheless, in case the provided information on transaction’s conflict patterns is inaccurate, conflicting transactions can be executed concurrently and cause replica states to diverge.

In case of such scenario, we argue that transactions may observe non-atomic/non-isolated states, violating transparency.

AGGRO [90] is a fully-replicated SMR system that exploits the idea of speculation in both the ordering phase and the processing phase. As we are more interested in how to leverage speculation to enhance transaction processing, we omit how AGGRO manages misspeculations due to mismatches between the optimistic and final order and assume that these two orders always coincide. In AGGRO, upon the delivery of a transaction T , T is attributed a scalar timestamp that reflects the optimistic order established for the transaction by the consensus-based ordering layer and is immediately executed, possibly concurrently with previously delivered transactions. T reads the latest item versions created by transactions ordered before it according to the optimistic order. T 's updates are firstly buffered in a private buffer, and then made atomically visible to other transactions when T finishes execution. Note that depending on the actual interleaving of concurrent read/write accesses performed by different threads, transactions can be executed in an order that violates the optimistic order. For instance, imagine T_i and T_{i+1} where T_i is ordered before T_{i+1} , and T_i updates key X and T_{i+1} reads X . If T_{i+1} issues a read on X before T_i writes to X , T_{i+1} will miss T_i 's write. As such, T_{i+1} is aborted as soon the conflict with T_i is materialized (i.e., when T_i writes to X). To alleviate this problem, when writing to a key, a transaction T creates a 'work-in-progress' version that blocks its following transactions when they read the data item, until T finishes execution and installs its updates. AGGRO guarantees transparency during the speculative execution of transactions.

Similar to AGGRO, OSARE [91] also applies speculative techniques to both the ordering and processing phases of fully-replicated SMR systems. In AGGRO, if a transaction's optimistic order does not match its final order, the transaction has to be re-executed upon its final order is learnt. OSARE tackles this problem by exploring multiple serialization orders. Specifically, as we have described above, in AGGRO transactions can be executed out-of-order (according to the optimistic order). Imagine a transaction T is executed out-of-order and has missed some preceding transaction's updates. Instead of abandoning T 's updates as in AGGRO, OSARE maintains T 's updates (as well as transactions that have read from T), and activates a new instance of T , which is now guaranteed to include the updates of these transactions. As such, multiple instances of T , executed according to different serialization orders, are maintained, such that if T 's final order mismatches its optimistic order, the system can possibly choose from multiple instances of T and only maintain the one that matches T 's final order. Similar to AGGRO, and for the same considerations, also OSARE employs transparent speculation techniques that guarantee that transactions are always and only allowed to observe a history that is equivalent to a sequential execution of a subset of the transactions eventually included in the final order (and possibly not matching the final order).

The above two protocols were evaluated only via simulation, while Archie [62] explores the similar ideas but has experimentally validated the benefits of its speculative ordering layer and speculative processing layer on an actual prototype. The major difference between Archie and the above speculative protocols lies in the fact that, in Archie, only a single speculative version is maintained for each data item. This is made possible as in

Archie, a transaction can only speculatively commit after all its preceding transactions (i.e., transactions ordered before it according to the speculative order). According to Archie, this design avoids the cost of maintaining and reading from multiple versions. Archie also ensures transparency during speculation.

Summary Finally, we compare all described speculative SMR systems in Table 2.4, where we included also SPARKLE, a speculative SMR protocol that was introduced in this doctoral dissertation and that is presented in Chapter 4. To the best of our knowledge, there is no speculative partially-replicated SMR systems other than SPARKLE, thus we could not include any. We can notice several interesting trends in the table.

First, while all the overviewed systems exploit speculation in the transaction processing phase to enhance parallelism, none of them imposes additional assumptions on the transaction logic - except for Eve, which does assume a priori knowledge on transaction's conflict patterns. SPARKLE requires estimating the partitions to be accessed by transactions, but as we have discussed in Section 2.3.2, this is generally the basic requirement for any partially-replicated SMR systems. This shows that speculation allows SMR systems to have robust performance for wider ranges of applications and spare programmers from dealing with additional requirements for transactions.

Second, all surveyed systems only employ internal speculation, but none of them employs external speculation. In contrast, several speculative DUR systems we overviewed in Section 2.4.1 employ external speculation. In general, we believe that SMR systems rarely exploit external speculation because the execution model of SMR systems do not allow extracting much performance gain by employing external speculation. While we will expand this discussion in Section 5.2.1, intuitively this is because the execution phase of SMR systems usually takes short time, thus employing external speculation (which masks the latency between a transaction's speculative commit and final commit) is not expected to provide large performance gain.

Overall, as can be seen in the table, Sparkle fits a major gap in the literature, namely the absence of speculative SMR protocols that support partial replication. However, note that although AGGRO, OSARE and Archie provide transparent speculation, this is not the case for SPARKLE. This is, in fact, related with the partially-replicated nature of SPARKLE, which makes it harder, and also more performance-wise expensive, to provide transparent speculation (see Section 5.2.2 for detailed discussion). Chapter 6 discusses how to possibly extend SPARKLE to ensure transparency.

	Replication mode	Intra-partition Parallelism	Additional requirements	Speculation mode	Transparency
Eve[69]	Full	Yes	Estimating read/write set	Internal	No
AGGRO[90]	Full	Yes	No	Internal	Yes
OSARE[91]	Full	Yes	No	Internal	Yes
Archie[62]	Full	Yes	No	Internal	Yes
SPARKLE	Partial	Yes	Estimating involved partitions	Internal	No

Table 2.4 – The taxonomy of discussed speculative SMR systems, and SPARKLE. We use Strict SER to denote strict serializability.

Chapter 3

Speculative Transaction Replication

3.1 Introduction

Modern large scale storage systems are increasingly deployed over geographically-scattered data centers [26, 73, 57]. Geo-replication allows storage systems to remain available even in the presence of outages affecting entire data centers and it reduces access latency by bringing data closer to clients. On the down side, though, the performance of geographically distributed data stores is challenged by large communication delays between data centers.

To provide ACID transactions, a desirable feature that can greatly simplify application development [113], some form of global (i.e., inter-data center) certification is needed to safely detect conflicts between concurrent transactions executing at different data centers. The adverse performance impact of global certification is twofold: (i) system throughput can be severely impaired, as transactions need to hold pre-commit locks during their global certification phase, which can cripple the effective concurrency that these systems can achieve; and (ii) client-perceived latency is increased, since global certification lies in the critical path of transaction execution.

Transparent speculation. In this chapter, we investigate the use of speculative processing techniques to alleviate both of the above problems. We focus on geo-distributed partially replicated transactional data stores that provide Snapshot Isolation, a widely employed consistency criterion [36, 41] (SI), and propose a novel distributed concurrency control scheme, Speculative Snapshot Isolation (SPSI), that supports a form of transparent speculative execution called *speculative reads*.

Speculative reads allow transactions to observe the data item versions produced by pre-committed transactions, instead of blocking until they are committed or aborted. As such, speculative reads can reduce the “effective duration” of pre-commit locks (i.e., as perceived by conflicting transactions), thus reducing transaction execution time and enhancing the maximum degree of parallelism achievable by the system — and, ultimately, throughput. We say that speculative reads are a *transparent speculation* technique, as misspeculations caused by it never surface to the clients and can be dealt with by simply re-executing the

affected transaction.

Avoiding the pitfalls of speculation. Past work has demonstrated how the use of speculation, either transparently or requiring source-code modification [49, 93, 67, 54, 119], can significantly enhance the performance of distributed [119, 99, 67, 93, 96] and single-site [49] transactional systems. However, these approaches suffer from several limitations:

1. Unfit for geo-distribution/partial replication. Some existing works in this area [99, 67, 119] were not designed for partially replicated geo-replicated data stores. On the contrary, they target different data models (i.e., full replication [99, 119]) or rely on techniques that impose prohibitive costs in WAN environments, such as the use of centralized sequencers to totally order transactions [67].

2. Subtle concurrency anomalies. Existing partially replicated geo-distributed transactional data stores that allow speculative reads [49, 59, 96] expose applications to anomalies that do not arise in non-speculative systems and that can severely undermine application correctness. Figure 3.1 illustrates two examples of concurrency anomalies that may arise with these systems. The root cause of the problem is that existing systems allow speculative reads to observe *any* pre-committed data version. This exposes applications to data snapshots that reflect only partially the updates of transactions (Fig. 3.1a) and/or include versions created by conflicting concurrent transactions (Fig. 3.1b). These anomalies can have a twofold harmful impact. First, they can lead to externalizing non-atomic/non-isolated snapshots to clients. Second, if the application runs in non-sandboxed environments, observing anomalous states may affect transaction execution to the extent to generate anomalous/unexpected behaviours [43]. This is the case, for instance, of JAVA programs that use the JPA interface [125] to access transparently the data maintained by a back-end data store. In this case, applications can exploit in arbitrary ways the data they fetch from the data store. As such, observing inconsistent states could lead applications to, e.g., crash or hang in infinite loops, as exemplified in Fig. 3.1.

3. Performance robustness. If used injudiciously, speculation can hamper performance. As we will show, in adverse scenarios (e.g., large likelihood of transaction aborts and high system load) misspeculations can significantly penalize both user-perceived latency and system throughput.

Contributions. Overall, this chapter introduces the following novel contributions:

- Speculative Snapshot Isolation (SPSI), a novel consistency model that is the foundation of STR (Section 3.3). Besides guaranteeing the familiar Snapshot Isolation (SI)¹ to *committed transactions*, SPSI provides clear and rigorous guarantees on the atomicity and isolation of the snapshots observed and produced by *executing transactions*. In a nutshell, SPSI requires an executing transaction to read data item versions committed before it started (as in SI), but it also allows to atomically observe the effects of non-conflicting transactions that originated on the same node and pre-committed before the transaction started. This shelters programmers from having to

¹The original definition of SI enforces real-time ordering guarantee as we described in Section 2.1.2.

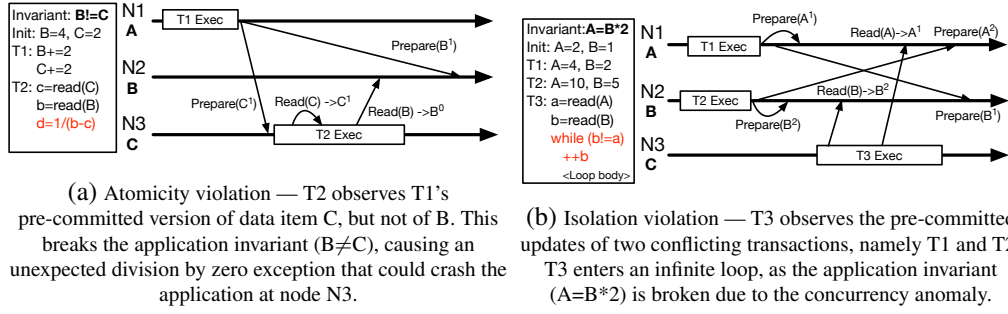


Figure 3.1 – Examples illustrating possible concurrency anomalies caused by speculative reads. T1, T2 and T3 are transactions; N1, N2 and N3 are three nodes that store data items A, B and C, respectively.

reason about complex concurrency anomalies that can otherwise arise in speculative systems.

- Speculative Transaction Replication (STR), a novel speculative transactional protocol for partially replicated geo-distributed data stores (Section 3.4). STR shares several key design choices with state-of-the-art strongly consistent data stores [26, 36, 100], which contribute to its efficiency and scalability. These include: multi-versioning, which maximizes efficiency in read-dominated workloads [15], purely decentralized concurrency control based on loosely synchronized physical clocks [26, 36, 101], and support for partial replication [72, 26]. The key contribution of STR is its innovative, fully decentralized, concurrency control mechanism, which aims not only to ensure (SPSI)-safe speculations in a lightweight and scalable fashion, but also to enhance the chances of successful speculation via a novel transaction timestamping mechanism that we called *Precise Clocks*. We also discuss possible extensions to enable STR to provide Serializability and guarantee real-time ordering.
- A lightweight yet effective self-tuning mechanism, based on a feedback control loop, that dynamically enables or disables speculation based on the workload characteristics (Section 3.4.5).
- We evaluate STR on up to nine geo-distributed Amazon EC2 data centers, using both synthetic and realistic benchmarks (TPC-C and RUBiS) (Section 3.8). Our experimental study shows that the use of transparent speculation (speculative reads) yields up to $11\times$ throughput improvements and $10\times$ latency reduction compared to a state of the art protocol, i.e., Clock-SI [36], that ensures SI without resorting to speculative techniques. This is achieved in a fully transparent way, i.e., requiring no compensation logic.

Structure of the chapter. The structure of this chapter is as follows:

- Section 3.2 describes the assumed system and data model.
- Section 3.3 introduces the SPSI consistency model.
- The STR protocol is presented in Section 3.4.
- Section 3.5 proves the correctness and safety of the STR protocol.
- Section 3.6 discusses how to extend the protocol to ensure real-time ordering and serializability.
- Section 3.7 provides a detailed discussion of the relations of SPSI and STR with respect to prior works, extending and deepening the review on state of the art performed in Chapter 2.
- Finally, Section 3.8 presents the evaluation results of the STR protocol.

3.2 System and data model

The target system model consists of a set of geo-distributed data centers, each hosting a set of nodes. In the following, we assume a key-value data model. This is done for simplicity and since our current implementation of STR runs on a key-value store. However, the protocol we present is agnostic to the underlying data model (e.g., relational or object-oriented).

Data and replication model. The dataset is split into multiple partitions, each of which is responsible for a disjoint key range and maintains multiple timestamped versions for each key. Partitions may be scattered across the nodes in the system using arbitrary data placement policies. Each node may host multiple partitions, but no node or data center is required to host all partitions.

A partition can be replicated within a data center and across data centers. STR employs synchronous master-slave replication to enforce fault tolerance and transparent fail over, as used, e.g., in [26, 9]. A partition has a master replica and several slave replicas. As will be clearer in Section 3.4, both the master replica and slave replicas (that are up-to-date) can serve read requests to their replicated data items. We say that a key/partition is remote for a node, if the node does not replicate that key/partition.

Synchrony assumptions. We assume a classic asynchronous distributed system model, with no assumptions on upper bounds on relative process speeds and communication latency. STR requires that nodes are equipped with loosely synchronized, conventional hardware clocks, which we only assume to monotonically move forward. Note that while the correctness of the presented protocol is not affected by the clock skew (i.e., either the clock skew is small or large), it is known [36, 26] that the performance of protocols based on physical clocks, like Clock-SI or Spanner, can be negatively affected in case of large clock skews (e.g., by introducing large delays or suffering from large abort rates). The same consideration applies to the protocol proposed here, which extends this class of protocols (and in particular Clock-SI) with speculative transaction processing techniques.

Additional synchrony assumptions are required to ensure the correctness of the synchronous master-slave replication scheme used by STR in presence of failures [44]. STR integrates a classic single-master replication protocol, which assumes perfect failure detection capabilities [21]. We note, though, that it would be possible to replace the replication scheme currently employed in STR to use techniques, like Paxos [40], which require weaker synchrony assumptions. Further, as we shall discuss in Section 3.4.6, the mechanisms employed by STR to ensure that SPSI semantics are guaranteed also in presence of failures hinge on the assumption that it is possible to establish a finite upper bound on the clock skew between nodes. The same additional synchrony assumption is necessary in order to allow the implementation of the variant of STR that ensures real-time ordering, which is presented in Section 3.6.1.

Transaction execution model. A transaction is executed, by a process called its *coordinator*, in the node where it was originated. When the transaction starts, it is attributed a snapshot time, which we call ST , according to the physical clock of its originating node.

When the transaction requests to commit, it undergoes a local certification phase, which checks for conflicts with concurrent transactions on all locally replicated partitions. If the certification phase succeeds at a partition, the transaction *pre-commits* at that partition obtaining a pre-commit timestamp, note PC (in the following, we use the terms pre-committed and prepared as synonyms). If the transaction passes the certification phase for all the local partitions that it involves, the transaction *local commits* (at the node in which the transaction originated), or *speculative commits*, and it is attributed a local commit timestamp, noted LC . Next, it executes a global certification phase that detects conflicts with transactions originated at any other node in the system and aims at pre-committing the transaction on every master replica of the partitions that the transaction updated.

We define the snapshot observed by a transaction T , or simply snapshot of transaction T , as the set of (local or final committed) versions that were read by T . We say that a transaction T^1 is included in the snapshot of a transaction T^N , if T^N reads a version created by T^1 either directly, or indirectly, i.e., there exists a sequence of transactions $T^1, \dots, T^i, \dots, T^N$ such that T^{i+1} reads a version created by T^i .

Note that, since our transaction execution model defines an additional (local) commit event, and since we intend to allow transactions to observe the effects of local committed transactions that originated at the same node, we need to revise the definition of concurrent transactions. Intuitively, two transactions $T1, T2$ are concurrent if their execution intervals overlap, i.e., if the execution interval of $T1$ ends after the execution interval of $T2$ starts, or vice versa. We always identify the start of a transaction's execution interval using its ST . The end of the execution interval of $T1$ and $T2$, though, is determined in different ways, depending on whether these are originated at the same node, or not. If $T1$ and $T2$ originate at different nodes, we identify the end of their execution interval using their FC . If $T1$ and $T2$ originate at the same node, we define the end of the execution interval of $T \in \{T1, T2\}$ as its LC , provided that T was only local committed but not final committed, or as its FC , if T was final committed.

Transactions that pass the global certification phase are said to *final commit* and are

attributed a final commit timestamp, noted FC . Commit requests are confirmed to applications only if the transaction is final committed, which guarantees that speculative states never surface to clients. However, the versions created by a local committed transaction T can be exposed to other transactions via the *speculative read* mechanism. We say that these transactions *depend* on T .

3.3 The SPSI consistency model

We introduce Speculative Snapshot Isolation (SPSI), a consistency model that generalizes the well-known SI criterion to define a set of guarantees that shelter applications from the subtle anomalies (Fig. 3.1) that may arise when using speculative techniques. Note that while the original definition of SI enforces real-time ordering guarantee, SPSI is defined upon a variant of SI [41], namely Generalized SI (GSI), which provides no real-time order guarantee. Thus, in the remainder of the paper, with the term SI we always refer to GSI, unless otherwise specified.

Before presenting the SPSI specification, we first recall the definition of SI [41], which we adapted to match the terminology adopted in our transaction execution model and to simplify the original notation:

- SI-1. (*Totally-ordered committed versions*) For each data item, its committed versions are totally ordered.
- SI-2. (*Transactions read consistent snapshots*): All read operations issued by a transaction with snapshot timestamp ST observe the most recent version committed by transactions with commit timestamp $FC \leq ST$.
- SI-3. (*No Write-Write Conflicts*) The write-sets of any committed concurrent transactions must be disjoint.

We now introduce the SPSI specification:

- SPSI-1 (*Totally-ordered final committed versions*) For each data item, its final committed versions are totally ordered.
- SPSI-2 (*Per-node totally-ordered final and speculatively committed versions*) At each node, the final and speculative committed versions of each data item are totally ordered.
- SPSI-3. (*Transactions read consistent speculative snapshots*) A transaction T originated at a node N with snapshot time ST must observe the most recent versions created by transactions that i) final commit with timestamp $FC \leq ST$ (independently of the node where these transactions originated), and ii) local commit with timestamp $LC \leq ST$ and originated at node N .
- SPSI-4. (*No Write-Write Conflicts among Final Committed Transactions*) The write-sets of any final committed concurrent transactions must be disjoint.

- SPSI-5. (*No Write-Write Conflicts among Transactions in a Speculative Snapshot*) Let S be the snapshot of a transaction. The write-sets of any concurrent transactions included in S must be disjoint.
- SPSI-6. (*No Dependencies from Uncommitted Transactions*) A transaction can only be final committed if it does not depend on any non-final committed transaction.

SPSI-1 and SPSI-2 jointly define the ordering of (final and speculatively) committed data item versions. Note that while SPSI-1 applies to all possible nodes, SPSI-2 only requires the total ordering of item versions within a node – speculatively committed versions in different nodes are not required to be totally ordered.

SPSI-3 extends the notion of snapshot, at the basis of the SI definition, to provide the illusion that transactions execute on immutable snapshots, which reflects the execution of all the transactions that local committed before T 's activation and that originated on the same node. By demanding that the snapshots over which transactions execute reflect *only* the effects of locally activated transactions, SPSI allows for efficient implementations, like STR's, which can decide whether it is safe to observe the effects of a local committed transaction based solely on local information.

Note that property SPSI-3 is specified for *any* transaction, including the ones that eventually abort (e.g., because some other SPSI property is violated). Hence, SPSI-3 must hold throughout the execution of transactions. This has also another relevant implication: assume that a transaction T , which started at time t , reads speculatively from a local committed transaction T' with timestamp $LC \leq t$, and that, later on, T' final commits with timestamp $FC > t$; at this point T violates the first sub-property of SPSI-3. Hence, T must be aborted before T' is allowed to final commit. The same applies in case T' aborts: since SPSI-6 prohibits developing dependencies from aborted transactions, also in this case, T must be aborted before T' is.

SPSI-4 coincides with SI-3, ensuring the absence of write-write conflicts among concurrent final committed transactions. SPSI-5 complements SI-2 by ensuring that the effects of conflicting transactions can never be observed. Finally, SPSI-6 ensures that a transaction can be final committed only if it does not depend on transactions that may eventually abort.

Note that the SI definitions presented above **does not** ensure real-time ordering guarantee; as our SPSI definitions are based on them, the SPSI definitions do not provide real-time ordering either. We will discuss in Section 3.6.1 on how to circumvent this problem.

Which anomalies does SPSI allow? SPSI provides identical guarantees to SI for final committed transactions. As for local committed and active transactions, SPSI allows for histories that would be rejected by SI, e.g., observing a version locally committed by a transaction that eventually aborts due to a conflict with some remote transaction. However, we argue that these anomalies allowed by SPSI are unharmed for applications designed to operate using SI. This is easy to show if one considers that SPSI ensures that any transaction T behaves like if it had executed under SI in a history that includes only the transactions known by the node in which T originated at the time in which T was activated. More formally, the snapshot observable by T in any SPSI-compliant history $\mathcal{H}$ is equivalent to

the one that T would observe in some SI-compliant history $\mathcal{H}'$, which differs from $\mathcal{H}$ only because $\mathcal{H}'$ may omit some remote transaction concurrent with T . In other words, any snapshot observable with SPSI can be obtained via a (possible) history that would be legal using SI. Clearly, if an application works correctly with SI, i.e., it is correct with any SI-compliant history (including history $\mathcal{H}'$), the application will be also be correct when faced with history $\mathcal{H}'$ — and, thus, when executing the SPSI-compliant history $\mathcal{H}$.

Which anomalies does SPSI prevent? In Fig. 3.1 we have already exemplified some of the concurrency anomalies that SPSI prevents, and which could lead applications to hang or crash. Interestingly, while analyzing the TPC-C and RUBiS benchmarks, we have identified several concurrency bugs that may arise and cause application crashes, if SPSI's guarantees are not enforced.

```

// New-Order
...
Order order;
storage->Put(order);
for (int i = 0; i < order.ol_count; i++) {
    OrderLine order_line = create_ol(order, i);
    storage->Put(order_line);
    ...
}

// Order-Status
...
Order order = storage->Read(customer.last_order);
for (int i = 0; i < order.ol_count; i++) {
    OrderLine ol = storage->Read(order.ol, i)
    // Parse throws a Null Pointer Exception if ol is null
    parse(ol);
    ...
}

```

Listing 3.1 – Potential anomaly prevented by SPSI in TPC-C.

Listing 1 illustrates one of the anomalies we spotted in TPC-C benchmark, which involves the New Order (NO) and Order Status (OS) transactions. NO inserts a new order for a customer and then creates some number of corresponding order lines. OS fetches the identifies of the last order of a given customer, and the retrieves the corresponding order lines. In a partially-replicated setting, the order record may be stored in the node where the NO transaction was activated, but the order lines may be stored in some different node. An injudicious use of speculative reads may allow a OS transaction to read the pre-committed order record of a concurrent NO, but then allow the OS to miss the corresponding order lines (an atomicity violation that is prevented by SPSI-1). In this case, the parse method in OS would be fed with a null pointer and generate an unexpected exception, which would never occur with SI (or SPSI) and could lead to a crash of the application.

3.4 The STR protocol

This section introduces the Speculative Transaction Replication (STR) protocol. For reasons of clarity, we present the design of STR incrementally. We first present a non-speculative base protocol that implements a SI-compliant transaction system. This base

protocol is then extended with a set of mechanisms aimed to support speculation in an efficient and SPSI-compliant way. Finally, we discuss the fault tolerance of STR.

3.4.1 Base non-speculative protocol

The base protocol is a multi-versioned, SI-compliant algorithm that relies on a fully decentralized concurrency control scheme similar to that employed by recent, highly scalable systems, like Spanner or Clock-SI [36, 26]. In the following, we describe the main phases of STR’s base protocol.

Execution. When a transaction is activated, it is attributed a *snapshot timestamp*, noted as ST , equal to the physical time of the node in which it was originated. The read snapshot determines which data item versions are visible to the transaction. Upon a read, a transaction T observes the most recent version v having final commit timestamp $v.FC \leq T.ST$. However, if there exists a pre-committed version v' with a timestamp smaller than $T.ST$, then T must wait until the pre-committed version is committed/aborted. In fact, as will become clear shortly, the pre-committed version may eventually commit with a timestamp $FC \leq ST$ — in which case T should include it in snapshot — or $FC > ST$ — in which case it should not be visible to T .

Note that read requests can be sent to any replica that maintains the requested data item. Also, if a node receives a read request with a read snapshot ST higher than its current physical time, the node delays serving the request until its physical clock catches up with ST . Instead, writes are always processed locally and are maintained in a transaction’s private buffer during the execution phase.

Certification. Read-only transactions can be immediately committed after they complete execution. Update transactions, instead, first check for write-write conflicts with *concurrent local* transactions. If T passes this local certification stage, it activates a, 2PC-based, *global certification phase* by sending a pre-commit request to the master replicas of any key it updated and for which the local node is not a master replica. If a master replica detects no conflict, it acquires pre-commit locks, and proposes its current physical time for the pre-commit timestamp.

Replication. If a master replica successfully pre-commits a transaction, it synchronously replicates the pre-commit request to its slave replicas. These, in their turn, send to the coordinator their physical time as proposed pre-commit timestamps.

Commit. After receiving replies from all the replicas of updated partitions, the coordinator calculates the commit timestamp as the maximum of the received pre-commit timestamps. Then it sends a commit message to all the replicas of updated partitions and replies to the client. Upon receiving a commit message, replicas mark the version as committed and release the pre-commit locks.

This protocol has a potential for high scalability. Unfortunately, though, in geo-distributed settings, its throughput can be severely limited by convoy effects caused by the pre-commit locks. These locks are held throughout the transactions’ certification phase,

which in geo-distributed data stores entail the latency of at least one inter-data center RTT — or more if data partitions are replicated in different data-centers to allow for disaster recovery. Throughout this period, concurrent transactions attempting to read pre-committed data are conservatively blocked to ensure that if a version of that data item is included in the snapshot of a transaction T , then that version is guaranteed to have been committed (i.e., the transaction that created that version will not eventually abort) with a final timestamp smaller than T 's snapshot timestamp (i.e., the transaction that created that version will not eventually commit obtaining a final timestamp larger than T 's snapshot timestamp).. This inherently limits the maximum degree of concurrency (and hence throughput) achievable by the system.

As we mentioned, the idea at the basis of STR is to tackle this problem by allowing transactions to observe pre-committed versions. Materializing this idea to build STR raised several technical challenges: guaranteeing (SPSI-)safe speculations (Section 3.4.2), maximizing the likelihood of successful speculation (Section 3.4.3) and ensuring robust performance even in adverse workload settings (Section 3.4.5) .

3.4.2 Enabling SPSI-safe speculations

Let us discuss how to extend the base protocol described above to incorporate speculative reads, while preserving SPSI semantics. The example executions in Fig. 3.1 illustrate two possible anomalies that could lead transactions to observe non-atomic snapshots, which violate property SPSI-1 (Fig. 3.1a), or snapshots reflecting the execution of two conflicting transactions, which violate property SPSI-3 (Fig. 3.1b).

STR tackles these issues as follows. First, it restricts the use of speculative reads, as mandated by SPSI-1, by allowing to observe only pre-committed versions created by local transactions. This is sufficient to prevent the anomalous executions illustrated in Fig. 3.1a and Fig. 3.1b, since in both executions the transaction executing at node N_3 (i.e., the transaction that in both executions experiences an anomaly) will be prevented from reading speculatively pre-committed data.

However, the above mechanisms still do not suffice to ensure, in all possible scenarios, properties SPSI-3 and SPSI-5. There are, in fact, two other subtle cases that have to be taken into account, both involving speculative reads of versions created by local committed transactions that updated some remote key.

The first scenario, illustrated in Fig. 3.2, is associated with the possibility of including in the same snapshot a local committed transaction, $T1$ — which will eventually abort due to a remote conflict, say with $T2$ — and a remote, final committed transaction, $T3$, that has read from $T2$. In fact, the totally decentralized nature of STR's concurrency protocol, in which no node has global knowledge of all the transactions committed in the system, makes it challenging to detect scenarios like the ones illustrated in Fig. 3.2 and to distinguish them, in an exact way, from executions that did not include transaction $T2$ — in which case the inclusion of $T1$ and $T3$ in $T4$ would have been safe.

The mechanism that STR employs to tackle this issue is based on the observation that such scenarios can arise only in case a transaction, like $T4$, attempts to read speculatively

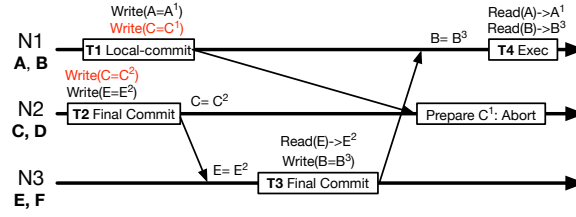


Figure 3.2 – History exemplifying indirect conflicts between a local committed transaction, $T1$, and a final committed transaction originated at a different node, $T3$. If $T4$ included both $T1$ and $T3$ in its snapshot, it would violate SPSI property 3.

from a local committed transaction, like $T1$, which has updated some remote key. The latter type of transactions, which we call “unsafe” transactions, may have in fact developed a remote conflict with some *concurrent* final committed transaction (which may only be detected during their global certification phase), breaking property SPSI-3. In order to detect these scenarios, STR maintains two additional data structures per transaction: *OLC* (Oldest Local Commit) and *FFC* (Freshest Final Commit), which track, respectively, the read snapshot of the oldest “unsafe” local committed transaction and the commit timestamp of the most recent remote final committed transaction, which the current transaction has read from (either directly or indirectly). Thus, STR blocks transactions when they attempt to read versions that would cause *FFC* to become larger than *OLC*. This mechanism prevents including in the same snapshot unsafe local committed transactions along with remote final committed transactions that are concurrent and may conflict with them. For example, in Fig. 3.2, STR blocks $T4$ when attempting to read B from $T3$, until the outcome of $T1$ is determined (not shown in the figure).

The second scenario arises in case a transaction T attempts to speculatively read a data item d that was updated by a local committed transaction T' , where d is not replicated locally. In this case, if T attempted to remotely read d , it may risk to miss the version of d created by T' , which would violate SPSI-1. To cope with this scenario, whenever an unsafe transaction local commits, it temporarily (until it final commits or aborts) stores the remote keys it updated in a special *cache partition*, tagging them with the same local commit timestamp. This grants prompt and atomic (i.e., all or nothing) access to these keys to any local transaction that may attempt to speculatively read them.

3.4.3 Promoting successful speculation via Precise Clocks

Recall that, SPSI-1 requires that if a transaction T reads speculatively from a local committed transaction T' , and T' eventually final commits with a commit timestamp that is larger than the read snapshot of T , then T has to be aborted. Thus, in order to increase the chance of success of speculative reads, it is important that the commit timestamps attributed to final committed transactions are “as small as possible”.

To this end, STR proposes a new timestamping mechanism, i.e., *Precise Clocks*, which is based on the following observation. The smallest final commit timestamp, FC , attributable to a transaction T that has read snapshot ST must ensure the following prop-

erties:

- **P1.** $T.FC > T.ST$, which guarantees that if T reads a data item version with timestamp ST and updates it, the versions it generates has larger timestamp than the one it read.
- **P2.** $T.FC$ is larger than the read snapshot of all the transactions $T_1, \dots, T_n$, which (a) read, before T final committed, any of the keys updated by T , and (b) did not see the versions created by T , i.e., $T.FC > \max\{T_1.ST, \dots, T_n.ST\}$. This condition is necessary to ensure that T is serialized after the transactions $T_1, \dots, T_n$, or, in other words, to track write-after-read dependencies among transactions correctly.

Ensuring property P1 is straightforward: instead of proposing the value of the physical clock at its local node as pre-commit timestamp, the transaction coordinator proposes $T.ST + 1$. In order to ensure property P2, STR associates to each data item an additional timestamp, called *LastReader*, which tracks the read snapshot of the most recent transaction that has read that data item. Hence, in order to ensure property P2, the nodes involved in the global certification phase of transaction T propose, as pre-commit timestamp, the maximum among the *LastReader* timestamps of any key updated by T on that node.

It can be easily seen that the Precise Clocks mechanism allows to track write-after-read dependencies among transaction at a finer granularity than the timestamping mechanism used in the base protocol — which, we recall, is also the mechanism used by non-speculative protocols like, e.g., Spanner [26] or Clock-SI [36]. Indeed, as we will show in Section 3.8, the reduction of commit timestamps achievable via Precise Clocks does not only increase the chances of successful speculation, but also reduces abort rate for non-speculative protocols.

It should be noted that the Precise Clocks mechanism does bring overheads, in terms of both additional storage requirements as well as additional computational costs, which we discuss next.

As will be shown in Section 3.8.1, the Precise Clocks mechanism introduces a storage overhead (for storing the *LastReader* metadata at the key granularity) that can be quantified as approximately 10% with realistic workloads (i.e., TPC-C and RUBiS). We argue that one could further lower the spatial overhead, at the cost of increasing abort rate, by tracking the *LastReader* information at a coarser granularity, e.g., associating a single *LastReader* with a set of keys or even an entire partition.

For what concerns computational overheads, the Precise Clocks mechanism requires updating the *LastReader* variable upon each read operation, whereas typical SI protocols avoid the cost of tracking reads. However, STR targets a geo-distributed system setting in which throughput is normally limited by the global certification phase, which is affected by large communication delays between data centers. Keeping into account that these delays are often in the order of hundreds of milliseconds, we argue that the additional cost imposed by the *local* update of the *LastReader* variables is likely to be negligible. In fact, as we will experimentally show in Section 3.8, the overhead of the Precise Clocks mechanism is, in practice, largely outweighed by the benefits it provides.

Algorithm 1: Coordinator protocol

```
1 function transaction startTx()
2   Tx.ST ← current_time()
3   Tx.Coord ← self()
4   Tx.OLCSet ← { < ⊥, ∞ > }
5   Tx.FFC ← 0
6   return Tx

7 function value read(transaction Tx, key Key)
8   if Key is locally replicated or in cache then
9     <Value, Tw> ← local_partition(Key).readFrom(Tx, Key)
10    Tx.OLCSet.put(Tw, min_value(Tw.OLCSet))
11  else
12    send { read, Tx, Key } to any p ∈ Key.partitions()
13    wait until receiving <Value, Tw>
14    Tx.FFC ← max(Tx.FFC, Tw.FFC)
15    wait until min_value(Tx.OLCSet) ≥ Tx.FFC
16    return Value

17 function result commitTx(transaction Tx)
18   // Local certification
19   LCTime ← Tx.ST + 1
20   for <P, Key, Value> ∈ Tx.WriteSet
21     if local_replica(P).prepare(Key, Value, LCTime) = <prepared, TS>
22       LCTime ← max(LCTime, TS)
23   else
24     abort(Tx)
25   if Tx updates non-local keys
26     Tx.OLCSet.put(self(), Tx.ST)
27   for <P, Key, Value> ∈ Tx.WriteSet
28     local_replica(P).localCommit(Tx, LCTime, Key, Value)
29   // Global certification
30   for <P, Key, Value> ∈ Tx.WriteSet s.t. Key's master partition is not local
31     send [prepare, Tx, Key, Value, LCTime] to Key's master partition
32   wait until receiving [prepared, TS] from Tx.InvolvedReplicas
33   wait until all dependencies are resolved
34   CommitTime ← max(all received TS, LCTime)
35   commit(Tx, CommitTime)
36   return committed
37   wait until receiving aborted
38   abort(Tx)
39   return aborted

40 function void commit(transaction Tx, timestamp CT)
41   Tx.FFC ← CT
42   Tx.OLCSet ← { < ⊥, ∞ > }
43   for each Tr s.t. ∃ dependence from Tr to Tx
44     if Tr.ST ≥ CT then
45       remove Tx from Tr's read dependency
46       Tr.OLCSet.remove(Tx)
47       Tr.FFC ← max(Tr.FFC, CT)
48     else
49       abort(Tr)
50   atomically commit Tx's local committed updates and remove cached updates
51   send [commit, Tx, CT] to every remote partition updated by Tx

52 function void abort(transaction Tx)
53   abort transactions with dependencies from Tx
54   atomically remove Tx's local committed updates
55   send [abort, Tx, ] to every remote partition updated by Tx
```

3.4.4 Pseudo-code of the STR Protocol

The pseudocode of the STR protocol is reported in Algorithms 1 and 2, which describe, respectively, the behavior of transaction coordinators and of data partitions.

Start transaction. Upon activation, a transaction is assigned a snapshot timestamp (ST) equal to the current value of the node's physical clock. Its FFC is set to 0 and its OLCSet, i.e., the set storing the identifiers and snapshot times of the unsafe transactions from which the transaction reads from, to $\{< \perp, \infty >\}$ (Alg. 1, 1-6).

Speculative read. Read requests to locally-replicated keys are served by local partitions. A read request to a non-local key is first served at the cache partition to check for updates from previous local committed transactions. If no appropriate version is found, the request is sent to any (remote) replica of the partition that contains this key (Alg. 1, 8-13). Upon a read request for a key, a partition updates the *LastReader* of the key and fetches the latest version of the key with a timestamp no larger than the reader's read snapshot (Alg. 2, 18). If the fetched version is final committed, or it is local committed and the reader is reading locally, then the partition returns the value and id of the transaction that created the value; otherwise, the reader is blocked until the transaction's final outcome is known (Alg. 2, 15-26). The reader transaction updates its OLCSet and FFC, and only reads the value if the minimum value in its OLCSet is greater than or equal than its FFC. If not, the transaction waits until the minimum value in its OLCSet becomes larger than its FFC (Alg. 1, 14-17). This condition may *never become true* if the transaction that created the fetched value conflicts with transactions already contained in the reader's snapshot. In that case, the reader will be aborted after this conflict is detected and stop waiting.

Local certification. After a transaction T finishes execution, its write-set is locally certified. The local certification is essentially a local 2PC across all local partitions that contain keys in the transaction's write-set, including the cache partition if the transaction updated non-local keys (Alg. 1, 18-24). For each updated key, a partition detects if T has write-write conflicts with other transactions that have updated the same key, which we refer to as T' , as follows.

If the latest version of a conflicting key updated by T' has timestamp larger than $T.ST$, T is aborted. This is true independently of whether the conflicting key is in the pre-commit, local commit or final commit state.

Let us now discuss the case in which the most recent existing version has a timestamp smaller than or equal to $T.ST$. There are two cases to consider. If the most recent version is final committed or local committed, then a new version of the key is immediately added to the data store and its state is set to pre-committed. Else, i.e., the most recent version is pre-committed, T is forced to wait till T' is either committed, aborted, or local committed.

Note that if the most recent version is in *local committed* state, it follows that when T' final commits, the *FC* timestamp of this version might become larger than $T.ST$ and cause the abort of T (Alg. 1, 49). In order to cope with this case, T is added to the set of transactions (Alg. 2, 37-38) whose outcome depends on the FC of T' .

If the certification check passes, the partition proposes a prepare timestamp according to the Precise Clocks rule (Alg. 2, 27-42). Upon receiving replies from all updated local partitions (including the cache partition), the coordinator calculates the local commit timestamp as the maximum between the received prepare timestamps and the transaction’s read snapshot plus one. Then, it notifies all the updated local partitions. A notified partition converts the pre-committed record to local committed state with the local commit timestamp (Alg. 1, 27-28 and Alg. 2, 43-45). If the transaction updates non-local keys, the transaction is an ‘unsafe’ transaction, so it adds its snapshot time to its *OLCSet* (Alg. 1, 25-26).

Global certification and replication. After local certification, the keys in the transaction’s write-set that have a remote master are sent to their corresponding master partitions for certification (Alg. 1, 30-31). Same as the local certification phase, master partitions check for conflicts, propose a prepare timestamp and pre-commit the transaction (Alg. 2, 27-38). Unlike the case in which T is a local transaction, though, if T is remote, its pre-commit request will be blocked in case of conflict with a local committed transaction T' and T is not going to be local committed. We recall, in fact, that STR only allows locally originated transactions to expose their pre-committed states to other local transactions.

Next, a master partition replicates the prepare request to its slave replicas and replies to the coordinator (Alg. 2, 40-42). After receiving a replicated prepare request, the slave partition aborts any conflicting local committed transactions and stores the prepare records. As slave replicas can be directly read bypassing their master replica, slave replicas also track the *LastReader* for keys; so, each slave also proposes a prepare timestamp for the transaction to the coordinator (Alg. 2, 8-10).

Final commit/abort. A transaction coordinator can final commit a transaction, if (i) it has received prepare replies from all replicas of updated partitions, and (ii) all dependencies are resolved. The commit decision, along with the commit timestamp, is sent to all non-local replicas of updated partitions. T ’s *FFC* is updated to its own commit timestamp, and its *OLCSet* is set to infinity (Alg. 1, 40-51). Upon abort, the coordinator removes any local committed updated version, triggers the abort of any dependent transaction and sends the decision to remote replicas (Alg. 1, 52-55).

3.4.5 Dynamically tuning speculation

Speculative reads are based on the optimistic assumption that local committed transactions are unlikely to experience contention with remote transactions. Although our experiments in Section 3.8 show that this assumption is met in well-known benchmarks such as TPC-C and RUBiS, this is an application-dependent property. In fact, the unrestrained use of speculation in adverse workloads can lead to excessive misspeculation and degrade performance.

In order to enhance the performance robustness of STR, we coupled it with a lightweight self-tuning mechanism that dynamically decides whether to enable or disable the speculative mechanisms, depending on the workload characteristics. The tuning scheme takes a black-box approach that is agnostic of the data store implementation and also totally

Algorithm 2: Partition protocol

```
1  upon receiving [read, Tx, Key] from partition P
2    reply P.readFrom(Tx, Key)
3  upon receiving [prepare, Tx, Key, Value, PC] from partition P
4    reply P.prepare(Tx, Key, Value, PC)
5  upon receiving [commit, Tx, CT] from partition P
6    for each Key updated by Tx : KVStore.updateVers(Key, <Tx, final committed, CT>)
7    for each Tw ∈ Tx.WaitingReaders : unblock Tw
8  upon receiving [abort, Tx] from partition P
9    for each Key v pre-committed by Tx : KVStore.removeVersion(Key, Tx)
10   for each Tw ∈ Tx.WaitingReaders : unblock Tw
11  upon receiving [replicate, Tx, Key, Value, PC]
12    abort all conflicting pre/local committed transactions and transactions read from them
13    KVStore.insertVers(Key, <Tx, pre-committed, max(Key.LastReader+1, PC), Value>)
14    reply [prepared, max(Key.LastReader+1, PC)] to Tx.Coord
15  function <value, transaction> readFrom(tx Tx, key Key)
16    while (true)
17      Key.LastReader ← max(Key.LastReader, Tx.ST)
18      <Tw, State, Value> ← KVStore.latest_before(Key, Tx.ST)
19      if State = committed
20        return <Value, Tw>
21      else if State = local committed and Tx is originated locally
22        add dependence from Tx to Tw
23        return <Value, Tw>
24      else
25        Tw.WaitingReaders.add(Tx)
26        wait until unblocked by Tw
27  function <state, timestamp> prepare(tx Tx, key Key, value Value, timestamp PC)
28    while (true)
29      <Tw, Type, TS, _> ← the freshest version of Key in KVStore
30      if (TS > Tx.ST) return <aborted, ⊥>
31      if (Type = final-committed ∨ (Tx is originated locally ∧ Type = local committed))
32        break
33      if (Tx is originated locally) // Tw is pre-committed
34        wait until Tw to local commit ∨ final commit ∨ abort
35      else // Tx is not originated locally and Tw is either pre-committed or local committed
36        wait until Tw to final commit ∨ abort
37      if Type = local committed
38        add dependence from Tx to Tw
39      KVStore.insertVers(Key, <Tx, pre-committed, max(Key.LastReader+1, PC), Value>)
40      if P.isMaster() = true
41        send [replicate, Tx, Key, Value] to its replicas
42      return <prepared, max(Key.LastReader+1, PC)>
43  function void localCommit(transaction Tx, timestamp LCT, key Key, value Value)
44    KVStore.updateVers(Key, <Tx, local committed, LCT, Value>)
45    for each Tw ∈ Tx.WaitingReaders : unblock Tw
```

transparent to application developers. It relies on a simple feedback-driven control loop, steered by a centralized process. Every three minutes, the centralized process enables and then disables speculative reads each for a short period of time (10 seconds) in all nodes. Then the process gathers performance measures from all nodes, compares the throughput achieved with speculative reads enabled and disabled, and accordingly configures the system.

We opted for a simple and quickly converging scheme, instead of more complex ap-

proaches (e.g., based on off-line trained classifiers or more sophisticated on-line search strategies [116]), since our experimental findings confirm that, for a given workload, the decision whether or not to use speculation has a straightforward effect on throughput (no jitterlike behavior).

Our current implementation allows system administrators to initiate the self-tuning process periodically or upon request. The current self-tuning scheme could thus be naturally extended to detect statistically meaningful changes of the average input load via robust change detection algorithms, like CUSUM [11], and react to these events by re-initiating the self-tuning mechanism.

3.4.6 Fault tolerance

We note that STR requires that a transaction coordinator collects replies from all the nodes that replicate data updated by the transaction. This approach assumes the existence of an underlying group management toolkit providing, e.g., virtual synchrony guarantees [17]. It is then straightforward, upon a view change event, to purge faulty nodes and reconfigure the system to ensure progress. Specifically, the replicas of a partition that transit to a new view could rely on the virtual synchrony properties to ensure that they all deliver the same set of updates, as well as agree upon the new partition’s master, before starting to operate in the view.

We note also that, unless some additional mechanism is employed, in presence of replicas’ failures, property SPSI-3 could be violated, as illustrated in Figure 3.3. In the considered execution, transaction $T1$, with ST 80, reads a key K from replica 2. Since the LR variable (*last_reader*, which keeps track of the last transaction to have read a data item) is only updated on the replica that processes the read request, the information on $T1$ ’s read request is lost in case of fault of replica 2. It is then possible for an update transaction $T2$ to commit with FC smaller than 80: this happens since, after detecting the failure of replica 2 and excluding it from the system, $T2$ computes its FC on the basis of a set of prepare messages that no longer reflect the state of replica 2.

This issue can be tackled by enforcing that, in the new view, update transactions are attributed a FC “sufficiently” in the future to avoid the anomaly illustrated above, i.e., their FC s should be set large enough to be larger than the ST s of any transaction that could have read a key on any failed replicas. This could be achieved by computing, during the reconfiguration phase following a replica’s fault, the upper bound on the clock skew among the replicas in the active view before the fault occurred, which we denote as ϵ . When resuming transaction processing in the new view, it is sufficient to ensure that the FC timestamps computed by any transaction coordinator are larger than or equal to $\epsilon + t_{reconf}$, where t_{reconf} is the local time, at each transaction coordinator, in which the new view was delivered. This can be ensured by having the transaction coordinators correct the FC timestamps they determine in Alg. 1 line 34, so that these are guaranteed to be larger or equal than $\epsilon + t_{reconf}$.

It should be noted that the above described fault-tolerance mechanisms are not implemented in the current STR prototype.

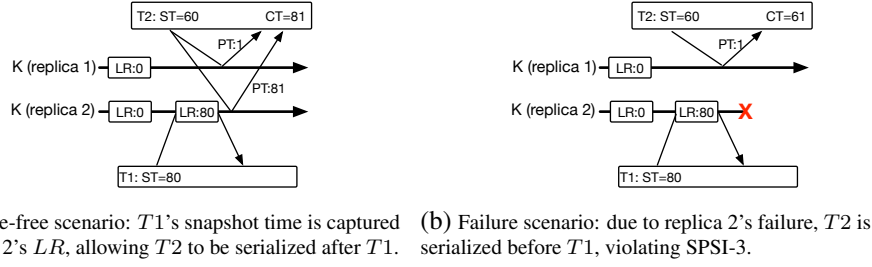


Figure 3.3 – Exemplifying how replica failure may compromise SPSI-3. *LR* and *PT* denote last reader and prepare time, respectively.

3.5 Correctness proof

In this section, we prove that STR is safe, i.e., it does not violate any SPSI property, and live, i.e., no transaction may be blocked for infinite time and will eventually either commit or abort. The following analysis assumes a failure-free scenario. A discussion on how to cope with failure scenarios is provided in Section 3.4.6.

3.5.1 Safety

Theorem 1. *SPSI-1: For each data item, its final committed versions are totally ordered*

Proof. Assume, by contradiction, that two transactions $T1$ and $T2$ update K and both final commit producing a version of K tagged with the same FC . In order to final commit and produce a new version of key K , a transaction must first prepare on all replicas of K . Assume without loss of generality that at a given replica, R , transaction $T1$ enters the prepared state before $T2$ does. At the time in which $T2$ requests to prepare on R , the version of key K produced by $T1$, which we note K_{T1} can be in three possible states:

- K_{T1} is pre-committed: in this case, if $T2.ST$ is smaller than the PC of K_{T1} , $T2$ is immediately aborted, which contradicts the assumption that $T2$ final commits. In case $T1$ originated at R , it will undergo a local commit phase and, possibly, K_{T1} will be attributed a LC timestamp larger than its current PC . In such a case, $T2$ waits until $T1$ local commits. This scenario is equivalent to the case in which $T2$ requests to prepare while K_{T1} is local committed — which we describe next. The only case left to discuss is the one in which $T1$ did not originate at R : in this scenario, $T2$ is blocked until $T1$ final commits. We discuss this scenario in the following (i.e., K_{T1} is final committed).
- K_{T1} is local committed: if the LC timestamp of K_{T1} is larger than $T2.ST$, $T2$ is aborted, which contradicts the assumption. Otherwise, if $T2$ originated at R , $T2$ pre-commits a new version of K and a dependence is added from $T2$ to $T1$. When $T1$ final commits two cases are possible: i) if $T1.FC$ is larger than $T2.ST$, $T2$ will be aborted — leading to contradicting the assumption; ii) if $T2.ST$ is larger than

$T1.FC$, according to Alg. 1 line 19 and 34, $T2$ will final commit with an FC that is at least $T2.ST + 1$. Therefore, $T2.FC$ will be necessarily different from $T1.FC$, which also contradicts the assumption.

- K_{T1} is final committed: if $T1.FC$ is larger than $T2.ST$, $T2$ will be aborted; otherwise, by the same argument used for the previous case (K_{T1} is local committed), $T1$ and $T2$ will not be attributed the same FC . In both cases, we obtain a contradiction.

□

Theorem 2. *SPSI-2: At each node, the final and speculative committed versions of each data item are totally ordered.*

Proof. We also prove SPSI-2 by contradiction. Specifically, we assume that at a node R , which replicates key K , two transactions $T1$ and $T2$ update K , and consider the following three scenarios in which SPSI-2 would be violated: 1) both $T1$ and $T2$ local commit, producing two versions of K with the same LC , 2) $T1$ final commits a version with timestamp FC , whereas $T2$ local commits with a timestamp LC , such that $FC = LC$, and 3) both $T1$ and $T2$ final commit, producing two versions of K with the same FC . Note that the last case stands for a contradiction of SPSI-1, which we have already proven to be impossible. As such, here we focus on proving the impossibility of the first two cases.

We start by considering the first case. In this case, without loss of generality, we assume that at node R , transaction $T1$ enters the prepared state before $T2$ does. At the time in which $T2$ requests to prepare on R , the version of key K produced by $T1$, noted K_{T1} , can be in two possible states:

- K_{T1} is pre-committed: in this case, if $T2.ST$ is smaller than the pre-commit timestamp of K_{T1} , $T2$ is immediately aborted, contradicting the assumption that $T2$ local commits. Otherwise, $T2$ will be blocked until $T1$ local commits – a case that we describe next.
- K_{T1} is local committed: if the LC timestamp of K_{T1} is larger than $T2.ST$, $T2$ is aborted, which contradicts the assumption. Otherwise, according to Alg. 1 line 19 and 22, $T2$ will local commit with an LC that is at least $T2.ST + 1$ and, therefore, necessarily different from $T1.LC$ — also contradicting the assumption.

Next, we discuss the case when $T1$ final commits with timestamp FC and $T2$ local commits with timestamp LC , where $LC = FC$. There are two possible sub-cases here: either $T2$ prepares before $T1$ does, or, vice versa, $T1$ prepares before $T2$ does.

If $T2$ prepares before $T1$ does, then when $T1$ tries to prepare on R , the version created by $T2$, namely K_{T2} , can be in pre-committed or local committed state. By using the same argument employed to exclude the possibility that $T1$ and $T2$ can ever local commit with the same LC , it is possible to show that, in this sub-case, $T1$ either aborts, or final commits with a timestamp larger than $T2$'s LC . Thus, this sub-case is not possible.

Let us finally consider the sub-case in which $T1$ prepares before $T2$ does. When $T2$ tries to prepare at R , K_{T1} can be in three possible states. We have already discussed the scenarios in which K_{T1} is in pre-committed or local committed state, when $T2$ attempts to prepare. Thus, here, we only discuss the case in which K_{T1} is final committed. If $T1.FC$ is larger than $T2.ST$, $T2$ will be aborted; otherwise, according to Alg. 1 line 19 and 21, $T2$ will obtain a pre-commit timestamp at least as large as $T2.ST + 1$, and since we are assuming that $T2.ST > T1.FC$ and considering that $T2.FC \geq T2.ST + 1$ it follows that $T2.FC > T1.FC$. Hence, we obtain a contradiction. $\square$

Theorem 3. SPSI-3. (*Transactions read consistent speculative snapshots*) A transaction T originated at a node N with snapshot time ST must observe the most recent versions created by transactions that i) final commit with timestamp $FC \leq ST$ (independently of the node where these transactions originated), and ii) local commit with timestamp $LC \leq ST$ and originated at node N .

Proof. We prove SPSI-3 by contradiction. Assume there are two transaction $T1$ and $T2$: at time $t1$, $T1$ creates, through local/final commit, a version of K denoted as K_{T1} , which is the most recent version that is ever going to be local/final committed with $LC/FC \leq T2.ST$; however, at time $t2$, $T2$ reads K but does not observe K_{T1} . We discuss the case in which K_{T1} is local or final committed separately.

In case K_{T1} is local committed, K_{T1} can be local committed either no later than, or after $T2$ reads K , namely either $t1 \leq t2$ or $t1 > t2$. In the former case, when $T2$ reads K , K_{T1} must be local committed already; in the latter case, K_{T1} can either be non-existent or pre-committed. We discuss all these three cases below:

- non-existent: in this case, $T2$ reads the current latest version of K , which is different from K_{T1} , and also updates the *LastReader* of K , causing any later transaction updating K (including $T1$) to local/final commit with a timestamp at least $T2.ST + 1$. This contradicts the assumption that K_{T1} is the most recent version ever going to be local/final committed with $LC/FC \leq T2.ST$.
- pre-committed: in this case, $T2$ will be blocked and it will be unblocked when $T1$ local commits, which we will discuss below.
- local committed: in this case, if K_{T1} is indeed the most recent version below $T2$, according to the read rule, $T2$ will fetch the most recent version below its snapshot, i.e. K_{T1} . This contradicts the assumption that $T2$ misses K_{T1} .

Next, we consider the case that K_{T1} is a final committed version. When $T2$ reads K , K_{T1} can be non-existent, or in pre-committed, local committed (if $T1$ is a local transaction) or final committed state. As the above three states have already been discussed above, let us only discuss the case that K_{T1} is in final committed state. Basically, if K_{T1} is the latest version below $T2.ST$, by the same argument used for the previous case (local committed state), $T2$ will read K_{T1} , contradicting the assumption that $T2$ will miss K_{T1} . $\square$

Theorem 4. SPSI-4. *(No Write-Write Conflicts among Final Committed Transactions) The write-sets of any final committed concurrent transactions must be disjoint.*

Proof. By contradiction, we assume two concurrent conflicting transactions $T1$ and $T2$ both update a key K and final commit. Without loss of generality, we assume that $T1.ST \leq T2.ST < T1.FC$.

To final commit, both $T1$ and $T2$ have to prepare on all replicas of K . Assume R is a node that replicates K . We first consider the case that $T1$ prepares on R before $T2$ does so. In this scenario, when $T2$ requests to prepare K on R , the version of K created by $T1$ may be in the following states:

- pre-committed: in this case, since $T1.FC > T2.ST$, $T2$'s certification check on R will fail, causing $T2$ to abort, which contradicts the assumption.
- local committed: $T1$ may be local committed at R , which implies that $T1$ originated at R . In that case, if $T1.LC > T2.ST$, $T2$ will be aborted. Otherwise, if $T2$ was not originated at R , $T2$ will be blocked until $T1$ final commits — a case that we discuss next. If $T2$ was originated at R and $T1.LC \leq T2.ST$, $T2$ will be prepared and a dependence is added from $T2$ to $T1$ (line 38 of Alg. 2). Eventually when $T1$ final commits, since we have assumed that $T1.FC > T2.ST$, $T2$ will be aborted (line 49 of Alg. 1).
- final committed: as $T1.FC > T2.ST$, $T2$ will fail to prepare on R and abort, which contradicts the assumption.

Now let us consider the case in which $T2$ prepares K on R before $T1$ does. Note that according to lines 19 and 21 of Alg. 1, the PC , LC and FC of $T2$ will all be at least as large as $T2.ST + 1$, which is larger than $T1.ST$. Therefore, when $T1$ tries to prepare K on R , no matter in which state the version created by $T2$ is, according to lines 30 of Alg. 2, $T1$ will always be aborted. This contradicts the assumption that $T1$ final commits. $\square$

Theorem 5. SPSI-5. *(No Write-Write Conflicts among Transactions in a Speculative Snapshot) Let S be the set of transactions included in a snapshot. The write-sets of any concurrent transactions in S must be disjoint.*

Proof. We prove this property by contradiction, assuming that there exists a transaction T whose snapshot includes two concurrent transactions, $T1$ and $T2$, such that $T1$ and $T2$ have a write-write conflict on a key K . We note that a necessary condition in order for T to include in its snapshot any other transaction T' , is that T' must be either final committed or local committed. Thus, in the following we shall only analyze all possible scenarios in which $T1$ and $T2$ are either final or local committed and show that, by no means, T can include both of them in its snapshot.

$T1$ and $T2$ are both final committed. This is not possible, since SPSI-4 guarantees that if two concurrent transactions have write-write conflicts, they can not be both final committed. A contradiction.

T1 and T2 are both local committed. In this case, in order for $T1$ and $T2$ to be both included in T 's snapshot, $T1$ and $T2$ must be originated at the same node as T , say node R . Assume, without loss of generality, that at node R , transaction $T1$ enters the prepared state before $T2$ does. At the time in which $T2$ requests to prepare on R , the version of key K produced by $T1$, noted K_{T1} , can be in two possible states:

- K_{T1} is pre-committed: in this case, if $T2.ST$ is smaller than the pre-commit timestamp of K_{T1} , $T2$ is immediately aborted (lines 30 of Alg. 2), contradicting the assumption that $T2$ local commits. Otherwise, $T2$ will be blocked until $T1$ local commits – a case that we describe next.
- K_{T1} is local committed: since we are assuming that $T1$ pre-commits before $T2$ does, $T2$ can only succeed in pre-committing if $T2.ST > T1.LC$ — else, $T2$ would abort in lines 30 of Alg. 2. However, this implies that $T2$ is not concurrent with $T1$, yielding a contradiction.

T1 is local committed and T2 is final committed, or vice versa: without loss of generality, we assume $T1$ is local committed and $T2$ is final committed. We note that $T1$ should be originated at the same node as T , which we denote as R , in order for $T1$'s local committed versions to be readable by T .

We shall consider two scenarios: 1) K is replicated by R , and 2) K is not replicated by R . We discuss both scenarios in the following.

K is replicated by R . Since we assume that $T2$ final commits, this implies that $T2$ must be prepared on all replicas of K , including R .

We show next that R cannot be a master replica of K . Assume by contradiction that R is the master replica of K . In this case, if $T2$ prepares after $T1$ on R , $T2$ would be aborted (lines 30 of Alg. 2) when it requests to pre-commit at R - a contradiction, since $T2$ is assumed to final commit. If, conversely, $T2$ prepares before $T1$ on R , then $T1$ will be aborted at R when it tries to pre-commit - another contradiction, since $T1$ is assumed to local commit on R , which is only possible if $T1$ successfully pre-commits on R .

Let us now assume that R is a slave replica of K . There are two possible scenarios here: T includes in its snapshot first $T2$ and then $T1$, or vice versa. We first consider the case in which T include in its snapshot first $T2$ and then $T1$. In this case, when T reads, directly or indirectly, from $T2$, $T2$ must have already been prepared on all replicas of K , including R — a necessarily condition for $T2$ to final commit. We analyze two sub-cases, which correspond to the scenarios in which $T1$ attempts local commits on R before $T2$ prepares on R , and vice versa.

1. If $T1$ local commits on R before $T2$ prepares on R , according to Alg. 2 line 12, since $T1$ and $T2$ have a conflict on K , when R receives from the master replica of K the request to replicate $T2$'s pre-committed version of K , R would aborts $T1$.

2. If $T2$ prepares on R before $T1$ is local committed on R , then, since $T1$ conflicts with $T2$, $T1$ will fail to pre-commit on R .

In both cases, we obtain a contradiction.

Next, let us consider the case in which T includes in its snapshot first $T1$ and then $T2$. This implies that, before T includes $T1$ in its snapshot, $T1$ must be pre-committed on R . In order for T to later include $T2$ in its snapshot, $T2$ must have been replicated on R (as already discussed, this is a necessary condition for $T2$ to final commit). When this happens, though, $T1$ and T are aborted at R , before $T2$'s pre-committed versions are installed. This happens since $T1$ and $T2$ conflict and since T has read from $T1$, either directly or indirectly (see line 12 of Alg. 2 and lines 52-55 of Alg. 1). Therefore, T is guaranteed to abort before $T2$ final commits. This contradicts our assumption that T includes in its snapshot both $T1$ and $T2$.

K is not replicated by R . Let us consider the moment when T includes both $T1$ and $T2$ into its snapshot. At this point in time, the following inequations must hold:

- 1) $\min_value(T.OLCSet) \geq T.FFC$, since T has read, either directly or indirectly, from both $T1$ and $T2$, i.e., T did not block when executing line 15 of Alg. 1.
- 2) $\min_value(T1.OLCSet) \leq T1.ST$, because $T1$ updated a non-local key (Alg. 1 line 26),
- 3) $T.FFC \geq T2.FC$, because T has read, either directly or indirectly, from the final committed transaction $T2$. Thus, $T.FFC$ is necessarily larger than $T2$'s final commit timestamp (Alg. 1 line 14), and
- 4) $\min_value(T1.OLCSet) \geq \min_value(T.OLCSet)$, since T has read, either directly or indirectly, from $T1$. Thus, the smallest timestamp included in $T.OLCSet$ is necessarily larger than the smallest timestamp in $T1.OLCSet$ (Alg. 1 line 10).

By combining the above inequations, we can conclude that $T1.ST \geq T2.FC$. This implies that $T1$ and $T2$ are not concurrent which contradicts the assumption.

□

Theorem 6. SPSI-6. (*No Dependencies from Uncommitted Transactions*) A transaction can only be final committed if it does not depend on any non-final committed transaction.

Proof. As shown in Alg. 1 lines 40-55, a transaction T can only be final committed when all the transactions it depends on have final committed (with a smaller FC timestamp than T 's ST timestamp); when a transaction aborts, it aborts all transactions that depend on it. As such, a transaction can never commit if it still depends on any uncommitted transaction. □

3.5.2 Liveness

This section is devote to analyzing the liveness of STR and, more precisely, to prove that in a failure free execution transactions will either abort or commit after a finite amount of time (refer to Section 3.4.6 for a discussion on how to handle failure scenarios).

In Alg. 1, we can see that during its execution, a transaction T can only be blocked in the following four scenarios:

- S1. when T reads a key for which a pre-committed or a remote local-committed version exists that may be final committed with a smaller timestamp than $T.ST$ (lines 9 and 13 of Alg. 1).
- S2. when T reads a key for which a local or final committed version exists, whose creating transaction may conflict with another final or local committed transaction T has previously read from, i.e. the FFC of T is larger than the minimum value in its $OLCSet$ (line 15 of Alg. 1).
- S3. while executing its certification phase (line 32 and 37 of Alg. 1).
- S4. while waiting for resolving any of its speculative dependencies, which are developed whenever T reads a local committed transaction (line 33 of Alg. 1).

We inspect these scenarios in the following, and show that T can never be blocked indefinitely.

S1. The blocking statements in both lines 9 and 13 of Alg. 1 are rooted to line 26 of Alg. 2. This blocking condition occurs in two cases: i) if T attempts to read a version in pre-committed or local committed state that was created by a transaction T' originated on a different node than T ; ii) if T attempts to read a version in pre-committed state that was created by a transaction T' . In both cases, T' is executing its certification phase and, as we will discuss later (scenarios S3 and S4), T' will eventually final commit or abort, unblocking T .

S2. T blocks in line 15 of Alg. 1 if its updated FFC (reflecting the reading of the version returned by the `readFrom()` method) is larger than $\min_value(T.OLCSet)$. This implies that there must exist one or more unsafe transactions in the $OLCSet$ of T whose ST is smaller than the FFC of T . However, such unsafe transactions must be in local committed state, which implies that they are executing their certification phase and, as we shall discuss in scenarios S3 and S4, they will eventually abort or final commit and remove themselves from the $OLCSet$ of T . As a result, eventually, T will be unblocked.

S3. During the execution of T 's (local or global) certification phase, a partition may delay replying to T 's certification request if a key that T tries to prepare has already been prepared or local committed by another transaction (line 34 and 36 of Alg. 2).

Assume, by contradiction that T is blocked indefinitely due to the existence of a cycle in the transactions' wait-for graph [15] composed by a set of transactions $T_1, \dots, T_i, \dots, T_N$, where $T_N = T$, such that: P1) transaction T_i is blocked by transaction T_{i-1} , given that T_i is attempting to prepare a key that was already prepared or local committed by T_{i-1} , and P2) T_1 is blocked by T , for analogous reasons (recall that $T = T_N$). However, according to line 30 of Alg. 2, a transaction T' can only be blocked by a transaction T'' if either $T'.ST > T''.LC$ or $T'.ST > T''.PC$ hold (else T' is aborted).

Considering that the LC of a transaction is necessarily larger than the its PC (line 22 of Alg. 1), property P1 implies that:

$$\forall i \in [1, N] : T_i.ST > T_{i-1}.PC$$

and, since the ST of a transaction is necessarily smaller than its PC , we obtain that $T_N.ST > T_1.ST$. However, for property P2 to hold, it must be: $T_1.ST > T_N.PC > T_N.ST$. Hence, we obtain a contradiction and the proof follows.

S4. So far we have proved that T cannot be indefinitely blocked while reading or while executing its certification phase. It remains to prove that T does not block indefinitely because of some unresolved dependency (line 33 of Alg. 1). Let us assume, by contradiction, that T reads speculatively from a local committed transaction T' and that T' never final commits or aborts.

Let us assume, without loss of generality, that there exists a series of dependent transactions $T_1, \dots, T_i, \dots, T_{N-1}, T_N$, such that transaction T_i depends on (i.e., it has read a version local committed by) transaction T_{i-1} , where we have denoted with T_{N-1} and T_N , T' and T , respectively. Since a transaction can only speculatively read from another local committed transaction if the ST of the former is larger than the LC of the latter, and since the LC of a transaction is necessarily larger than its own ST , it follows that the above series of dependent transactions cannot be cyclic. Thus, the first transaction in the series, namely T_1 , cannot have any speculative dependency and, as such, it will eventually commit or abort — based on the above arguments for scenarios S1, S2 and S3 — and will eventually unblock T_2 , which in its turn will unblock the next transaction in the series and so forth, until eventually unblocking T .

3.6 Ensuring real-time ordering and serializability

3.6.1 Real-time ordering

SPSI extends the SI specification introduced by Pedone et al. [41] and both specifications, as discussed in Section 3.3, do not impose real-time ordering, i.e., they do not guarantee that:

- P_{RTO} : If a transaction commits at real time t , its effects must be visible to any transaction starting after real time t .

In fact, according to property SI-2, transactions determine their snapshots using the physical clock of their originating node. Since we assume physical clocks be possibly subject to arbitrary skews, it follows that a version committed by transaction T_1 at real-time t may be missed by a transaction T_2 starting at real-time $t' > t$ if, due to clock skews, the snapshot time (ST) of T_2 is smaller than the final commit timestamp (FC) attributed to T_1 . We note that this sort of anomaly, illustrated also in Figure 3.4a, affects not only STR, but also various other recent systems that adopt the same (or a similar) SI definition, e.g., [36, 30, 114].

In the following, we shall discuss how to extend STR to rule out this type of anomaly and ensure P_{RTO} . While there exist several approaches to enforce RTO for transactional protocols [26, 65], here we show how to integrate in STR the technique proposed by Spanner [26], which we call *delayed commit*.

Delayed commit

We integrate the approach employed by Spanner [26] to ensure real-time order, which achieves so by delaying the commit operation of transactions, after having attributed their commit timestamp. Specifically, assume that the upper bound of clock skew across system nodes is ϵ and the commit timestamp of a transaction is t : a node can only commit that transaction when its physical clock has passed $t + \epsilon$. Additionally, visibility of committed versions is prevented during the delayed commit phase: this is achieved by delaying by ϵ time units the sending of the final commit messages at the transaction coordinator, which has the effect of extending the effective duration of the pre-commit locks by ϵ .

Figure 3.4 illustrates how this technique can be used to avoid violating real-time order. In Fig. 3.4a, although $T2$ starts after $T1$ commits in real-time, due to clock skew, $T2$ is assigned snapshot time (ST) 100 at its origin node ($N1$), whereas $T1$ is attributed a final commit timestamp equal to 110. As such, if $T2$ reads item A , which was modified by $T1$, $T2$ will miss $T1$'s updated version, violating real-time order. By applying the commit delay technique (Fig. 3.4b), $N1$ only finalizes the commit of $T1$ (i.e., sending of the final commit messages to involved partitions and returning from the commit request issued by the application) after its local clock is larger than $110 + \epsilon$ (120). This way, $T1$ and $T2$ become concurrent transactions and are freed from any real-time ordering constraint — hence, it becomes legal for $T2$ to “miss” the version produced by $T1$. This approach essentially delays the commit of a transaction T long enough to guarantee the following key property:

- P_{DC} : Any transaction that starts in real-time after the commit of T obtains a snapshot timestamp larger than the commit timestamp attributed to T .

In other words, transactions that start after $T1$ commits in real-time and that, due to clock skews, may possibly miss $T1$'s updates are guaranteed to become concurrent with $T1$.

How to integrate with STR

Integrating the delayed commit technique into STR would be relatively straightforward. This technique, in fact, implies simply injecting a delay before line 44 of Alg. 1, namely after having established the final commit timestamp (FC) and determined whether any local, dependent transaction should be aborted.

Impact on correctness

We first discuss why introducing the delayed commit mechanism in STR does not compromise its correctness with respect to the SPSI's specification. Next we discuss why its use would ensure P_{RTO} .

SPSI. The delayed commit technique is equivalent to assume that the execution of line 34 of Alg. 1 is delayed by ϵ time units. However, as discussed in Section 3.2, our system

model assumes no bound on the relative speed of processes, so incorporating this technique into the presented protocol would not endanger its correctness with respect to the SPSP's specification.

P_{RTO} . We start by noting that the integration of delayed commit in STR would allow, just like in Spanner, to ensure property P_{DC} . Let us now discuss the case of non-speculative and speculative reads, separately:

- *non-speculative reads*: non-speculative reads, in STR, use the same visibility logic as in Spanner (a transaction's read returns the most recent version tagged with a commit timestamp smaller than its snapshot timestamp). Since the use of delayed commit ensures P_{DC} also in STR, it follows that a transaction starting at time t and reading non-speculatively committed data item d cannot miss any version final committed by a transaction by time t — thus, ensuring P_{RTO} .
- *speculative reads*: Through the use of speculative reads, it is possible for a transaction T to read local committed data versions, which will only be final committed (unless the creating transaction is aborted due to misspeculation) after the start of T . However, P_{RTO} only requires that the effects of the transactions committed by real time t are visible to transactions starting after time t , but does not preclude transactions from observing the effects of transactions that do commit after real time t .

Impact on the effectiveness of speculation

In STR speculation can fail only in two scenarios: (i) speculative reads return a version locally committed by a transaction that eventually commits with a final commit timestamp (FC) larger than the snapshot timestamp (ST) of the reading transaction, or (ii) speculative reads return a version locally committed by a transaction that eventually aborts due either to contention or to case (i). We note that, since delayed commit does not increase the final commit timestamps of transactions, it does not affect the chance that speculation will fail due to case (i). The likelihood of transactions' aborts due to contention, however, may increase as a consequence of the increase in the duration of the pre-commit locks caused by the delayed commit mechanism. On the one hand, this implies that, in contention-prone workloads, the use of the delayed commit technique may reduce the effectiveness of speculative reads, i.e., increase the likelihood that speculation will lead to transactions' aborts. On the other hand, though, when speculation does succeed, e.g., in low-contentions scenarios, the performance gains achievable through speculation are expected to be amplified: via the use of speculative reads, in fact, it is possible to bypass the wait for pre-commit locks held by concurrent transactions, whose duration is amplified by the delayed commit mechanism.

3.6.2 Serializability

Section 4.7 has described how STR ensures SPSP, a consistency criterion that extends SI to provide additional correctness guarantees regarding the speculative execution of transactions. Although SI has been widely adopted and is deemed as sufficient for many

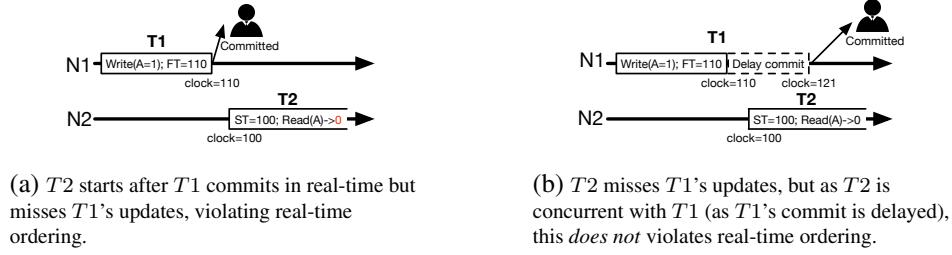


Figure 3.4 – Exemplifying how real-time ordering can be violated (a), and how commit delay can avoid the problem (b). $N1$ and $N2$ denote two nodes; $T1$ and $T2$ denote two transactions; arrowed lines denote the progress of time.

applications, it is known to allow for non-serializable executions [12] that may endanger correctness in certain scenarios. Since SPSI imposes the same restrictions as SI on non-speculative executions, i.e., on transactions that final commit, it follows that SPSI admits the same type of well-studied anomaly that also affects SI. This anomaly, which is often referred to as *write skew*, is illustrated in Figure 3.5, where two concurrent transactions withdraw from a person's saving and credit account, respectively, provided that the total balance after the withdrawal does not become negative. Since these two transactions do not generate any write-write conflict, when using either SI or SPSI, both transactions can commit, breaking the application's invariant that imposes the total balance to never go negative.

In the following, we discuss how to extend STR to ensure serializability. In the literature there exists a number of works that investigated how to ensure serializability on top of database systems that provide SI via the use of middleware/application-level solutions (e.g., the acquisition of additional locks [42] or the detection of specific conflict patterns [20]) that do not requiring any change to the underlying SI-compliant DBMS. We note that those solutions may be adaptable, at least in principle, to operate also in a partially-replicated distributed settings with loosely synchronized clocks, i.e., the deployment scenario targeted by STR. However, we argue that serializability may be attained in a simpler way by introducing a relatively straightforward alteration to the certification logic employed by STR.

Basically, instead of checking whether transactions have only generated write-write conflicts during (both local and global) certification, it is necessary to further check whether transactions have incurred any read-write conflict, i.e. whether the read-set of the transaction being certified has been invalidated by the commit of a concurrent transaction. More specifically, when certifying, both locally and globally, a transaction T , the following additional check should be performed: for every key in T 's read-set, the timestamp of the most recently final or local committed version must be smaller than or equal to the snapshot timestamp of T . If this is not the case, the transaction should be aborted. It is easy to see that this technique would avoid the problem illustrated in Figure 3.5: as Tx_1 and Tx_2 develop read-write conflicts, only one of the two would pass validation and be committed.

Nevertheless, note that this mechanism requires tracking transaction's read set in order to perform read-write conflict check. This may introduce large performance penalty especially for update transactions that perform large number of reads, e.g., range scans.

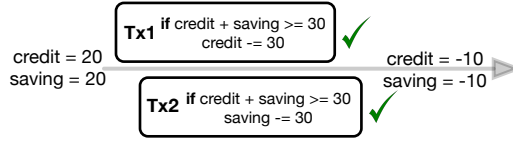


Figure 3.5 – An example execution illustrating the *write skew* anomaly using SI.

3.7 Related Work

This section is devoted to analyzing the relations between the novel contributions presented in this chapter, in particular SPSI and STR, with respect to existing research. This discussion extends the review already performed in Chapter 2, in the light of the concepts and techniques introduced in the previous sections. The analysis of the state of the art is structured in two logical parts: first, a comparison with existing replication protocols targeting geo-replication is presented. Next, we discuss relations with protocols also using speculation techniques.

Geo-replication. The problem of designing efficient mechanisms to ensure strong consistency semantics in geo-replicated data stores has been extensively studied. As already discussed in Section 2.3, there are mainly two complement approaches to implement geo-replicated systems: the state-machine replication (SMR) approach [122, 38] and the deferred update replication (DUR) approach [26, 36, 73]. The DUR approach is more scalable than SMR in update intensive workloads [124, 66] and, unlike SMR, it can seamlessly support non-deterministic transactions [104]. Section 2.3.1 mentioned that the main downside of the DUR approach is that locks must be maintained for the whole duration of transactions’ global certification, which can severely hinder throughput [121]. STR builds on the DUR approach and tackles its performance limitation via speculative techniques.

The property introduced in this work, SPSI, is related to PSI (Parallel Snapshot Isolation) [114], a consistency criterion that relaxes SI in order to reduce latency in geo-distributed data stores. When compared with SPSI, PSI specifies a weaker consistency criterion for final committed transactions: PSI requires that transactions read the most recent committed version of some data *only if* this is created by a transaction that originated at the same site. This allows for anomalies that are not possible in SI (called long forks [114]), and that are also excluded by SPSI, which guarantees SI-semantics for final committed transactions, i.e., they only observe the most recent committed version independently from the site in which it was originated. Further, PSI prohibits transactions from reading any version that is not final committed, which represents one of the key motivations underlying the definition of SPSI: sparing transactions from waiting for pre-commit locks to be released, while still providing rigorous consistency guarantees to shelter applications from arbitrary concurrency anomalies.

Speculation. The idea of letting transactions “optimistically” borrow, in a controlled manner, data updated by concurrent transactions has already been investigated in the past. As already discussed in Section 2.4.1, several protocols, such as SPECULA [99] and Aggro

[90], have applied this idea to local area clusters in which data is fully replicated via total-order based coordination primitives. Jones et. al. [67] applied this idea to partially replicated/distributed databases, by relying on a central coordinator to totally order distributed transactions. These solutions provide consistency guarantees on executing transactions (and not only on committed ones) that are similar in spirit to the ones specified by SPSI. However, some of these systems [90, 99] adopt a full-replication scheme, which requires all replicas to store the full dataset and apply all updates. This significantly hinders their scalability. Other systems, e.g., [67], instead, rely on the use of a global sequencer, which can become a system bottleneck and imposes unacceptably large latency in geo-distributed settings.

Other works in the distributed database literature, e.g., [59, 96, 49], have explored the idea of speculative reads (sometimes referred to as *early lock release*) in decentralized transactional protocols for partitioned databases, i.e., the same system model assumed by STR. However, these protocols provide no guarantees on the consistency of the snapshots observed by transactions (that eventually abort) during their execution and may expose applications to subtle concurrency bugs, such as the ones exemplified in Figure 3.1.

Another form of speculation that strives to reduce perceived-latency by exposing preliminary results to external clients, i.e., *external speculation* as mentioned in Section 2.4.1, has been explored by various works. Helland et. al. advocated the *guesses and apologies* programming paradigm [60], in which systems expose preliminary results of requests (*guesses*), but reconcile the exposed results if they are different from final results (*apologies*). A similar approach is adopted also in other recent works, like PLANET [93] and ICG [54]. Unlike STR, which is totally transparent to programmers, these approaches employ a form of external speculation, which requires source-code modification to incorporate compensation logics. Furthermore, these approaches are designed to operate on conventional storage systems, which do not support speculative reads of pre-committed data. As such, although these approaches may reduce user-perceived latency, they do not tackle the problem of reducing transaction blocking time, as STR does. We will provide experimental evidence supporting this claim in Section 3.8.

Some of the speculative transaction processing systems mentioned above, e.g., SPEC-ULA [99] and PLANET [93], rely on self-tuning mechanisms aimed at autonomously determining whether the use of speculation may be beneficial or not. As already mentioned, STR employs an ad-hoc self-tuning mechanism that aims at pursuing an analogous goal, i.e., dynamically enabling or disabling speculation based on the workload characteristics. More in general, there exists a large literature on self-tuning of transactional systems [39, 92, 89], which has shown the feasibility of using automatic techniques to predict and/or react timely to workload changes.

3.8 Evaluation

This section presents an extensive experimental study aimed at answering the following key questions:

1. What performance gains can be achieved by STR by allowing transactions to speculatively read pre-committed data?
2. How does STR compare with systems, like PLANET [93], which employ external speculation techniques and that, unlike STR, require programmers to develop compensation logics to deal with possible misspeculations?
3. Which workload characteristics have the strongest impact on the performance of STR?
4. How relevant is the Precise Clocks technique, when used in conjunction with both speculative and non-speculative protocols?
5. How effective is STR’s self-tuning mechanism to ensure robust performance in presence of workloads that are not favourable to speculative techniques?

Baselines. The first baseline protocol we consider is Clock-SI [36], which we extended to support replication, as explained in Section 3.4.1. We refer to this protocol as ClockSI-Rep. ClockSI-Rep is representative of state of the art transactional protocols based on decentralized physical clocks and it provides Snapshot Isolation, namely the consistency guarantee that SPSI extends to accommodate speculation. Thus, ClockSI-Rep is an appropriate baseline to evaluate the performance gains achievable by STR thanks to the use of speculative reads and Precise Clocks.

The second baseline we consider is representative of recent approaches [93, 60, 54] that propose programming models aimed to support *external speculation techniques* (in contrast to STR’s internal/transparent speculation), i.e., exposing uncommitted results to clients. Supporting it comes at the cost of extra complexity for the programmers, who are forced to identify the possible concurrency anomalies that may affect their programs and develop the corresponding compensation logics (which is not needed for STR). We build this baseline, which we call Ext-Spec, by developing a variant of ClockSI-Rep that externalizes to client the results of a transaction, once it passes its local certification phase and is still undergoing its global certification phase. Note that no compensation logic is executed when using Ext-Spec: this is done for simplicity and since in the considered benchmarks, speculation can lead only to the production of incorrect replies to clients, but does not compromise the internal consistency of the server-side of the application. It should be noted that this choice actually favors Ext-Spec, as it spares this baseline from the additional overheads associated with the execution of potentially complex compensation logic.

Since Ext-Spec and ClockSI-Rep share the same (distributed) concurrency control mechanism, as we will see, they deliver very similar peak throughput, final latency and abort rate. However, Ext-Spec’s use of external speculation can reduce speculative (but not final) latency, with respect to ClockSI-Rep.

Finally, our evaluation is focused on evaluating performance in absence of failures. Thus neither STR, nor the the baselines used in this study, implement the additional fault-tolerance mechanisms discussed in Section 3.4.6.

Experimental setup. We implemented the baseline protocols and STR in Erlang, based on *Antidote*², an open-source platform for evaluating distributed consistency protocols (such as the one in [5]). More precisely, the in-memory backend of Antidote (which provides a key-value store interface) has been extended to develop fully-fledged prototypal implementations of STR and of the aforementioned baselines. The code of all protocols used in this study is publicly accessible at <https://github.com/marsleeezm/STR>.

Our experimental testbed is deployed across nine DCs of Amazon EC2 spanning 4 continents, each of which consists of three m4.large instances. We use a replication factor of six, so each partition has six replicas, and each instance holds one master replica of a partition and slave replicas of five other partitions. These settings ensure that only a small fraction (less than <5%) of reads target remote partition. This choice allows us to evaluate the gains that stem from accelerating the commit phase using speculation techniques, which is the problem that STR aims at tackling. In fact, in scenarios with low locality/lots of remote reads, we expect the gains from STR to decrease, as STR does not allow transactions to speculatively read remote data items that are being updated so the relative benefit of STR will decrease. Further, we argue that in realistic settings, it is desirable to optimize data partitioning schemes to minimize the frequency of remote reads and avoid incurring high latency both during transaction execution — due to the need for fetching remote data upon reads — and in the commit phase — else, a large number of geo-distributed servers would have to be involved in the 2PC protocol.

Load is injected by spawning one thread per emulated client in some node of the system. Each client issues transactions to a pool of local transaction coordinators and retries a transaction if it gets aborted. We use two metrics to evaluate latency: the *final latency* of a transaction is calculated as the time elapsed since its first activation until its final commit (including possible aborts and retries); for Ext-Spec, we report also the *speculative latency*, which is defined as the time since the first activation of a transaction until its last speculative commit, i.e., the one after which it is final committed. Besides reporting abort rate, for Ext-Spec we also report the rate of *external misspeculation*, i.e., the percentage of transactions that were speculatively committed but finally aborted triggering the activation of some compensation logic (which we do not implement in this study, for simplicity). Each reported result is obtained from the average of at least three runs. As the standard deviations are low, we omit reporting them in the plots to enhance readability.

Unless otherwise specified, STR uses the self-tuning mechanism described in Section 3.4.5 to enable and disable the use of internal speculation. The reported results for STR refer to the final configuration identified by the self-tuning process.

3.8.1 Synthetic workloads

Let us start by considering a synthetic benchmark, which allows for generating workloads with precisely identifiable and very heterogeneous characteristics. The synthetic benchmark generates transactions with zero “think time”, i.e., client threads issue a new transaction as soon as the previous one is final committed.

²<https://github.com/SyncFree/antidote>

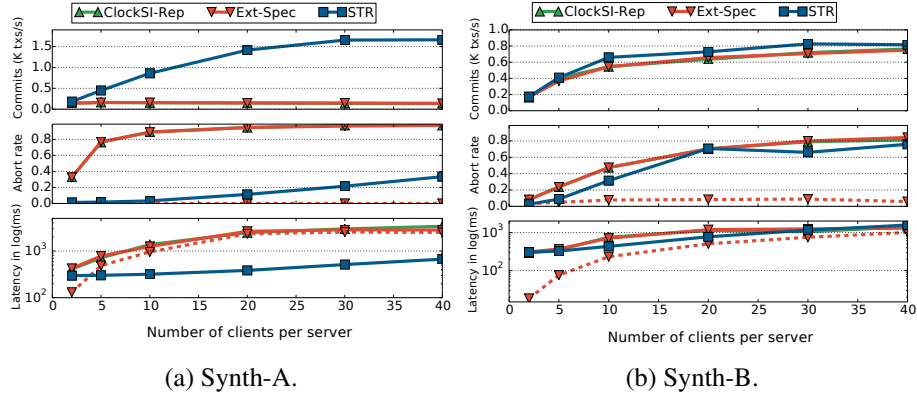


Figure 3.6 – The performance of different protocols for two synthetic workloads, representative of a favourable (Synth-A) and an unfavourable (Synth-B) scenario for internal speculation. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

Transaction and data access. Each node in the system hosts a single partition for which it serves as the master replica, yielding a total of 27 distinct partitions. Each node also serves as slave replicas for 5 additional partitions.

Each data partition has two million keys that are logically subdivided into two regions, called local and remote regions, where each region contains one million keys. The first region is only accessed by locally originated transactions, and the second region is only accessed by transactions that originate at a different node. Within each of these regions, we assume the existence of a “hotspot”, i.e., a relatively small set of keys that have a higher probability of being accessed, and we adjust the size of the hotspot to control contention rate. This allows adjusting in an independent way the likelihood of contention among transactions initiated by the same local node (local contention) and among transactions originated at remote nodes (remote contention), as we will detail, shortly.

A transaction reads and updates 10 keys, which are all selected from partitions for the originating node serves as either master or slave replica. 80% of the accesses of a transaction target the partition for which the originating node is the master replica of. The remaining 20% of the accesses target some partition for which the originating node serves as a slave replica. For the sake of brevity, we refer to the former partitions as *master partitions* and to the latter ones as *slave partitions*. Once a transaction has determined whether to target a master/slave partition, it accesses with 10% of probability the hotspot of the corresponding local/remote region, respectively.

We consider two workloads, which we obtain by varying the size of the hotspot sizes in the local and remote data partitions in order to synthesize two extreme scenarios that can be seen as representative of best and worst cases for internal speculation:

1. a workload, noted Synth-A, which generates very high local contention, by configuring the hot spots of master partitions to contain only a single key, but very low remote

contention, by configuring the hot spot of slave partitions to contain 800 keys. Due to the high likelihood of local contention, transactions are very likely to speculatively read versions that were local committed by some concurrent local transaction. Since remote contention is very low, though, internal speculation is very likely to succeed.

2. a workload, noted Synth-B, which has both very high local and remote contention, by using 10, resp. 3, keys in the hot spots of local, resp. remote, partitions. Like in workload Synth-A, transactions frequently use speculative reads, but, in this case, internal speculation is almost certainly doomed to fail due to the high remote contention.

Synth-A. Fig. 4.4.(a) clearly highlights the potential benefits that internal speculation can provide in favourable workload conditions. Both ClockSI-Rep and Ext-Spec fail to achieve any scalability and thrash, due to high abort rates (see middle plot), as soon as the degree of concurrency in the system grows to more than 2 clients. Conversely, STR scales almost linearly up to 20 clients and throughput saturates only at around 40 clients, achieving a $11.5\times$ gain with respect to both baselines (which achieve very similar throughput levels). Also, the abort rate of STR is significantly lower than for the two baseline protocols. This is explicable considering that, with the baselines, any transaction T that read a key pre-committed by some concurrent transaction T' is forced to block; when T' commits, it is very likely that T' generates a commit timestamp larger the read snapshot of T , which causes T to abort. In the same scenario, though, STR would allow T to speculatively read from T' ; also, the commit timestamp attributed to T' by Precise Clocks is likely to be smaller in absolute terms, and, with a higher probability than for the baselines, also smaller than the read timestamp of T . In this case, STR spares T from aborting, as well as from blocking — this allows STR not only to minimize the wasted work due to transactions' rollbacks, but also to enhance the degree of parallelism sustainable by the system.

It should be noted that since local contention dominates in this workload, most of the aborts occur during the local certification phase of transactions. Also, if transactions pass local certification, they are likely to avoid conflicts with remote transactions and, hence, commit with high probability. These considerations explain why Ext-Spec incurs an abort rate that is very similar to the one of ClockSI-Rep and to incur a very small external mis-speculation rate.

As for the latency, the bottom plot shows about one order magnitude smaller final latency for STR compared to the baselines with more than 2 clients. This is due to the fact that both ClockSI-Rep and Ext-Spec are thrashing due to high contention in this load range. For analogous reasons, the speculative latency of Ext-Spec is only lower than the final latency of STR at very low load (2 clients), where the abort rate is still relatively low.

Synth-B. Fig. 4.4.(b) shows that, even in such an unfavourable workload for internal speculation, STR can provide robust performance that is at par with the baseline protocols. Thanks to its self-tuning capabilities, in fact, STR automatically disables the use of speculative reads for 30 or more clients, which correspond to load levels in which internal speculation has an adverse effect on performance.

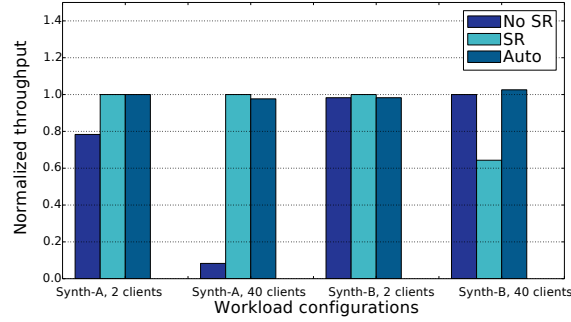


Figure 3.7 – Normalized throughput with respect to the best performing static configuration. *No SR*/*SR* denote enabling/disabling statically speculative reads in STR; *Auto* denotes the use of the self-tuning technique presented in Section 3.4.5.

# of keys Techniques	10	20	40	100
Physical	1/59%	1/60%	1/60%	1/72%
Precise	1.07/38%	1.07/38%	1.1/35%	1.41/48%
Physical SR	0.68/84%	0.57/83%	0.59/77%	0.97/75%
Precise SR	1.22/47%	1.21/44%	1.31/36%	1.59/49%

Table 3.1 – Normalized throughput/abort rate of different techniques, varying a transaction’s number of keys to update. *Physical*/*Precise* denotes the use of Physical Clocks/*Precise* Clocks; *SR* denotes that speculative reads are enabled. Throughputs reported in each column are normalized according to the throughput of ‘Physical’ in that column.

This is illustrated in Fig. 3.7, which reports the performance achieved by STR when statically configured to enable or disable speculative reads, as well as when using the self-tuning mechanism to select between these two configurations. More in detail, the y-axis of this figure reports the throughput of each variant of STR normalized with respect to the throughput of the variant that achieves best performance for the considered workload and number of clients (on the x-axis).

By Fig. 3.7, we can observe that, indeed, the use of speculative reads reduces throughput by around 40% in workload Synth-B with 40 clients and that the proposed self-tuning scheme can correctly identify the optimal configuration. By this plot, we can also observe that the choice of enabling/disabling internal speculation is not only affected by the workload type — as expected, speculative reads are beneficial in Synth-A but they are not in Synth-B — but also by the level load, fixed a given workload — speculative reads do not actually penalize throughput in Synth-B with 2 clients. Moreover, Figure 3.7 shows that without enabling speculative techniques, STR achieves similar throughput as the non-speculative baseline. This represents an experimental evidence supporting the efficiency of the proposed mechanism.

Benefits and overhead of Precise Clocks. This experiment aims at quantifying the benefits

stemming from the use of the Precise Clocks mechanism, when used in conjunction with both speculative and non-speculative protocols. To this end, in Table 3.1, we consider four alternative systems obtained by considering ClockSI-Rep (noted *Physical*) and extending it to use Precise Clocks (noted *Precise*) and/or speculative reads (noted *SR*). In this study we vary the transactions’ duration, and hence the corresponding abort cost, by varying the number of keys updated by a transaction. To avoid contention level to grow excessively fast when increasing the number of keys accessed by transactions, the key space is increased by the same factor.

Table 3.1 shows that Precise Clocks significantly reduces abort rate and can achieve as much as 38% of throughput gain over Physical Clock for a non-speculative protocol. Generally, the more keys transactions update, the larger is the abort cost and the larger the throughput gain achieved by Precise Clocks. Another interesting result is that enabling speculative reads with Physical Clock actually has negative effects on abort rate and throughput. In fact, as we have discussed in 3.4.3, physical clock based protocols, like Clock-SI or Spanner [36, 26], tend to generate large commit timestamp, which reduces the chances that speculative reads succeed. Finally, the collective use of Precise Clocks and speculative reads results in the best throughput gain (59% for transactions updating 100 keys).

We also assessed the additional storage overhead introduced by the use of Precise Clocks, which, we recall, requires maintaining additional metadata (a timestamp) for each accessed key. Our measurement shows that for the TPC-C and RUBiS benchmarks (Section 3.8.2), Precise Clocks requires about 9% of extra storage.

3.8.2 Macro benchmarks

Next, we evaluate the performance of STR by implementing two realistic benchmarks, namely TPC-C³ and RUBiS⁴. Unlike the previous synthetic benchmarks, TPC-C and RUBiS specify several seconds of “think time” between consecutive operations issued by a client. Hence, we need to use a much larger client population to saturate the system.

TPC-C. Our TPC-C workload consists of three representative transactions: the *payment* transaction, which has very high local contention and low remote contention; *new-order* transaction, which has low local contention and high remote contention; and *order-status*, a read-only transaction. We consider three workload mixes: 5% new-order, 83% payment and 12% order-status (TPC-C A, Fig. 4.5.(a)); 45% new-order, 43% payment and 12% order-status (TPC-C B, Fig. 4.5.(b)) and 5% new-order, 43% payment and 52% order-status (TPC-C C, Fig. 4.5.(c)). Each server is the master replica of five warehouses. Overall, TPC-C characterizes high-contention and heavy-update workloads.

Figure 4.5 shows that speculative reads bring significant throughput gains, as all three workloads have high degree of local contention. Compared with the baseline protocols (ClockSI-Rep and Ext-Spec), STR achieves significant speedup especially for the TPC-C A

³http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

⁴<http://rubis.ow2.org/>

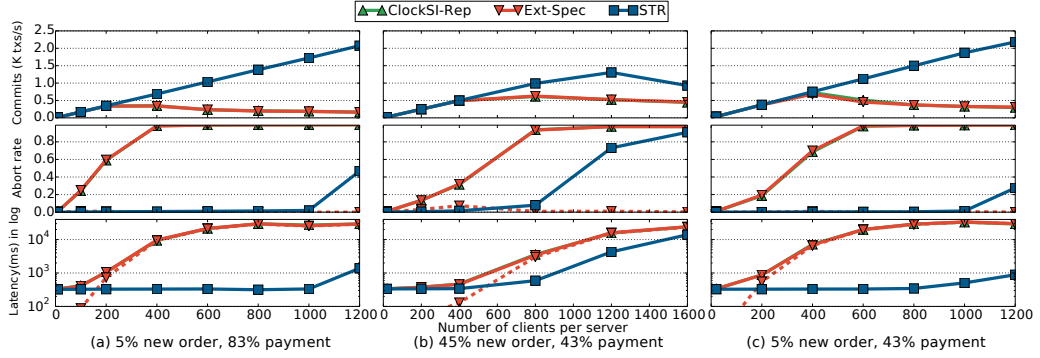


Figure 3.8 – The performance of different protocols for three TPC-C workloads. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

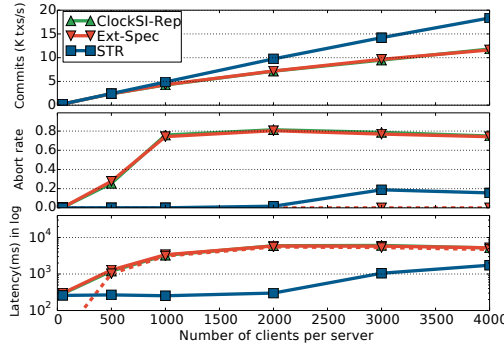


Figure 3.9 – The performance of different protocols for RUBiS. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

($6.13\times$), which has the highest degree of local contention due to having large proportion of payment transaction. For TPC-C B and TPC-C C, STR achieve $2.12\times$ and $3\times$ of speedup respectively. We see that the use of external speculation in this case barely brings any improvement on throughput over ClockSI-Rep. We also observe that the use of external speculation can significantly reduce the (speculative) latency perceived by clients, but only in low load conditions. This can be explained by looking at the abort rate plots, which clearly show that, as load increases, the likelihood that external speculation is successful quickly decreases.

In fact, with larger number of clients (more than 1000 clients per server), the latency of Ext-Spec and ClockSI-Rep is on the order of 5-8 seconds, as a consequence of the high abort rate incurred by these protocols. Conversely, STR still delivers a latency of a few hundred milliseconds.

RUBiS. RUBiS models an online bidding system and encompasses 26 types of transac-

tions, five of which are update transactions. RUBiS is a representative medium-contention, heavy-read workload. RUBiS is designed to run on top of a SQL database, so we performed the following modifications to adapt it to STR's key-value store data model: (i) we horizontally partitioned database tables across nodes, so that each node contains an equal portion of data of each table; (ii) we created a local index for each table shard, so that some insertion operations that require a unique ID can obtain the ID locally (instead of updating a table index shared by all shards by default). We run RUBiS's 15% update default workload and use its default think time (from 2 to 10 seconds for different transactions).

Also with this benchmark (see Figure 3.9) STR achieves remarkable throughput gains and latency reduction. With 4000 clients (level at which we hit the memory limit and were unable to load more clients), STR achieves about 43% higher throughput. The final latency gains of STR over the considered baselines extends up to $10\times$ latency reduction over ClockSI-Rep and Ext-Spec. Also in this case, external speculation is effective in reducing speculative latency only at very low load levels, before losing effectiveness and collapsing to the same performance of ClockSI-Rep.

3.9 Summary

This chapter introduced STR, an innovative protocol that exploits speculative techniques to boost the performance of distributed transactions in geo-replicated settings. STR is based on a novel consistency criterion, which we call SPeculative Snapshot Isolation (SPSI). SPSI extends the familiar SI criterion and shelters programmers from subtle anomalies that can arise when adopting speculative transaction processing techniques. Furthermore, using STR requires no source-code modification, and for both of these reasons it is fully transparent to programmers.

STR builds on recent, highly scalable transactional protocols based on physical clocks (like Clock-SI and Google's Spanner) and extends them with a novel item-based timestamping mechanism (Precise Clocks), speculative reads and a self-tuning mechanism. Via an extensive experimental study, we show that STR can achieve striking gains (up to $11\times$ throughput increase and $10\times$ latency reduction) in workloads characterized by low inter-data center contention, while ensuring robust performance even in adverse settings.

Chapter 4

Sparkle

4.1 Introduction

Nowadays, large-scale online services are faced with a number of challenging requirements. On the one hand, to tame the growing complexity of applications, distributed data storage systems have started embracing strong, transactional, semantics [26]. On the other hand, a number of works [63] have shown that the profitability of large-scale online services hinges on their ability to deliver low latency and high availability — an arduous goal given the sheer volume of traffic and data that modern applications need to cope with.

The above trends have fostered significant interest in the design of high performance transactional platforms capable of ensuring strong consistency and fault-tolerance even when deployed on large scale infrastructures, e.g., [122, 16, 86]. The techniques proposed by recent works in this area extend the classic State-Machine Replication (SMR) approach [109], a long-studied technique for building strongly consistent, fault tolerant systems. In a nutshell, SMR operates according to an *order then execute* approach: replicas rely on a consensus protocol [75] to agree, in a fault-tolerant way, on a total order in which transactions should be executed — which we refer to as *final order*. Transactions are then executed at each replica using a *deterministic concurrency control*, which ensures that their serialization order is equivalent to the final order [62].

Several recent works [122, 16, 86] have focused on addressing what is arguably the key scalability limitation of the classic SMR approach, namely its full replication model, by sharding applications' state across multiple partitions, which are then replicated across a number of machines. This approach, which we call *Partially Replicated State Machine* (PRSM), allows, at least theoretically, for scaling out the volume of data maintained by the platform, as well as the achievable throughput, by increasing the number of data partitions.

However, the partial replication model at the basis of the PRSM approach introduces also a major source of complexity: how to efficiently regulate the execution of transactions that access multiple partitions. While single-partition transactions (SPTs) can be processed at the partitions they access as in classic SMR systems, multi-partition transactions (MPTs) need to access data hosted at remote partitions and, as such, the deterministic concurrency

control also needs to cope with distributed inter-partition conflicts and enforce a transaction serialization order deterministically across replicas.

A simple approach to ensure that, at each partition’s replica, the transactions serialization order is equivalent to the final order is to execute all the transactions in a partition’s replica sequentially [16, 68]. Unfortunately, this solution limits the maximum throughput achievable by any partition to the processing rate of a single thread, failing to fully untap the performance potential of modern multi-core systems.

Other approaches, like Calvin [122], enable multiple threads to process a partition’s transactions concurrently [122, 95], but employ deterministic concurrency control techniques that suffer from two crucial limitations: (i) they rely on a single thread to schedule, in a deterministic way, the execution of all transactions, which inherently limits the scalability of the solution, and (ii) they assume the ability to accurately predict the data items to be accessed by transactions, which is a non-trivial task for complex, real-life applications [6].

This work tackles the above discussed limitations by introducing SPARKLE, a novel distributed deterministic concurrency control that enhances the throughput of state of the art PRSM systems by more than one order of magnitude through the use of *speculative* transaction processing techniques.

Speculation is used in SPARKLE to allow transactions to be processed “out of order”, i.e., to be tentatively executed in a serialization order that may potentially differ from the one established by the replica coordination phase. Thanks to speculative execution, not only can SPARKLE take full advantage of modern multi-core CPUs — by avoiding inherently non-scalable designs that rely on a single thread for executing [16] or scheduling transactions [122]. It also avoids any assumption on the a-priori knowledge of the transactions’ working sets, thus increasing the solution’s generality.

The key challenge one has to cope with when designing speculative systems, like SPARKLE, is to minimize the *cost and frequency of misspeculation*, which, in SPARKLE occur when two conflicting transactions are speculatively executed in a serialization order that contradicts the final order dictated by the replica coordination phase. This problem is particularly exacerbated in PRSM systems, since misspeculations that affect a MPT (e.g., exposing inconsistent data to remote partitions) can only be detected by exchanging information among remote partitions. As such, the latency to confirm the correctness of speculative MPTs is order of magnitudes larger than for the case of SPTs, and can severely hinder throughput.

SPARKLE tackles these challenges via two key, novel, techniques, which represent the main contributions of this work: SPARKLE’s deterministic concurrency control, which combines optimistic techniques with a timestamp-based locking scheme. The former aims to enhance parallelism. The latter increases the chances that the spontaneous serialization order of transactions matches the one established by the replica coordination phase and allows for detecting possible divergences in a timely way, reducing the frequency and cost of misspeculations.

SPARKLE strives to remove the inter-partition confirmation phase of MPTs from the critical path of execution of other transactions via two complementary approaches: i) con-

trolling, in a deterministic way, the final order of transactions, so as to schedule MPTs that access the same set of partitions consecutively; ii) taking advantage of this scheduling technique to establish the correctness of MPTs via a distributed coordination phase, which we call *Speculative Confirmation* (SC). SC is designed to minimize overhead, by exploiting solely information opportunistically piggybacked on remote read messages exchanged by MPTs, and maximize parallelism, by removing the MPT coordination phase from the critical path of transaction processing.

Via an extensive experimental study, based on both synthetic and standard benchmarks, we show that SPARKLE can achieve more than one order of magnitude throughput gains versus state of the art PRSM systems [122, 16], while ensuring robust performance even when faced with challenging workloads characterized by high contention and frequent MPTs.

The remainder of the chapter is organized as follows. Section 4.2 defines the assumed system model and Section 4.3 describes the execution model of generic PRSM systems. Section 4.4, Section 4.5 and Section 4.6 describe the execution of the single-partition transaction, multi-partition transaction and read-only transactions of SPARKLE. Then, the correctness proof SPARKLE is presented in Section 4.7. Then, Section 4.8 discusses related work. Section 4.9 presents the experimental evaluation results. Section 4.10 concludes the paper.

4.2 System and transaction model

System model. We consider the typical system model assumed by PRSM approaches, e.g., [122, 16, 80], in which application data is sharded across a predetermined number of partitions, each of which is replicated over a set of servers, which we refer to as replication group. In the following, we use the terms *partition's replica* and *server*, interchangeably. The architecture illustrated in Fig. 4.1 depicts a possible scenario, in which every partition is replicated in every data center. This deployment provides disaster tolerance, while allowing MPTs to be ordered and executed without requiring communication across data centers [122]. However, our model is generic enough to support scenarios in which certain data partitions may be replicated only in a sub-set of the available data centers.

We assume that servers may crash and that there exists a majority of correct replicas of each partition. While the techniques adopted by existing PRSM systems during the ordering phase are orthogonal to this work, they are normally based on consensus protocols. Therefore, we assume that the synchrony level in the system is sufficient (e.g., eventual synchrony [40]) to allow implementing consensus [44].

Transaction model. SPARKLE provides a basic CRUD transactional interface (create/insert, read, update and delete). Transactions can be aborted and re-executed multiple times before they are committed. We call the various (re-)executions of a transaction *transaction instances*.

Like in any PRSM system, e.g. [122, 16, 80], we assume that the transaction logic is

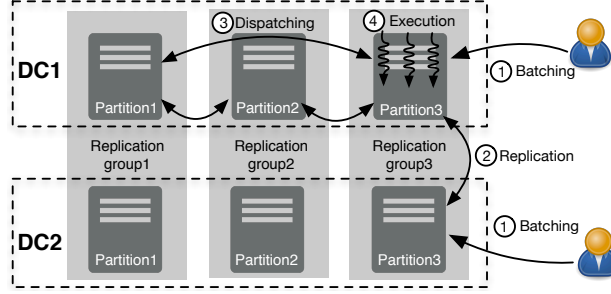


Figure 4.1 – Example of a typical deployment scenario for Sparkle.

deterministic and that, given a transaction and its input parameters, it is possible to identify which data partitions it accesses. This information is exploited to order and execute transactions only at the data partitions they actually access. Such an assumption is typically easy to meet in practice, given that data partitions are normally quite coarse grained. In fact, overestimating the set of partitions accessed by a transaction does not compromise consistency, but only impacts efficiency by causing unnecessary ordering and transaction execution. Unlike other PRSM solutions, e.g., [122], we do not assume any fine-grained information on the individual data items that transactions access.

As mentioned, we distinguish between single and multi partition transactions (SPTs and MPTs, respectively). We refer to the instances of an MPT at the various partitions it accesses as *sub-transactions* or *siblings*, and denote the set of partitions involved by an MPT T using the notation $involved(T)$. Unlike SPTs, which execute independently at each replica, MPTs require, in the general case, communication among siblings, as they may need to access data stored on remote partitions.

When a sub-transaction reads a local key for the first time, it disseminates the corresponding value to its siblings; when a sub-transaction issues a read to a remote key which has not been received yet, it blocks until the value is received. As remote keys do not need to be maintained locally, writes to remote keys are only applied to a private transaction's buffer (to be available if they are later read by the same transaction) that is discarded after the transaction's commit.

4.3 PRSM model

SPARKLE is a deterministic distributed concurrency control designed to accelerate the execution phase of a generic PRSM system, e.g., [122, 80, 16], which operates according to the abstract order-then-execute model defined below.

Ordering phase. The protocol used during the ordering phase is irrelevant for SPARKLE, provided that the final order it establishes ensures the following properties:

1. all the (correct) replicas of the same partition deliver the same *sequence*, $B_1, \dots, B_n$, of transaction batches, where each batch contains the same totally ordered set of (single- or multi-partition) transactions;

2. if an MPT T is delivered in the i -th batch by a partition, then T is delivered in the i -th batch of all the partitions it involves;
3. for any pair of MPTs, say T_1 and T_2 , that access a set of common partitions, say $S = \{P_1, \dots, P_n\}$, T_1 and T_2 are ordered in the same way by all the (correct) servers that replicate any partition in S , i.e., either $\forall P_i \in S T_1 \rightarrow T_2$ or $\forall P_i \in S T_2 \rightarrow T_1$;
4. the relation $<$ is acyclic, where $<$ is defined as follows: $T < T'$ iff any partition delivers T and T' in that order.

The ordering phase establishes a total order on the transactions executing at each partition, whereas the transactions executing at different partitions are only partially ordered. We refer to the order established by this phase as *final order*. We call the transactions ordered before/after a transaction T , T 's *preceding/following transactions*, respectively.

Existing PRSM systems ensure the above properties in different ways. Calvin, for instance, relies on a two-phase scheme (see Fig. 4.1). In the first one, called *replication phase*, servers periodically batch, e.g., for 5-10 msecs, the transactions received from clients and submit the resulting batch to an intra-partition consensus service. This merges the transactions gathered by every replica of a given partition and replicates them in a fault-tolerant manner. In the second phase, called *dispatching phase*, all partitions within the same DC exchange the transactions they delivered during the first phase. This ensures that MPTs are delivered at all the partitions that they need to access. Finally, the transactions gathered during the dispatching phase are deterministically sorted to ensure a consistent final order across all the replicas of every partition.

Execution phase. Once the ordering phase is completed, transactions are executed at all the partitions' replicas they involve. As already mentioned, in order to ensure inter-replica consistency, the execution phase must guarantee that, at all the replicas of a partition, the transactions delivered by the ordering phase are executed according to the same serialization order, i.e., their execution history is equivalent to a common *sequential* history. SPARKLE's concurrency control ensures this guarantee, while allowing transactions to be executed concurrently. As such, it ensures *serializability* semantics [15]. Further, if the protocol used during the ordering phase ensures real-time ordering between transactions (i.e., given two transactions T_1 , T_2 , where T_1 precedes T_2 according to real-time order, T_1 is serialized before T_2 by the ordering phase) then SPARKLE globally guarantees *strict serializability*.

Failure handling. Dealing with failures is relatively simple in PRSM-based systems (including SPARKLE). Since all correct replicas of a partition deliver the same transactions in each batch, MPTs can fetch remote data from any available replica. In order to provide end-to-end fault-tolerance guarantees, in case the replica originally contacted by a client fails (or is suspected to have failed), the client can contact any other replica provided that some complementary mechanism is employed to ensure exactly-once semantics [46, 103].

4.4 Single partition transactions

SPARKLE strives to achieve two seemingly antagonistic goals: maximizing the parallelism of transaction processing and ensuring that transaction execution is equivalent to a sequential execution complying with the final order.

To maximize parallelism, SPARKLE employs a multi-versioned, optimistic concurrency control that imposes no constraints on the processing order of transactions. Denoting with ts the logical timestamp that reflects the final order at a partition, threads select as the next transaction to *start*, the one with the smallest ts value. However, as transactions are processed concurrently, they can be speculatively executed according to a spontaneous, non-deterministic serialization order that contradicts the final order.

To ensure consistency, SPARKLE guarantees that a final committed transaction must have observed a snapshot that includes the versions produced by all its preceding transactions (according to the final order). This property is enforced by letting a transaction T *final commit* only if all its preceding transactions have final committed and if T did not miss any of the updates they produced — which can happen if T reads a data item before any of its preceding transactions writes to it, i.e., a write-after-read conflict. Misspeculations are detected at run-time, leading to the automatic abort and restart of the affected transactions. Transactions are restarted with the same timestamp to ensure deterministic execution across replicas.

In order to enhance efficiency and reduce the chance of misspeculations, SPARKLE incorporates a timestamp-based locking scheme. The timestamp of transactions, i.e. ts , establishes a total order on item versions created by final and speculatively committed transactions, and also defines the visibility of versions: a transaction only reads the latest version produced by speculatively or final committed transactions ordered before it. When writing a data item for the first time, a transaction T locks the data item, which prevents it from being accessed by T 's following transactions before T finishes execution. Also, when writing, T inspects the data item's *read_dependencies*. These register which transactions have already read this data item, and allow T to abort any following transaction that missed T 's updates. Correspondingly, when reading, it is checked if a transaction with a lower timestamp has locked the data item: in the negative case, the reader registers its timestamp in *read_dependencies* to notify future writers; else, the execution of the reader transaction is suspended till the writer completes.

Next, we provide additional details on the management of SPTs. The pseudo-code, along with the used data structures, as shown in Alg. 3 and Alg. 4.

Start. Upon activation, each transaction initializes three main data structures: its *readset*, *writeset* and *abort_flag*. The *readset* and *writeset* are private buffers that store the data items read and updated by the transaction, respectively. *abort_flag* is used to check whether the transaction has been aborted by other transaction.

Execution. During its execution, a transaction T may read and update multiple data items. Before executing any operation, T checks its *abort_flag* to determine if it has been flagged for abort by some preceding transaction. In this case, T is aborted and re-executed (Alg. 3, 2

and 13). Prior to its first update to a data item, T tries to obtain an exclusive lock to it. If the lock is held by a different transaction T' that follows T in the final order (i.e., the ts of T' is larger than that of T), T ejects T' from the lock and sets the *abort_flag* of T' to *true* (Alg. 4, 7-9). Conversely, if the locking transaction T' precedes T , T waits for T' to finish execution (Alg. 4, 5-6). Once T successfully obtains the lock on the data item, it applies the update to its *writeset* (Alg. 3, 3-11).

While executing a read operation, T first attempts to read from its *writeset* and *readset*, to return any version it has previously written to or read from. Else, T redirects its read to the data store and checks the state of the lock guarding the data item it intends to read (Alg. 3, 14-15). Similar to the above locking procedure, T is suspended if the data item is currently being locked by any of its preceding transactions (Alg. 4, 11-12). Otherwise (i.e., the item is not locked, or locked by T 's following transactions), T scans the version list and returns the version with the largest timestamp smaller than its ts . Note that this may not be the version that T would observe, had transactions been executed serially according to the final order, as other transactions preceding T may later produce more recent versions. Thus, T appends its ts to the *read_dependencies* of the data item (Alg. 4, 13-15). This allows aborting T if a write-after-read conflict is later detected.

Suspended transactions. As mentioned, a transaction T is suspended if it tries to read/update a data item that is currently being locked by a preceding transaction. In that case, the thread executing T can start executing the next unprocessed transaction according to the final order, so to enhance parallelism. T will eventually be unblocked when the contending transactions release the lock requested by T . At this point, the thread responsible of T can resume its execution.

Speculative/final commit. After completing its execution, T attempts to speculatively commit, so to make its writes visible to other transactions. For each data item it updated, T inserts a new version in the item's version chain, timestamped and ordered by its ts (Alg. 3, 19 and 20). Meanwhile, T releases the corresponding lock and checks the *read_dependencies* tracked by this data item and aborts any (therein registered) transaction with a larger timestamp (as they missed T 's update on this item) by setting their *abort_flag* to *true* (Alg. 4, 28-32). Additionally, T prunes the identifiers of any final committed transaction still tracked in *read_dependencies*, which are unnecessary as they no longer risk to abort (omitted in the pseudo-code). While applying its updates, if T finds that any of its obtained locks has already been preempted, it aborts itself by removing all inserted versions and releasing any remaining lock (Alg. 4, 25-26). Else, T is considered to be *speculatively-committed* (Alg. 3, 18-22).

Next, T checks if it can final commit, which is only possible if i) all its preceding transactions have already committed and ii) its *abort_flag* is still *false*. As T 's updates have already been applied in the previous step, the final commit logic is very fast, requiring essentially to only increase the counter that tracks the timestamp of the most recent final committed transaction. Recall, in fact, that the *read_dependencies* of final committed transactions are pruned in an opportunistic way by transactions that update those data items in the future (Alg. 3, 24-28).

Algorithm 3: Concurrency Control

Data structures associated to a transaction T
Int TS *▷ an integer denoting T 's final order in its partition.*
Data structures associated to an instance T^* of T
Map<KeyID, Value> RS, WS *▷ read- and write-set of T^* .*
Bool ABORT_FLAG *▷ indicating if T has been aborted by another tx.*
Data structures associated to a thread TH
set<TxInstance> SC_TXS *▷ txs. speculatively committed by TH .*
Data structures associated to each partition
Int NEXT_TX *▷ the timestamp of the next transaction to commit.*

```
1 update(TxInstance  $T^*$ , Key  $k$ , Value  $v$ )
2   if  $T^*$ .ABORT_FLAG == true; localAbort( $T^*$ )
3   if  $k \in T^*$ .WS
4      $T^*$ .WS[ $k$ ] =  $v$ 
5   else
6     result = Lock( $T^*$ ,  $k$ )
7     if result == OK
8        $T^*$ .WS[ $k$ ] =  $v$ 
9       return OK
10    else ▷ If lock request failed (i.e. abort returned)
11      return result
12 read(TxInstance  $T^*$ , Key  $k$ ):
13   if  $T^*$ .ABORT_FLAG == true; localAbort( $T^*$ )
14   if  $k \in T^*$ .WS return  $T^*$ .WS.get( $k$ ) ▷ Check if  $T^*$  wrote to  $k$ .
15   if  $k \notin T^*$ .RS ▷ Check if  $T^*$  already read  $k$ .
16      $T^*$ .RS.set( $k$ , Read( $T^*$ ,  $k$ )) ▷ Store loc. value in read-set
17   return  $T^*$ .RS.get( $k$ )
18 speculativeCommit(Thread  $TH$ , TxInstance  $T^*$ )
19   for  $\{k, v\} \in T^*$ .WS
20     if AddVersion( $T^*$ ,  $k$ ,  $v$ ) == ABORT
21       localAbort( $T^*$ )
22    $TH$ .SC_TXS.add( $T^*$ )
23 finalCommit(Thread  $TH$ )
24   for  $T^* \in TH$ .SC_TXS
25     if  $TH$ .NEXT_TX ==  $T$ .TS ant  $T^*$ .ABORT_FLAG == false
26        $TH$ .SC_TXS.remove( $T^*$ )
27       delete all metadata of  $T^*$ 
28        $TH$ .NEXT_TX++
29 localAbort(TxInstance  $T^*$ )
30   for  $\{k, v\} \in T^*$ .WS
31     UnlockAndRemove( $T^*$ ,  $k$ )
32   remove all local data items from  $T^*$ .WS and  $T^*$ .RS
```

later.

Identifying transactions and snapshots. We define the LAN (*Local Abort Number*) of a transaction instance T^X executing at partition X as the number of times that T aborted and restarted at X due to *local conflicts*. SPARKLE ensures that the only cause of local aborts

Algorithm 4: Backend protocol

Data structures associated to each key entry of kv

TxInstance LOCK_TX $\triangleright$ the transaction holding lock on k .
Set<TxInstance> WAIT_TXS $\triangleright$ txs. blocked when trying to access k .
Set< {TxInstance, Int} > READ_DEPS $\triangleright$ txs. that have spec. read k .
List< {Int, Value} > VERSIONS $\triangleright$ timestamped versions for k .

1 $\triangleright$ The following operations are atomic.
2 **Lock**(TxInstance T^* , Key k)
3 **if** kv[k].LOCK_TX.TS == $\emptyset$
4 kv[k].LOCK_TX = T^* ; **return OK**
5 **elif** kv[k].LOCK_TX.TS < T^* .TS $\triangleright$ Locked by T^* 's proceeding tx..
6 kv[k].WAIT_TXS.add(T^*)
7 **else** $\triangleright$ Locked by T^* 's following tx..
8 kv[k].LOCK_TX.ABORT_FLAG = **true**
9 kv[k].LOCK_TX = T^* ; **return OK**

10 **Read**(TxInstance T^* , Key k)
11 **if** kv[k].LOCK_TX $\neq \emptyset$ **and** kv[k].LOCK_TX.TS < T^* .TS
12 kv[k].WAIT_TXS.add(T^*)
13 **else** $\triangleright$ Not locked, or locked by T^* 's following tx..
14 $\{ts, v\}$ = the largest entry in kv[k].VERSIONS with ts < T^* .TS
15 kv[k].READ_DEPS.add($\{T^*, ts\}$)
16 **return** v

17 **UnlockAndRemove**(TxInstance T^* , Key k)
18 **if** kv[k].LOCK_TX == T^*
19 kv[k].LOCK_TX = $\emptyset$
20 Unblock following transactions of T^* in kv[k].WAIT_TXS
21 **else**
22 remove T^* 's inserted version from kv[k].VERSIONS
23 abort transactions in kv[k].READ_DEP that have read from T^*

24 **AddVersion**(TxInstance T^* , Key k , Value v)
25 **if** kv[k].LOCK_TX != tx
26 **return ABORT**
27 **else**
28 kv[k].LOCK_TX = $\emptyset$
29 kv[k].versions.append($\{T^*, v\}$)
30 Abort txs. in kv[k].READ_DEP that missed T^* 's update
31 Unblock following txs. of T^* in kv[k].WAIT_TXS
32 **return OK**

for a transaction T_i^X (i denoting the final order of T at X) is a conflict with some local transaction that *precedes* T_i^X in the final order. It follows that when the last transaction, say T_i^X , that precedes a MPT, say T_{i+1}^X , at partition X *final* commits, any instance of T_{i+1} at X (currently active or subsequently activated) is guaranteed not to undergo any further local abort and to observe a locally consistent snapshot. We call the LAN of this instance of T_{i+1}^X the *final* LAN of transaction T_{i+1}^X .

LANs allow for tracking aborts due to local conflicts, but not aborts due to remote conflicts. These occur in case a sibling executing at a remote partition Y had previously sent inconsistent data and has to be restarted. We address this issue by associating with a

sub-transaction instance T^X a vector clock, called GAV (*Global Abort Vector*). The GAV of T^X maintains an entry for each partition $Y \in \mathcal{P}$ and it stores: in the entry associated with the local partition X , the LAN of T^X ; for every entry associated with a remote partition $Y \neq X$, the LAN of the transaction instance T^Y , running at partition Y , from which T^X received remote data.

The GAV of a transaction instance T^X serves to identify the snapshot it observed and to establish its consistency. Indeed, if the GAV of T^X contains, in each of its entry, the final LAN of every sibling, then T^X must have observed a consistent global snapshot — as this implies that, at every involved partition, T^X observed a local consistent snapshot. We call such a GAV the *final* GAV for T^X , or simply for T , as all siblings of T share the same final GAV.

Determining the final GAV. SPARKLE determines the final GAV via a *speculative confirmation* (SC) scheme. When T_i^X *speculatively* commits, T_i^X broadcasts to its siblings a SC message containing its GAV, and an *abort set* which contains the identifier and LAN of every local transaction instance aborted by T_i^X .

Partition X can determine the final GAV for T_i^X only if:

- C1. T_{i-1}^X has been final committed.
- C2. For each partition $Y \in \mathcal{P}$, X received an SC message from T_{i-1}^Y tagged with the final GAV of T_{i-1} .

When these two conditions hold the final GAV of T_i is computed as follows: for each involved partition Y , the Y -th entry of T_i 's final GAV is the largest LAN specified for T_i^Y in the abort set of *any* SC message received from Y .

The above mechanism is defined in a recursive way, as the final GAV of T_i can only be computed once T_{i-1} has final committed. This implies, in its turn, that the final GAV of T_{i-1} must also be known - as MPTs are final committed only after their final GAV is known. The base step of this recursion is the first transaction in the batch, noted T_1^X , which is guaranteed to never abort. As such, all the entries of T_1 's final GAV are necessarily equal to zero and T_1^X can be used as an initial “anchor” to bootstrap the SC scheme: as T_1^X speculatively commits, it can be immediately final committed; when X receives the SC messages from all the siblings of T_1 , since these SC messages are tagged with T_1 's final GAV, X can determine the final GAV of T_2 , and so forth.

Figure 4.2 exemplifies a scenario in which T_1^X aborts the first instance of T_2^X due to a local conflict on data item k and notifies partition Y via an SC message (the SC message is sent by T_1^X upon final commit since, being the first transaction of the batch, it cannot abort and omits the speculative commit phase). Upon reception of T_1^X 's SC message, Y establishes the final GAV for T_2 , i.e., $[1,0]$, and T_2^Y restarts with that GAV. When this instance of T_2^Y speculatively commits, it emits an SC message that is used at partition X to establish the final GAV for T_3 . After speculatively committing, the instance of T_2^Y with GAV=[1,0] can be final committed, since its GAV coincides with T_2 's final GAV.

Algorithm 5: MPT execution at partition X

Data structures associated with an MPT T :

Array of $\text{int}[\text{numPartitions}]$ GAV $\triangleright$ Current known GAV of T .
Array of $\text{GAV}[\text{numPartitions}]$ SCMSG_GAV $\triangleright$ GAVs of the last SC
 $\triangleright \dots$ msg received from each partition

Data structures associated with every instance T^* of an MPT T :

map $\langle \text{keyID}, \text{value} \rangle$ RS, WS $\triangleright$ read- and write-set of T^* .
map $\langle \text{TID}, \text{int} \rangle$ ABORTSET $\triangleright$ Map storing the LANs of any
 $\triangleright \dots$ local tx. instance aborted by T^* .

```
1 read(TxInstance  $T^*$ , Key  $k$ ):
2   if  $k \in T^*.WS$  return  $T^*.WS.get(k)$   $\triangleright$  Check if  $T^*$  wrote to  $k$ .
3   if  $k \notin T^*.RS$   $\triangleright$  Check if  $T^*$  already read  $k$ .
4     if  $k$  is local
5        $T^*.RS.set(k, \text{localRead}())$   $\triangleright$  Store loc. value in read-set
6       send  $\langle k, T^*.RS.get(k), T.GAV[X] \rangle$  to  $T$ 's siblings
7     else  $\triangleright$  Remote key, wait for its value from corr. partition.
8       wait receive  $\langle k, v, \text{LAN} \rangle$  s.t.  $\text{LAN} \geq T.GAV[k.locPart()]$ 
9        $T^*.RS.set(k, v)$   $\triangleright$  Store remote value in read-set.
10      if  $\text{LAN} > T.GAV[k.locPart()]$   $\triangleright$  Remote sibling has aborted.
11        remoteAbort( $T$ ,  $k.locPart()$ ,  $\text{LAN}$ ,  $\langle k, v \rangle$ )
12      return  $T.RS.get(k)$ 
13 localAbort(TxInstance  $T^*$ , TxInstance  $T'^*$ )  $\triangleright$   $T^*$  aborts  $T'^*$ .
14    $T'.GAV[X]++$ 
15    $T^*.ABORTSET.add(\langle T', T'[X] \rangle)$ 
16   restart a new instance of  $T'$ , cloning its RS from  $T'^*$ ...
17   ...and removing any local key from it
18 remoteAbort(Tx  $T$ , Partition  $Y$ , Int  $lan$ ,  $\langle \text{Key } k, \text{Value } v \rangle$ ):
19    $T.GAV[Y]=lan$   $\triangleright$  Update LAN of tx  $T$  at partition  $Y$ .
20   if  $\exists$  an active instance  $T^*$ 
21     restart a new instance of  $T'$ , cloning its RS from  $T'^*$ ...
22     ...and removing from it any key received from  $Y$  except for  $k$ 
23 tryCommit(TxInstance  $T^*$ ):
24   speculativeCommit( $T^*$ )  $\triangleright$  Spec. apply  $T$ 's writes to local keys.
25   send: $\langle \text{SC}, T^*.ABORTSET, T, T.GAV \rangle$  to  $T$ 's siblings
26   waitFinalGAV( $T$ )
27    $\triangleright$  This tx. instance has the final GAV if it did not abort so far.
28   finalCommit( $T^*$ )
29 waitFinalGAV(Tx  $T$ ):
30    $T' \leftarrow T.getPrecedingTx()$ 
31   wait until  $T'$  has final committed  $\triangleright$  Cond. C1.
32    $\forall Y \in T.remotePartitions()$ 
33     wait until  $T'.GAV = T'.SCMSG\_GAV.get(Y)$   $\triangleright$  Cond. C2.
34 upon receiving  $\langle \text{SC}, \text{ABORTSET}, T, \text{GAV}_{msg} \rangle$  from partition  $Y$ 
35    $T.SCMSG\_GAV[Y] \leftarrow \text{GAV}_{msg}$   $\triangleright$  Store GAV of last SC from  $Y$ .
36   for each  $\langle T', \text{LAN} \rangle \in \text{ABORTSET}$   $\triangleright$  For each tx aborted by  $T$ .
37     if  $\text{LAN} > T'.GAV[Y]$ :  $\triangleright$  Skip aborted txs we already know of
38       remoteAbort( $T'$ ,  $Y$ ,  $\text{LAN}$ ,  $\langle \perp, \perp \rangle$ )
```

Pseudo-code. Alg. 5 shows the pseudo-code for managing a homogeneous batch of MPTs at partition X . To simplify presentation we assume FIFO-ordered channels. We omit discussing write operations, as these are managed as in SPTs.

Additional Data structures. For the management of MPT, each partition maintains the following data structures (in addition to the ones presented in Alg. 3) for each MPT T : i) GAV: T 's currently known Global Abort Vector; ii) SCMSG_GAV: a map that stores, for every sibling sub-transaction T^Y , with $Y \in \mathcal{P}$, the GAV of the most recent SC message received at X from any instance of T^Y . Additionally, for each MPT instance T^* the following data-structures are used: i) RS/WS, which store the transaction instance's read-set and write-set, respectively; ii) the ABORTSET, a map that stores the largest LAN of any local transaction so far aborted by T^* . At any time, at partition X for an MPT T there is at most one active instance T^* that is associated with the current GAV of T at X : upon its activation, T^* is associated with the currently known GAV for T and whenever the GAV of T changes, T^* is aborted and a new instance is restarted associated with the new GAV.

Read logic. When T^* reads a key, it first checks if it previously wrote to or read it. In these two cases the value stored in T^* 's write-/read-set is returned, respectively. Else, i.e., first access to a key, if the key is local, T^* fetches its value from the local storage and broadcasts it to its siblings. This message is tagged with the transaction instance's LAN, which coincides with the local entry of the GAV of T . If the key is hosted at a remote partition, say Y , T^* waits for the key's value from Y and checks if the received LAN is larger than the Y -th entry of the GAV of T . In this case, T^* had previously received stale data from a sibling running at Y , which later aborted. Thus, T^* is aborted and restarted. If the LAN of the value received from Y coincides with the Y -th entry of the GAV of T at X , instead, the value is added to the read-set and is returned.

Handling aborts. When T^* aborts a local transaction instance T'^* (Alg. 5, 13), the local entry of the GAV of T' , i.e., its LAN, is increased. Next, T'^* and its LAN are added to the ABORTSET of T^* and a new instance of T' is activated. The read-set of this new instance is initialized with a clone of the read-set of its previous "incarnation", purged of any *local* data. This ensures that the new transaction instance retains any remote data received so far, avoiding re-fetching it remotely.

When X learns about the abort at a remote partition Y of an instance of transaction T with a given LAN (Alg. 5, 18), X accordingly updates the Y -th entry of T 's GAV and aborts any local instance of T . The restarted instance of T inherits, in this case, the read-set of its previous incarnation purged of any data previously received from Y , with one exception: if the remote abort is detected when receiving a remote value (Alg. 5, 11), this value belongs to a fresh remote snapshot at Y and can be retained in the read-set.

Commit logic. When T^* completes its execution (Alg. 5, 23), it speculatively commits and broadcasts SC messages to all partitions in $\mathcal{P}$. The SC messages disseminate the ABORTSET of T^* , informing remote partitions about the local transaction instances aborted by T^* at X . Upon reception of a SC message from partition Y (Alg. 5, 34): i) the SCMSG_GAV associated with partition Y is updated with the GAV_{msg} of the transaction instance that sent the SC message; ii) for any transaction T' included in the SC message's ABORTSET, if the

corresponding LAN is larger than the Y -th entry of the GAV of T' , it means that X detected a new remote abort at Y . Thus, the `remoteAbort()` method is called.

For T_i^* to be final committed, its GAV must coincide with the final GAV for T_i . This is determined (Alg. 5, 29) after waiting for transaction T_{i-1} (i.e., the transaction immediately preceding T_i) to have final committed (Cond. C1), which implies that the final GAV of T_{i-1} is known. So, to determine the final GAV of T_i , it suffices to wait for the reception, from every remote partition, of an SC message tagged with the final GAV of T_{i-1} (Cond. C2, Alg. 5, 33). After this moment, in fact, no instance of T_i can any longer be aborted at any partition. Thus, if a speculatively committed transaction instance T^* returns from `waitFinalGAV()` without being aborted, it means that none of the instances of T 's preceding transactions have invalidated T^* global snapshot. In this case, T^* 's GAV coincides with the final GAV for T and T^* can be final committed.

Implicit dissemination of SC messages. To reduce the overhead of the SC mechanism, SPARKLE exploits a key optimization, not reported in the pseudo-code: instead of sending *ad hoc* SC messages, these are piggybacked on the messages used by MPTs to disseminate the results of read operations.

Dealing with heterogeneous MPTs The correctness of the SC mechanism presented above hinges on the assumption that the batch is composed solely by MPTs accessing the same partitions. This ensures that the first MPT of the batch never undergoes aborts. Clearly, this property no longer holds if batches are composed by mixes of SPT and MPTs involving heterogeneous sets of partitions. In fact, in the general case, a (multi-partition) transaction can undergo an unknown number of aborts if it is preceded even just by a single transaction and is executed concurrently with it.

To cope with the above problem, SPARKLE only allows executing an MPT, if all its preceding uncommitted transactions are either read-only transactions (ROT) or MPTs that access the same set of partitions (which we refer to as homogeneous MPTs). While trivially ensuring the correctness of the SC mechanism, if naively employed, this technique can also hinder parallelism. For instance, if three homogeneous MPTs are interleaved by two SPTs, then these three MPTs have to be executed sequentially.

SPARKLE tackles this issue via a *scheduling* mechanism, which operates as follows. First, upon delivery of a transaction batch, at the end of the ordering phase, each partition deterministically reorders the MPTs in the batch by grouping them according to the set of partitions they access¹. The resulting final order is composed by a sequence $G_1, \dots, G_n$ of transaction groups, where each group G_i contains the transactions that access the same set of partitions. Next, each partition deterministically re-orders its SPTs and ROTs, serializing them in between each pair of consecutive MPT groups, with the goal of “spacing them out”. The number of SPTs and ROTs serialized in between two groups are calculated in a deterministic fashion, with the goal of ensuring that each group interval is filled with an even number of SPTs/ROT. Note that since MPTs can not be executed while there are preceding active SPTs (because the proposed scheduling mechanism aims at ensuring that

¹The current prototype uses a single thread to re-order transactions, as in all tested workloads the scheduling thread was never the bottleneck.

the first MPT in a transaction group never aborts), in between two groups we always place SPTs before ROTs, to space out SPTs and the following MPT group. Note that since only transactions of the same batch can be reordered, and that these are necessarily concurrent, scheduling does not compromise real-time order.

4.6 Read-only Transactions

Since ROTs do not alter the state of the data store, they can be executed at a single partition's replica and serialized in an arbitrary order, provided that they observe a consistent snapshot of the data store. To minimize overheads, in SPARKLE ROTs are executed concurrently with the remaining update transactions, but in a non-speculative fashion, i.e., by assigning them a timestamp associated with a final committed transaction. This allows sparing ROTs from the overheads associated with registering themselves among the read dependencies of the keys they read — which becomes unnecessary since, being serialized after a final committed update transaction, ROTs are guaranteed to observe a stable snapshot.

While single partition ROTs can be freely assigned any serialization order by their local partition, this is not the case for read-only MPTs. In this case, it is necessary to ensure that a read-only MPT is assigned the same serialization order at all the partitions it involves. SPARKLE tackles this problem through a deterministic scheduling policy, which serializes every read-only MPT before any other transaction of their batch — this ensures the stability of the snapshot over which they are executed and allows them to be executed in a non-speculative fashion, analogously to read-only SPTs.

4.7 Correctness proof

In this section, we discuss the correctness of SPARKLE. Specifically, we intend to show that SPARKLE's concurrency control ensures that, at all replicas of any partition, the history of committed transactions is equivalent to the same sequential history, whose order we denote with $<_S$. This serialization order determined by $<_S$ is obtained by applying the deterministic transaction grouping and scheduling rules discussed in Section 4.7.2 to the final order determined by the dispatching phase.

We start by discussing the case of batches consisting solely of SPTs in Section 4.7.1. Next, Section 4.7.2 discusses the case of batches composed by a single group of homogeneous MPTs. Then, Section 4.7.3 analyzes the case of generic batches composed by SPTs and heterogeneous MPTs. Finally, we prove that SPARKLE enforces real-time order in Section 4.7.4.

4.7.1 Single-partition transactions

Let us first examine the case of batches containing only SPTs. We show that any *final committed* SPT T_i , having serialization order i in the batch observes a snapshot that reflects the updates produced by every transaction serialized before T_i , i.e., any read issued by T_i

on a key K must return the version created by the transaction T_h , whose serialization order, h , is the largest among all transactions that precede or equal i and that write to K .

By contradiction, let us assume that in an execution in which T_i final commits (i.e., T_i does not get aborted during its execution), T_i reads a key K and fetches a version created by transaction T_f , where $f <_S h$ (i.e., T_i misses the version of K created by T_h and observes an earlier version created by T_f). Depending on whether T_i has updated K before reading, there are two possible scenarios:

- T_i updated K before reading: in this case, it is not possible for T_i to read T_f 's version, as in SPARKLE transactions first attempt to read from their own write-set before trying to read from other transactions (Alg 3, 14). In fact, in such a case $i = h$, as T_i will read its own updates. This leads to a contradiction.
- T_i did not update K before reading: there are two sub-cases in this scenario: 1) T_i reads K before T_h updates it, and 2) T_i reads K after T_h updates it.

In the former case, when T_h commits (either speculatively or finally), it will abort T_i as T_i missed its update (Alg 4, 30). Recall, in fact, that since T_i is serialized after T_h , T_i can only final commit after T_h final commits (Alg 3, 25), and T_h will abort T_i before T_h final commits because T_i misses T_h 's update. Thus, we obtain a contradiction.

In the latter case, T_h may or may not have final committed yet. If T_h has final committed, T_i is forced to read the version of K created by T_h (Alg 4, 14), since this is more recent than the one created by T_f . Otherwise, since T_h has updated K but has not final committed, it must be still holding lock on K . Thus, T_i will be blocked until T_h final commits, and then T_i will eventually read T_h 's update on K (Alg 4, 11). In both scenarios we obtain a contradiction, proving the claim.

4.7.2 Multi-partition transactions of homogeneous batches.

Let us now consider the scenario of batches containing homogeneous MPTs that involve the same set of partitions. We intend to show that any MPT T_i that final commits must necessarily observe a globally consistent snapshot, i.e., T_i observes the effects of every transaction T_h that, according to $<_S$, is serialized before T_i .

As described in Section 4.7.2, SPARKLE determines the final GAV of a transaction T_i^X by ensuring conditions C1 and C2, which are recalled as below:

- **C1)** T_{i-1}^X has been final committed.
- **C2)** for each partition $Y \in \mathcal{P}$ (i.e., the set of partitions involved by T_{i-1}), X received an SC message from T_{i-1}^Y tagged with the final GAV of T_{i-1} .

Also, T_i^X can only be final committed if its GAV (which intuitively identifies the global snapshot observed by T_i^X) matches its final GAV.

We structure our proof in two steps. In step 1, we show that, under the assumption that T_i^X can correctly identify its final GAV, it follows that T_i^X is final committed only if it observed a globally-consistent snapshot. In Step 2, we show that, by enforcing C1 and C2, the final GAV of a transaction is correctly identified.

Step 1. As described above, let us assume to have identified the final GAV of a transaction T_i^X as g . T_i^X can only be final committed if its GAV is the same as its (calculated) final GAV g . Recall that the GAV of T_i^X maintains an entry for each partition $Y \in \mathcal{P}$ and it stores: in the entry associated with the local partition X , the LAN of T^X ; for each entry associated with a remote partition $Y \neq X$, the LAN of the transaction instance T_i^Y , running at partition Y , from which T_i^X received remote data.

If the GAV of T_i^X is equivalent to its final GAV (which is necessary to final commit T_i^X), it means that at any partition $y \in \mathcal{P}$, T_i^Y can no longer abort due to a local conflict (else its GAV would change). Namely, the data that T_i^Y has disseminated to its siblings is necessarily locally consistent, i.e., T_i^Y cannot have missed any preceding transaction at its local partition Y and, thus, must be tagged with T_i^Y 's final LAN. Essentially, it means that T_i^X must have observed locally-consistent data of all partitions in $\mathcal{P}$, and, as such, T_i^X must have observed a globally-consistent snapshot.

Step 2. This property is proven by induction. Let us first analyze the case of T_1^X , namely the first transaction in the batch. Recall that transactions of different batches are never concurrently executed (see Section 4.5). As such, T_1^X and its sibling can never abort. Therefore, the final LAN of T_1^X , as well as all its siblings, must be 0. In other words, all sub-transactions of T_1 can always be directly final committed after their execution.

Next, assume that SPARKLE determines the final GAV correctly for transactions $T_2, \dots, T_{i-1}$, we prove that the final GAV of T_i is computed correctly, i.e., once SPARKLE determines that the GAV of T_i^X is final, this will no longer change. Note, though, if conditions C1 and C2 hold for T_i^X , then X must have received a SC message tagged with the final GAV for all the transactions that precede T_i according to $<_S$ (namely, $T_1, \dots, T_{i-1}$), from every partition involved by these transactions (which, recall, are assumed to be homogeneous). Since these messages are tagged with the corresponding final GAVs, it follows that none of these transactions can experience a local abort any longer (else their GAVs would not be final). By the FIFO property on channels, it is also ensured that if partition X received a SC message tagged with the final GAV from T_h^Y (where $h < i$), then partition X must have necessarily received all the SC messages that T_h^Y can ever generate (since the SC message tagged with the final GAV for T_h^Y is the last SC message that partition Y generates for T_h).

Overall, once conditions C1 and C2 hold for T_i^Y , it is impossible for any transaction T_h^Y , where $h < i$, to cause further local aborts of T_i^Y . Additionally, all SC messages that can ever be generated by any transaction T_h^Y , where $h < i$, must have already been delivered at partition X .

Thus, it is impossible for the GAV of T_h to change.

4.7.3 Heterogeneous batches.

Recall that SPARKLE deterministically reorders, at each partition, each delivered transaction batch into a sequence of batches composed by homogeneous MPTs. Since the logic for reordering the heterogeneous batches is deterministic and since two heterogeneous batches are never executed concurrently, it follows that, at all replicas, the same set of MPTs are executed in the same serialization order.

It remains, however, to prove that the correctness of SPTs and ROTs is not compromised by the scheduling policy adopted by Sparkle to cope with heterogeneous MPT batches.

First, as described in Section 4.6, a read-only transactions (ROT) is always executed on a stable snapshot, namely a snapshot containing all transactions up to its previous batch. Therefore, the execution of ROTs is not affected due to re-ordering or concurrent execution with other transactions.

Secondly, the correctness of SPTs is not affected either, as we have proven that SPTs always observe a local snapshot that contains all preceding transactions, no matter the preceding transactions are SPTs or MPTs. Thus, the above correctness argument can be directly applied to re-ordered SPTs.

Finally, as described in Section 4.7.2, each MPT group, G , can only be concurrently executed with ROTs and with SPTs ordered after G . As SPARKLE's concurrency control ensures that no transaction can ever be aborted by its following transaction or ROTs, the execution of an MPT group will not be interfered by transactions ordered after it. Also, recall that SPARKLE only allows executing an MPT, if all its preceding uncommitted transactions are either ROTs or homogeneous MPTs. Thus, the execution of each MPT in a group can only be affected, i.e. aborted, by some preceding transactions belonging to the same group. As such, an MPT group can essentially be regarded as a homogenous batch.

4.7.4 Real-time ordering

As described in Section 4.3, if the ordering protocol ensures real-time ordering between transactions, SPARKLE guarantees strict serializability. Nevertheless, SPARKLE also employs a deterministic scheduling mechanism that strives to reorder transactions to enhance transaction processing throughput. So, in the following, we prove that the SPARKLE still guarantees real-time ordering guarantee in spite of this scheduling mechanism.

Assumption by contradiction that there are two transactions T_1 and T_2 , such that T_2 starts after T_1 has finished, but T_2 is ordered before T_1 in $<_S$, where $<_S$ denotes the serialization order determined by SPARKLE. This means that either T_2 is ordered in a consensus instance that precedes the one in which T_1 was ordered, or that T_1 and T_2 were inserted in the same batch and the scheduling logic of SPARKLE reordered T_2 before T_1 .

Since T_2 starts after T_1 is completed, i.e., after T_1 's results have been returned to the client that submitted T_1 , at the time in which T_2 is submitted, the consensus instance/ordering phase for T_1 must have necessarily ended. As such, T_2 must necessarily be inserted in a consensus instance that follows the one in which T_1 was included. This leads to a

contradiction of the assumption, hence the claims follows.

4.8 Related Work

This section analyzes the contributions of SPARKLE with respect to existing research, which extends the review presented in Chapter 2. We structure the analysis in three parts: first, we discuss relations with existing transactional stores; next, a comparison with existing partially-replicated state machines is presented; finally, we compare SPARKLE with other systems supporting deterministic execution.

Transactional stores. A large body of works has investigated how to build consistent, yet scalable, transactional stores. As already discussed in Section 2.3, existing systems can be coarsely classified based on whether they adopt the *deferred update replication* (DUR) [66] or the *state-machine replication* (SMR)[75] approaches. In DUR-based systems, e.g. [26, 82, 73, 127], transactions are first locally executed at a single replica and then globally verified, via an agreement protocol based on consensus [66] and/or Two Phase Commit [50]. Speculation has been employed in DUR-based solutions either at the level of the local concurrency control scheme (e.g., exposing pre-committed state rather than blocking processing [82, 99]) or at the consensus level (e.g., skipping communication steps in absence of conflicts among concurrently submitted transactions [98, 73, 127]).

Unlike DUR-systems, with SMR, e.g., [122, 38], replicas first agree on the serialization order of transactions, using consensus-based coordination schemes, and then execute them using a deterministic concurrency control. Section 2.3 has discussed the complementary pros and cons of the DUR and SMR approaches, which are fit for different workloads. The focus of this work is on SMR-based systems, which excel in contention-prone workloads, whereas DUR systems can suffer from lock-convoying and high abort rates [124].

Partially-replicated state machines. The PRSM approach [86, 16, 122, 80] extends the classic SMR scheme to support a more scalable partial replication model. Existing PRSM systems rely on diverse techniques to implement a deterministic concurrency control.

Some approaches eliminate the possibility of non-deterministic execution [16, 68] by allowing the execution of only a single thread per partition. This approach spares from the use (and cost) of any concurrency control, but it also inherently limits the maximum throughput achievable by any partition to the processing rate of a single thread. Some works [77, 68] have argued that this limitation can be circumvented by using a larger number of smaller partitions, delegating each partition to a different thread of the same machine. However, this approach can increase significantly the frequency of MPTs, since, when using smaller partitions, it is more likely for transactions to access data scattered over multiple partitions. Accesses to multiple partitions, even if maintained by the same machine, impose severe synchronization overheads among the different instances of the same MPT running at different partitions, which need to block until the corresponding remote instances execute and disseminate data to other partitions (see Section 4.9).

Other systems, e.g., [122, 95], conversely, allow for concurrent execution of transac-

tions and enforce deterministic execution by relying on a single thread to acquire, according to the final order, the locks required by transactions, before executing them. Unfortunately, as we show in Section 4.9, in typical OLTP workloads dominated by short running transactions, the scheduler thread quickly becomes a bottleneck as the degree of parallelism increases. Further, in order to acquire all the locks needed by a transaction before its execution, these solutions require mechanisms for predicting the transaction’s data access pattern — a non-trivial problem in complex real-life applications [6]. The solutions proposed in the literature to cope with this issue are quite unsatisfactory: existing techniques either require programmers to conservatively over-estimate the transaction’s working set [95] (e.g., at the granularity of transaction tables, even though transactions need to access just a few tuples), or they estimate it by simulating the transactions execution, and then abort them if the working set’s estimation turns out to be inaccurate during (real) execution. The former approach can severely hinder parallelism. The latter can impair performance in workloads that contain even a small fraction of, so called, *dependent transactions* [122], i.e., whose set of accessed data items is influenced by the snapshot they observe.

SPARKLE tackles these limitations by combining speculative transaction processing techniques — which exploit out of order processing techniques to enhance parallelism with no a priori knowledge of transactions’ working sets — and scheduling mechanisms — which redefine, in a deterministic way, the serialization order of transactions established by the ordering phase to minimize the cost of detecting misspeculations.

Deterministic execution. The problem of designing efficient deterministic concurrency controls has also been studied for classical SMR systems adopting a full replication model [56, 62, 90, 99, 69]. Some of these works, e.g., [62, 99, 69], employ speculative transaction processing techniques, as in SPARKLE. Though, unlike these solutions, SPARKLE targets a partial replication model, which, as already discussed, raises additional challenges related to the processing of MPTs. Analogously to SPARKLE, Eve [69] incorporates scheduling techniques to maximize the efficiency of speculation. However, unlike SPARKLE, Eve’s scheduling mechanism requires a priori knowledge on transactions’ conflict patterns.

The deterministic concurrency control of SPARKLE has relations also with the works on deterministic execution of multi-threaded applications, typically aimed at debugging and testing [10, 13, 32, 33, 106]. These mechanisms intercept *all* non-deterministic events affecting threads’ execution (to be later replayed). In the context of SMR/PRSM systems, though, a deterministic concurrency control scheme has to tackle a different problem: ensuring that the serialization order of transactions is equivalent to the one established by the replica coordination phase.

4.9 Evaluation

This section is devoted to experimentally evaluate SPARKLE, by comparing it with two state of the art PRSM systems, namely S-SMR [16] and Calvin [122].

This study aims to answer the following main questions:

- How effective is SPARKLE in exploiting the parallelism potential of large multi-core architectures? (Section 4.9.3)
- What throughput gains does SPARKLE achieve in a typical deployment (like the one illustrated in Fig. 4.1), where data is fully replicated across data centers and sharded across the nodes of a medium size? (Section 4.9.4)
- How efficient is the management of MPTs? And do the performance gains brought about by processing MPTs in a speculative fashion justify the additional complexity that speculation introduce? (Section 4.9.4)
- How does SPARKLE’s performance scale when deployed over large scale clusters (up to 40 servers)?(Section 4.9.4)

4.9.1 Implementation and setup

We implemented SPARKLE and S-SMR, based on Calvin’s code base [120]. The original code base uses STL’s *unordered_map* as the in-memory back-end to store data, which we found out to become the system’s bottleneck at high thread counts. Therefore, in our implementation, we replaced it with *concurrent_hash_map* from Intel’s TBB library [64]. The repository containing the code used in this study is publicly accessible [78].

To quantify the scalability of SPARKLE on large multi-core architectures (Section 4.9.3) we use a machine equipped with two Intel Xeon E5-2648L v4 CPUs, consisting in total of 28 cores (56 hardware threads). All other experiments were conducted on the Grid’5000 cluster [52] using 8 *genepi* machines, each of which has two 4-cores Intel Xeon E5420 QC CPUs. Unless otherwise noted, all protocols use three cores for auxiliary tasks needed for the evaluation (e.g., network communication and workload generation); other than that, Calvin dedicates one core for serializing lock requests and four other cores to execute transactions, SPARKLE uses five cores to execute transactions, and S-SMR only uses one core to execute transactions.

The presented results are the average of three runs. We also report the results’ standard deviation, but since the differences in performance across different runs are usually within 5%, in various plots, standard deviations are not visible.

As in prior work [122], we emulate the ordering phase by injecting a 200ms delay and use 10 milliseconds batch time. To avoid overloading the system, we adjust the arrival rate to be 10%-20% larger than the maximum sustainable throughput (determined via a preliminary test). Therefore, batch sizes vary depending on the workload, ranging from 10s to 100s of transactions. Omitting the ordering phase allows for focusing the evaluation on scenarios where throughput is bottlenecked by the execution phase. This is typically the case when one employs batching techniques [45, 105] to increase the maximum throughput sustainable by the ordering phase. As for the choice of the delay of the ordering phase, we argue that using smaller values would reduce user perceived latency but it would not affect the throughput of the considered solutions.

4.9.2 Benchmarks

Synthetic benchmark. In this benchmark each partition contains one million keys, split in two sets, which we call “index” and “normal” keys, respectively. All transactions start by reading and updating five index keys selected uniformly at random. If the transaction is a ‘dependent transaction’, it reads five additional normal keys, whose identity is determined by the values read from the five index keys (i.e., the read- and write-set of dependent transactions can only be determined during execution). Else, if the transaction is non-dependent, it reads and updates five randomly selected normal keys. If a transaction accesses more than one partition, it divides equally its accesses among its involved partitions. For instance, if a dependent transaction accesses two partitions, then it accesses three index keys and three normal keys of a partition and the other two index and two normal key from the second partition. Multi-partition transactions, unless otherwise noted, always access two partitions.

We shape the workloads generated via this synthetic benchmark by varying three parameters: contention level (low and medium contention), percentage of dependent transactions (0%, 1%, 10%, 50% and 100%) and percentage of distributed transactions (0%, 1%, 10% and 50%). We control contention by varying the number of index keys of each partition (using the remaining keys as normal keys): in the low contention scenario, partitions use 50000 index keys; 1000 index keys per partition are used, instead, for the medium contention case.

TPC-C. The TPC-C benchmark [24] has five transaction profiles: NewOrder, Payment, OrderStatus, StockLevel and Delivery. NewOrder and Payment are update transactions that access a warehouse hosted on a remote partition with probability 10% and 15%, respectively. OrderStatus and StockLevel transactions are read-only, single-partition transactions (SPTs). Delivery transactions are update SPTs. Finally, NewOrder and Payment are independent transactions, while the other three are dependent transactions (i.e., their working set depends on the database state and cannot be predicted statically).

We consider three different transaction mixes, containing 10%, 50% and 90% update transactions. All transaction mixes always contain only 4% of Delivery transactions, while the other two update and read-only transactions evenly share the rest of the proportion. Except in Section 4.9.3, we populate 12 warehouses per data partition in all TPC-C experiments.

4.9.3 Single node deployment

Before testing SPARKLE in distributed settings, we focus on single node performance, evaluating its scalability on a large multi-core machine equipped with 56 hardware threads.

When testing SPARKLE and Calvin we deploy a single data partition, and increase the total number of worker threads up to 50, dedicating 6 threads to workload generation. Conversely, since S-SMR can only utilize one worker thread per data partition, the only way to let it exploit the parallelism of the underlying architecture is by varying the number of partitions, which we increase up to 50 (using the same amount of data). We use the 90% update TPC-C workload, and adjust the contention level by varying the number of

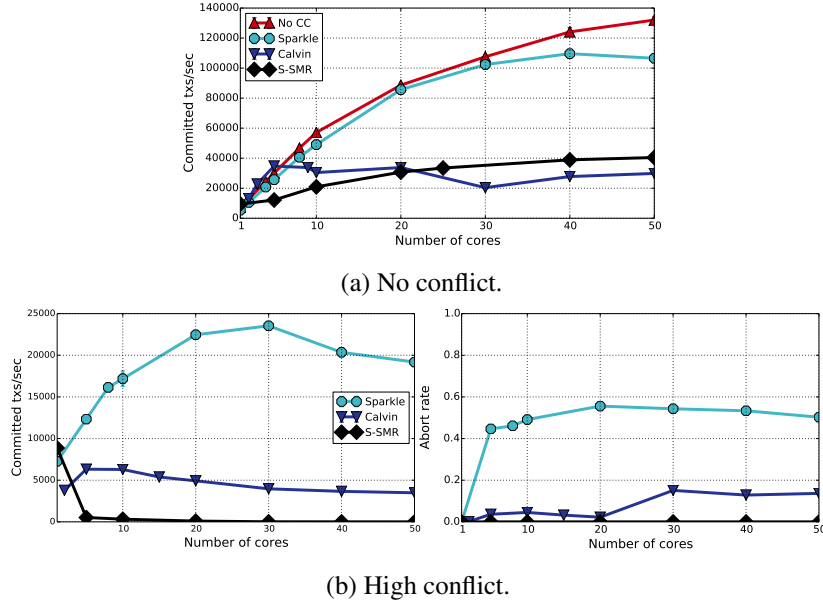


Figure 4.3 – Single node deployment, TPC-C 90% update workload.

warehouses to generate two extreme scenarios: a very high conflict workload, in which only a single warehouse is populated, and a no conflict workload, in which we populate a large number of warehouses (200) and alter the workload to generate no conflicts (by having concurrent requests access disjoint warehouses). For the no-conflict workload, we consider an additional *NoCC* baseline, i.e., a protocol which implements no concurrency control whatsoever. This represents an ideal baseline that allows us to better understand the scalability limit and overhead of each protocol.

Fig. 4.3a reports the performance of the considered protocols using the no-conflict workload. As we can see, SPARKLE has almost identical throughput to the ideal *NoCC* baseline up to 30 threads, incurring less than 20% overhead with 40 and 50 threads. These results clearly highlight the efficiency and practicality of SPARKLE’s concurrency control. Conversely, Calvin’s throughput only scales up to five threads (one locker thread and four worker threads). At higher thread counts, the scheduling thread turns into the system’s bottleneck, severely hindering its scalability. Last but not least, we can see that S-SMR achieve good scalability and outperforms Calvin when using more than 25 threads. However, S-SMR achieves $2.6\times$ lower throughput than SPARKLE at 50 threads. This is due to the fact that, in this workload, approximately 10% of transactions access a remote warehouse, which with S-SMR may be stored on a different partition (unlike SPARKLE and Calvin, which do not need to use multiple partitions per node to enable parallelism). Despite in this test, communication between the sub-transactions of a MPT take place via efficient Unix Domain sockets, MPTs impose a large overhead as the data exchanges between sibling sub-transactions impose a synchronization phase between the worker threads of different partitions and leads to frequent stalls in the processing.

In the high contention workload (Fig. 4.3b), the absolute peak throughput achieved by

SPARKLE is clearly lower than in the previous scenario. Yet, we observe up to approx. $6\times$ speed-up versus the best baseline, i.e., Calvin, which scales only up to 5 threads, as in the previous workload, before being bottlenecked by the sequencing thread. This striking performance gain is achieved despite, as expectable, SPARKLE incurs a high contention rate, given its speculative nature and the high probability of conflicts between of transactions. The most dramatic performance drop, though, is experienced by S-SMR. In this case, when using more than a single thread, the data (which is populated with a single warehouse) has to be sharded over multiple partitions (in this case we partition by district up to 10 threads, and then using random hashing), forcing most transactions to access more than a single partition. This is particularly onerous for long read-only transactions, such as OrderStatus, which access hundreds of keys and force the worker threads of different partitions to synchronize hundreds of times to process a single transaction.

4.9.4 Distributed deployment

Let us now analyze the performance of SPARKLE when deployed over a medium scale cluster encompassing 8 machines.

We start by presenting, in Fig. 4.4, the results for the synthetic benchmark considering four scenarios, which differ by the percentage of MPTs they generate. In each of the 4 plots in Fig. 4.4 we vary, on the X-axis, the percentage of dependent transactions, and report throughput and abort rate for all the considered solutions when using a low (LC) and medium contention (MC) workload. For the case of S-SMR, since its performance is oblivious to the contention level (given that it processes transactions sequentially at each partition), we only report results for the LC workload.

First, let us discuss Fig. 4.4a first, which reports results for a workload that does not generate any MPT. We can see that SPARKLE overall achieves the highest throughput, and that its performance is slightly reduced in the MC workload, but is not affected by the rate of dependent transactions. In this scenario S-SMR also achieves approx. 60% lower throughput than SPARKLE. This can be explained considering that SPARKLE (and Calvin) can process transactions concurrently, using all the available cores (5 in this testbed), whereas S-SMR's single thread execution model intrinsically limits its scalability.

Finally, looking at Calvin's throughput, we can see that its throughput reduces dramatically as the ratio of dependent transaction increases. Nevertheless, even with 0% of dependent transaction, Calvin's throughput is throttled by its scheduling thread, which leads it to achieve lower throughput than both SPARKLE and S-SMR. With 100% of dependent transactions, Calvin thrashes, as the likelihood for dependent transactions to be aborted (possibly several time) quickly grows even in low/medium conflict workloads. In fact, Calvin needs to execute a so called *reconnaissance* phase for dependent transactions to estimate their read- and write-sets, and if the prediction turns out to be wrong during execution, these transactions have to be aborted and re-executed. Note that the high frequency of abort of dependent transactions imposes overhead not only to worker threads, but also to Calvin's scheduler thread – upon each abort and restart of a (dependent) transaction, the scheduler thread has to release and acquire its locks, incurring non-negligible overhead.

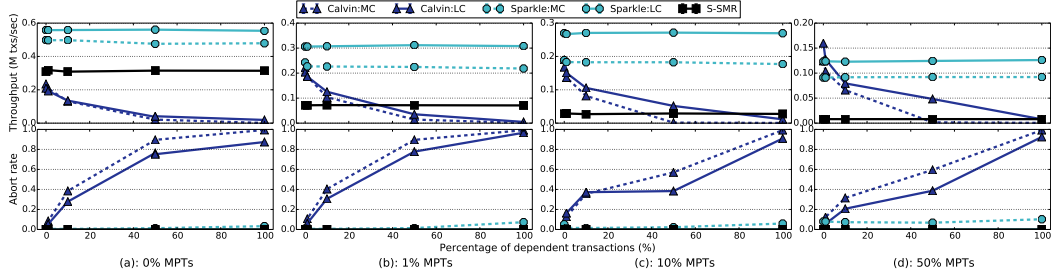


Figure 4.4 – 8 nodes cluster, synthetic benchmark generating workloads with varying contention level, percentage of MPTs and of dependent transaction. *MC* stands for medium contention and *LC* for low contention.

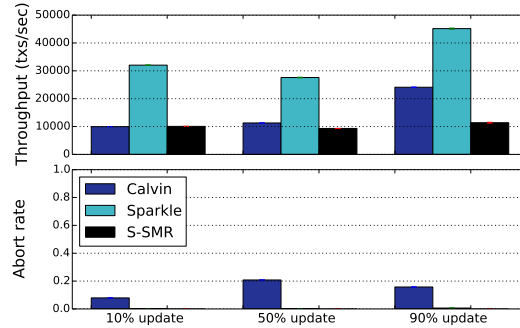


Figure 4.5 – 8 nodes cluster, TPC-C workloads.

Figs. 4.4b, 4.4c and 4.4d report the results obtained when increasing the percentage of MPTs to 1%, 10% and 50%, respectively. The first observation we make is that S-SMR's throughput drops significantly as the rate of MPT grows. As already mentioned in Section 4.9.3, MPTs incur a large overhead with S-SMR, due to the synchronization they impose between the worker threads of different partitions. Since S-SMR uses a single worker thread per partition, whenever a MPT is forced to block waiting for remote data from a sibling partition, no other transaction can be processed at that partition — unlike in Calvin or SPARKLE. In distributed settings, as the communication latency between partitions is strongly amplified (with respect to the single machine scenario considered in Section 4.9.3) the performance toll imposed by MPT also grows radically and S-SMR's throughput is severely throttled by network latency: with 50% MPTs, S-SMR's throughput drops by about $40\times$ compared with the case of no MPTs!

The throughput of Calvin and SPARKLE throughput reduces more gradually as the MPT increases. This is because both of them allow activating the processing of different transactions, whenever an MPT is blocked waiting for remote data. Similar to what already observed in Figure 4.4a, also in this case, the throughput of Calvin drops dramatically in presence of even a small fraction of dependent transactions, approximately by a factor $2\times$ with as low as 10% dependent transactions.

By analyzing SPARKLE, we see that although its throughput also reduces with distributed transactions, its throughput is not affected as significantly as with S-SMR. It is

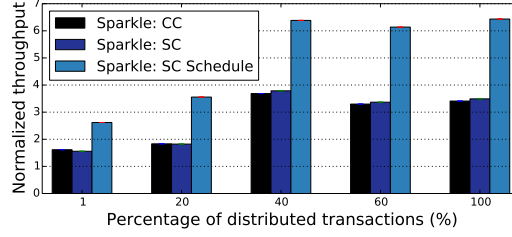


Figure 4.6 – Throughput of SPARKLE : *CC*, SPARKLE : *SC* and SPARKLE : *SC* + *schedule*, normalized to that of SPARKLE : *Cons*, while varying the percentage of MPTs.

also worth noting that in the 50% MPTs scenario and in absence of dependent transactions, Calvin achieves 30% higher throughput than SPARKLE. This can be explained by considering that, in this workload, Calvin’s throughput is upper bounded by the processing speed of MPTs (which take orders of magnitude longer than SPTs) and not by its scheduling thread. Also, due to its pessimistic/lock-based nature, Calvin does not require MPTs to undergo a confirmation phase. Despite SPARKLE strives to minimize the performance impact of the MPTs’ confirmation phase (via the combined use of scheduling techniques and of the SC mechanism), this still introduces additional communication overhead. Nonetheless, we highlight that, in the 50% MPT scenario, SPARKLE outperforms Calvin as soon as the ratio of of dependent transactions is as large as 1%, achieving an average throughput gain (across the considered MPT ratios) of more than one order of magnitude. Analogous gains are observed also with respect to S-SMR.

Next, we present the results obtained using the TPC-C benchmark. Figure 4.5 shows that SPARKLE outperforms Calvin and S-SMR in all workloads, with peak gains of approx. $3\times$ and approx. $4\times$, respectively. The key reason why S-SMR achieves relatively poor performance is that these three TPC-C workload generate a small, but not negligible fraction (varying from approx. 1%, for the 10% update workload, to approx. 10%, for the 90% update workload) of MPT transactions. Calvin’s performance, instead, can be explained considering that three out of the five transaction profiles are dependent transactions, which impose heavy load on the locking thread and are prone to incur frequent restarts.

Benefits of SC and scheduling

Next, we conduct an experiment aimed to quantify the performance benefits brought about by using, either jointly or in synergy, two key mechanisms used by SPARKLE to regulate MPT’s execution: SC and scheduling. Further, we aim to quantify to what extent the use of speculative transaction processing (in particular allowing MPTs to disseminate speculative data to their siblings) can enhance the throughput of MPT transactions. To this end, we compare the performance of four SPARKLE variants:

- SPARKLE: *Cons*: a conservative variant in which MPTs are only allowed to send remote data to their siblings if they are guaranteed to have observed a locally consistent snapshot, i.e., if their preceding transaction has final committed. This spares MPTs from the need (and cost) of any confirmation, but also throttles down throughput severely as it precludes any form of parallelism between MPTs in execution at the same partition.

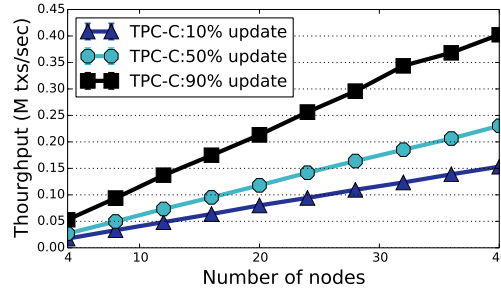


Figure 4.7 – The performance of SPARKLE for three workloads varying the scale of the system.

- **SPARKLE:CC**: in which, as in SPARKLE, MPTs disseminate to their siblings the data they read locally in a speculative fashion. Unlike SPARKLE, though, this variant uses a conservative confirmation (CC) scheme, which sends confirmation messages only when transactions final commit, and not when they speculatively commit. The CC scheme is significantly simpler than SC, as, with CC, a transaction generates exactly one confirmation message, and not an a priori unknown number, as it is the case for SC. However, with CC, a partition can send the confirmation for its $i + 1$ -th transaction, only upon final committing its i -th transaction, which, in its turn, depends on the reception of the confirmation message that is only sent upon the final commit of the $i - 1$ -th transaction. Thus, the throughput of MPTs becomes inherently upper bounded by the rate of completion of the inter-partition confirmation phase, which involves an all-to-all synchronous communication between the involved partitions.
- **SPARKLE:SC**, which uses SCs but not scheduling;
- **SPARKLE:SC+Schedule**, which uses SC and scheduling.

We use the low conflict micro benchmark configuration and generate varying ratios of MPTs. For better readability, in Fig. 4.6 we report the normalized throughput of the three protocols allowing speculative reads across partitions against SPARKLE:Cons. The plot allows us to draw three main conclusions. First, all variants achieve significant (up to approx. $3\times$) w.r.t. SPARKLE:Cons, confirming the relevance of using speculative processing techniques to cope with MPTs. Second, unless coupled with scheduling, SC provides no perceivable benefit with respect to a simpler CC approach: without scheduling, most MPTs need to resort to using a CC scheme, hence the throughputs of SPARKLE:CC and SPARKLE:SC is almost identical. Finally, it allows us to quantify the gains reaped through the joint use of scheduling and SC: up to $2\times$ throughput increase when compared to SPARKLE:CC.

Large scale deployment

Finally, we evaluate the scalability of SPARKLE by using the three previously described TPC-C workloads and increase the number of nodes in the system from 4 to 40. Also in this case, each node hosts a data partition with 12 warehouses, so in this experiment we are scaling both the cluster size and the volume of data. As a consequence, the degree of contention between transactions also remains theoretically constant as the platform's

scale grows (and similar to the levels observed for the medium scale cluster reported in Figure 4.5, and, hence, omitted).

Figure 4.7 shows that SPARKLE scales linearly to 40 nodes, for all three workloads, confirming that the use of speculative transaction processing techniques employed by SPARKLE are effective also in large scale data stores and that they do not compromise what is arguably one of the most relevant property of the PRSM approach: its scalability.

4.10 Summary

In this chapter, we introduced SPARKLE, a novel distributed deterministic concurrency control that achieves more than one order of magnitude gains over state of the art PRSM systems via the joint use of *speculative* transaction processing and *scheduling* techniques. Via an extensive experimental study encompassing both synthetic and realistic benchmarks, we show that 1) SPARKLE has negligible overhead compared with a protocol implementing no concurrency control, in conflict-free workloads, 2) SPARKLE can achieve more than one order of magnitude throughput gains, comparing with state of the art PRSM systems, in workloads characterized by high conflict rates and frequent MPTs.

Chapter 5

Comparing STR and Sparkle

This chapter compares STR and SPARKLE, such that readers can better understand the inherent differences between the two protocols and the suitable workloads and applications for each protocol. However, note that the prototypes of these two protocols are built on different systems, i.e., Antidote [1] for STR and Calvin [122] for SPARKLE, which use Erlang and C++ respectively. These two programming languages are known for having a large performance gap [3, 4]. Further, it should also be noted that SPARKLE and STR provide different guarantees for what concerns consistency – recall that SPARKLE ensures strict serializability, whereas STR provides snapshot isolation. Last but not least, STR and SPARKLE also employed different sets of benchmarks, which are provided by their base systems. In the light of these considerations, it is not feasible to perform apple-to-apple qualitative comparison between these two protocols, thus we focus on *qualitative*, rather than *quantitative* comparison. First, we discuss the suitable applications and workloads for both systems, following our discussion in Section 2.3.4; then, we classify the two systems depending on the feature of their speculative techniques.

5.1 Comparison of base execution models

As STR and SPARKLE extend DUR/SMR execution models, they also inherit the intrinsic properties of their underlying models. Let us revisit our discussion in Section 2.3.4 about DUR/SMR models, to highlight the pros and cons of STR and SPARKLE.

First, if an application requires *non-deterministic* and/or *interactive-style* transaction, the application can only be supported by STR, but not SPARKLE. This is because SPARKLE’s order-then-execute model requires transactions to be deterministic and does not allow clients to issue new operations to a transaction while it is being executed, which is required by interactive transactions.

Second, SPARKLE is likely to achieve higher throughput for high contention workloads than STR. In fact, as discussed in Section 2.3.4, in DUR systems transactions are exposed to the risk of contention during both the execution and ordering/certification phases, whilst SMR systems expose transactions to the risk of contention only during the execution phase.

Finally, STR is expected to be more efficient than SPARKLE when faced with workloads that include a large fraction of computationally intensive update transactions. In fact, update transactions need to be executed in all replicas in the SMR model, but only in one replica in the DUR model. As already observed in the literature [124, 66], DUR systems exhibit a better scalability than SMR systems in these scenarios.

5.2 Comparison of speculation techniques

5.2.1 Internal and external speculation

The current prototypes of STR and SPARKLE only employ internal speculation but not external speculation (recall our definition of internal/external speculation in Section 2.4). Nevertheless, as pointed out also in recent work [93, 54], externalizing speculative results may allow for reaping further performance benefit, such as reducing end-to-end latency [93] or increasing system throughput by allowing external clients to perform optimistic actions (e.g., submitting the next request in an interactive application) based on speculative results [54]. Thus, we discuss the feasibility and potential benefits of both systems to adopt external speculation.

It is straightforward to adopt external speculation in both STR and SPARKLE. Currently in both systems, transactions are speculatively committed before being final committed (after transactions pass local certification in STR, and after transactions finish execution and apply updates in SPARKLE), but the speculative commit is not notified to external clients. Thus, it would be simply necessary to externalize the speculative commit events to applications/users to achieve external speculation in both systems.

Though, we argue that enabling external speculation is likely to achieve large performance gain in STR but not in SPARKLE. Essentially, as external speculation masks the interval between a transaction’s speculative commit from its final commit, the larger the interval is, the higher benefit external speculation should bring. In STR, in between a transaction’s speculative and final commit, the transaction has to undergo global certification; since global certification can take up to hundreds of milliseconds in geo-distributed setting, external speculation will yield large latency reduction and/or throughput gain. In contrast, note that in a typical geo-replicated setting (as described in 4.2), the ordering phase of SPARKLE will take much longer time than the execution phase. Since external speculation can not mask the latency of the ordering phase, it is unlikely to achieve significant performance gain. In fact, if SPARKLE was extended to perform speculation also on the transaction delivery order (e.g., as in AGGRO [90]), then external speculation can lead to more significant benefits also for SPARKLE.

5.2.2 Transparency

As already explained in Section 3.3, STR’s consistency criterion SPSI ensures that even aborted transactions observe SI-compliant snapshots, i.e., speculation is transparent to applications. This is, however, not the case for SPARKLE: SPARKLE ensures serializability for final committed transactions, but aborted transactions may observe arbitrary snapshots. For

instance, recall that to speed up the execution of MPTs, SPARKLE allows sub-transactions of an MPT to exchange data when they are speculatively, but not finally, committed. Imagine that there are two MPTs in two partitions, T_i and T_{i+1} in partitions X and Y , such that, at both partitions, T_{i+1} reads data items that are updated by T_i ; if T_{i+1}^X reads the updates of T_i^X and disseminates its data, while T_{i+1}^Y disseminates its data without reading from T_i^Y , then T_{i+1} is executed observing only part of the updates produced by T_i . This is an execution that could not take place, if transactions were executed sequentially. As such, it violates transparency (see Section 2.4).

A simple way to support transparent speculation in Sparkle would be to disable the speculative confirmation and the speculative data dissemination mechanisms, i.e., disallowing transactions to speculatively read from remote partitions (which we refer to *remote speculative reads* in the following text). However, as we have shown in the evaluation result of SPARKLE (Section 4.9.4), disabling these mechanisms would throttle down considerably performance in presence of even small fractions of MPT transactions, which is the key reason why transparent speculation is not supported in SPARKLE.

Hence, the challenge here is how to extend SPARKLE to ensure transparency without compromising its efficiency. It should be noted that although STR guarantees transparent speculation through the use of efficient mechanisms (as described in Section 3.4.2), STR only allows transactions to speculatively read from transactions originated at the same node. If STR was to allow transactions to read speculatively from transactions originated at different nodes, transparency could be compromised.

Overall, we argue that it is very challenging to design efficient mechanisms to ensure transparency while permitting *remote speculative reads*. We will provide a detailed discussion on this in Chapter 6.

5.3 Summary

We conclude the comparison of STR and SPARKLE using Table 5.1. As already mentioned, due to the difficulty for performing an apple-to-apple quantitative comparison, the discussion is mostly qualitative. Nevertheless, we shed lights not only on which workloads are expected to be best fit to the characteristics of the two proposed protocols, but also highlighted some relevant trade-offs regarding how these protocols support speculative transaction processing techniques.

	Application requirements				Workload features	
	Non-deterministic transaction	Interactive transaction	Transparent speculation	External speculation	Increasing contention	Increasing update ratio
STR	✓	✓	✓	↑↑	↓↓	↓
SPARKLE	✗	✗	✗	↑	↓	↓↓

Table 5.1 – A table summarizing the pros and cons of STR and SPARKLE. ✓ and ✗ means a requirement can be supported by a system or not, respectively. ↑ and ↓ mean performance increases or decreases, respectively; more arrows indicates higher tendency for increasing/decreasing.

Chapter 6

Conclusion

This chapter concludes this dissertation. We first discuss some limitations of STR and SPARKLE and future research directions opened by this dissertation. Then, we conclude the thesis with a final discussion on the results obtained herein.

6.1 Limitations and Future work

6.1.1 Ensuring transparency for speculative reads across nodes

As we have already discussed in Section 5.2.1, ensuring *transparency* while supporting *remote speculative reads* (i.e., reading uncommitted versions of data items that are locally replicated) is still an open question for both STR and SPARKLE. STR ensures transparency, but remote speculative reads are only possible if the version to be speculatively read was created by a transaction that originated at the same local node. SPARKLE does allow speculative remote speculative reads, but does not ensure transparency. To combine these two useful features, we see the following major challenges:

1. The first challenge consists of revisiting the definition of SPSI, which currently prohibits speculative reads from observing the updates produced by speculatively committed transaction originated at a different node. The key challenge here is how to adapt the formal definition, in particular of SPSI-2, while preserving the original spirit of SPSI, i.e., ensuring that the history generated using speculative reads (across nodes) is a history that could still be generated by a system that supports SI. Intuitively, this variant of SPSI should guarantee that the snapshot observed by a transaction does not include the updates produced by concurrent conflicting transactions, while still enabling implementations that can ensure such properties in an efficient way.
2. The second major challenge is related precisely to how to implement the mechanisms used to support speculative reads in an efficient way, i.e., ideally based solely on local decisions or, at least, minimizing the need for exchanging information across

nodes. This is far from being trivial since by allowing transactions to read speculatively committed data produced by remote transactions, it is possible to generate chains of (possibly transitive) conflicts among transactions in execution at a large set of distributed nodes - a problem that does not arise with STR/SPSI, as speculative dependencies can only develop among transactions from the same node. If we allow a transaction to speculatively read data from multiple nodes, though, the local certification phase may have to be performed on all nodes that the transaction can speculatively read from (to detect other conflicting speculatively committed transaction), which would be considerably slower and more expensive than the current local certification scheme. Thus, the practicability of this extension hinges on the efficiency of the mechanisms to enforce SPSI properties across nodes.

6.1.2 STR

Implementing alternative self-tuning mechanisms

An interesting research opportunity raised by STR is related to the design and evaluation of alternative self-tuning mechanisms. The current self-tuning scheme takes a black-box approach that is agnostic of the data store implementation, and uses a simple feedback-driven control loop that decides to turn on or off speculation based on measured throughput, homogeneously for all nodes. This self-tuning scheme could be extended in several directions. First, instead of making homogeneous tuning decision for all nodes, the tuning scheme could make diverse tuning decisions for separate nodes, as each node may experience different workloads, thus being fit for using different speculation degrees. Further, a more fine-grained adaptive policy could be used, so as to support different speculation degrees for different transaction types (and/or transactions' inputs). This could be beneficial considering that different transaction types (or even the same transaction type, when fed with different inputs) often generate different data access patterns — thus being or less more amenable for using aggressive speculative policies. Finally, the tuning scheme may adopt more sophisticated tuning algorithms, e.g., gradient descent [117], in order to cope with more complicated scenarios that require considering multiple parameters, such as the two extensions we have discussed.

6.1.3 Sparkle

Coping with flexible replication settings

SPARKLE is a deterministic concurrency control mechanism that aims at enhancing the performance of partially-replicated state machines (PRSM). In Chapter 4, SPARKLE was evaluated assuming a typical deployment for PRSM systems, also targeted by other recent solutions for PRSM, where each datacenter hosts the full data set while internally splits data into multiple partitions. Nevertheless, SPARKLE is not restricted to such specific deployment, and it could be interesting to consider and experiment deploying SPARKLE in alternative partially-replicated settings, e.g. in a setting where partitions are distributed in different geographically datacenters, such that no datacenter has the full dataset. Such a

scenario could, for instance, occur in case certain subsets of data are mandated to be stored exclusively in specific geographical sites, e.g., due to political restrictions. Considering SPARKLE in such geo-distributed setups poses several new challenges:

1. In such settings, the latency of inter-partition communication (to exchange data among sub-transactions running at different partitions) is subject to a strong increase. Note that this latency is injected during the critical path of execution of transactions and, as such, can have a significant adverse impact on the throughput achievable by the system. This opens up the possibility to extend the speculation techniques already leveraged by SPARKLE to hide this latency. In order to tackle this issue, though, one would have to resort to a different kind of speculative techniques with respect to the ones currently used in Sparkle. In fact, SPARKLE uses speculation to allow transactions to observe uncommitted data. However, in order to mask the latency for exchanging data among the sub-transactions of a MPT, speculation should be used for predicting what data items versions will be received from remote partitions.
2. The system's ordering protocol depends heavily on the fact that the latency between partitions are low, such that it could perform the dispatching phase in each data center to order transactions. In geo-distributed setups, this is not true anymore and the ordering phase will have large latency due to the long communication delay between partitions. To mitigate this problem, one may consider employing genuine ordering protocols [48, 23], in which delivering a transaction only requires all the involved partitions of the transaction to communicate, instead of all partitions in the system. In case workloads do not consist of large proportion of transactions that access partitions across different geo-distributed partitions, these ordering protocol will provide much lower latency.

Optimizing the execution of multiple heterogeneous batches

SPARKLE parallelizes the execution of multiple homogeneous MPTs through the use of the speculative confirmation mechanism. However, transactions that access different sets of partitions, i.e. heterogeneous transactions, can not be executed concurrently – if the transactions that precede a MPT T are still active and they access different sets of partitions than T does, T can not be executed until all its preceding transactions have final committed. This design choice is necessary to ensure the correctness of the SC mechanism. In fact, if an MPT T is aborted by a preceding MPT T' that accesses a different set of partitions, not all the sub-transactions of T may be notified in case T' aborts (since T' only broadcasts its SC messages to the partitions it involves). As a result, a wrong final GAV could be computed for T , leading to the erroneous decision of committing T prematurely. Clearly, the design decision of enabling speculative processing only within a set of homogenous MPTs can considerably hamper parallelism, if there exists a large number of heterogeneous sets of MPTs.

We see several possible ways to cope with this issue. A naive approach to enhance parallelism is to always allow MPTs to execute concurrently, without considering whether

they are homogeneous or not. In this case, though, when an MPT speculatively commits, it should broadcast its SC messages to all partitions in the system. This is necessary as now in theory, an MPT can abort any following transaction that accesses arbitrarily different partitions. Thus, it needs to inform all possible siblings of its following transactions about its abort set. While this mechanism can bring large overhead when the system composes of large number of partitions, there are several other possible approaches. For instance, one may allow at most two sets of consecutive heterogeneous transactions to be executed concurrently, and at the end of processing a homogeneous batch, send a confirming ‘anchor’ message concluding the transactions aborted by all transactions in this homogeneous group, to all partitions involved in the next group.

6.1.4 Combining STR and Sparkle

As discussed in Chapter 5, SPARKLE and STR present different pros and cons and thus are suitable for different workloads. Therefore, an interesting research direction is to design a protocol that combines both SPARKLE and STR, in order to get the best of both worlds. Though, in order to do this, there are several following research questions to be answered.

First, how to combine these two execution models without compromising correctness. In the literature, there exist several works that combine the DUR and SMR models [71], however, to the best of our knowledge, these works only consider fully-replicated settings and do not exploit any speculation techniques. In the partially-replicated setting, DUR and SMR models raise additional challenges, since, to enhance scalability, in partially-replicated settings one wants to avoid coordination schemes that require involving all the nodes in the system in order to commit transactions and, in contrast, impose coordination only among transactions that do access some partition in common. The absence of a global coordination mechanism makes it naturally more complex to detect conflicts between transactions using different execution modes.

Last but not least, another relevant problem is deciding when to switch between different execution models. This switching decision could be made by a system developer, or by employing self-tuning methods, used either on off-line [28, 27] or on-line [82, 124] learning methods.

6.2 Final remarks

Today, large-scale online applications are growing with unprecedented speed, serving millions of clients in geographical scale. To tame the growing complexity of applications, distributed data storage systems have started embracing strong, transactional, semantics. However, providing strongly-consistent transactions comes with a hefty cost, associated with the need to coordinate the execution of concurrent transactions to ensure consistency. At large scale and, increasingly, in geo-distributed system deployments, the coordination costs are further amplified and result in significant overheads affecting system’s throughput and latency. A myriad of works have investigated how to provide weaker forms of transactional semantics in order to enhance system performance. Unfortunately, though, these

proposals can introduce subtle concurrency anomalies, which can place substantial burdens on application developers.

This dissertation studied how to leverage speculation techniques in order to enhance the performance of partially-replicated, geo-distributed transactional data stores. Speculation techniques were applied to two major approaches to provide transactional semantics for replicated data stores, namely deferred update replication (DUR) and state machine replication (SMR), leading to the proposal of STR and SPARKLE, respectively.

STR is a DUR-based transactional system that employs speculative read mechanisms to allow transactions to observe the data item versions produced by pre-committed transactions, instead of blocking until they are committed or aborted. Speculative reads can reduce the ‘effective duration’ of pre-commit locks, allowing achieving higher throughput.

SPARKLE is an innovative deterministic concurrency control designed for Partially-Replicated State Machines (PRSMs). SPARKLE employs a massively-parallel speculative execution model, which allows transactions to be processed out of order, with the objective of maximizing concurrency and taking full advantage of modern multi-core microprocessors. Nonetheless, it ensures that transactions only final commits only if it observes a snapshot equivalent to when transactions are sequentially executed.

Both proposals were evaluated both systems using standard benchmarks and showed that both systems are able to achieve up to an order of magnitude throughput increase compared with state of the art systems.

The mutual pros and cons of the proposed approaches, as well as their limitations, were analyzed, which allowed for identifying a set of possible research directions aimed at further improving the solutions proposed in this dissertation.

Overall, this dissertation has proved that speculation can be an effective means to mask the overheads incurred to enforce strong consistency in partially replicated transactional data stores. Further, it sheds lights on one relevant problem that has to be considered when designing speculative mechanisms for this class of distributed systems: speculation can expose transactions to concurrency anomalies that may not arise in non-speculative systems and that can possibly endanger application’s correctness. Excluding the possibility for this sort of anomalies to occur in a speculative system, i.e., achieving speculation transparency, is possible, but it is not trivial neither from a theoretical nor from a practical perspective.

On the theory side, the main challenge lies in precisely defining what type of properties should a speculative (distributed) concurrency control mechanism ensure. Such guarantees should be flexible enough to enable the development of efficient speculative protocols, yet stringent and clear enough to avoid exposing programmers to subtle concurrency issues and complicated formal semantics. In this sense, the SPSI semantics, proposed in Chapter 2, represent an instance of a possible approach to tackling this problem.

Bibliography

- [1] Antidote source code repository. <https://github.com/SyncFree/antidote>.
- [2] Facebook users. <https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/>.
- [3] The performance benchmark of c++. <https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/gpp.html>.
- [4] The performance benchmark of erlang. <https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/hipe.html>.
- [5] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceedings of the IEEE 36th International Conference on Distributed Computing Systems*, pages 405–414. IEEE, 2016.
- [6] A. L. P. N. Alonso. *Database replication for enterprise applications*. PhD thesis, Universidade do Minho, 2017.
- [7] M. S. Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 32nd IEEE International Symposium on Reliable Distributed Systems*, pages 163–172. IEEE, 2013.
- [8] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):141–155, 2017.
- [9] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research*, pages 223–234, 2011.
- [10] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the 33th International Conference on Dependable Systems and Networks*, pages 149–158, June 2003.

- [11] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
- [12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.
- [13] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails. In *Proceedings of the 2nd Workshop on Determinism and Correctness in Parallel Programming*, 2011.
- [14] P. A. Bernstein and N. Goodman. Multiversion concurrency control – theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.
- [15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Pub. Co. Inc., Reading, MA, 1987.
- [16] C. E. Bezerra, F. Pedone, and R. Van Renesse. Scalable state-machine replication. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342. IEEE, 2014.
- [17] K. Birman and T. Joseph. *Exploiting virtual synchrony in distributed systems*, volume 21. ACM, 1987.
- [18] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: distributed deterministic dataflow programming for erlang. In *Proceedings of the 13th ACM SIGPLAN workshop on Erlang*, pages 51–60. ACM, 2014.
- [19] I. Briquemont, M. Bravo, Z. Li, and P. V. Roy. Conflict-free partially replicated data types. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science*, pages 282–289, Nov 2015.
- [20] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4):20, 2009.
- [21] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
- [22] CHANG, Fay, DEAN, Jeffrey, GHEMAWAT, Sanjay, HSIEH, C. Wilson, WALLACH, and A. Deborah. Bigtable : A distributed storage system for structured data. *Acm Transactions on Computer Systems*, 26(2):1–26, 2008.
- [23] P. R. Coelho, N. Schiper, and F. Pedone. Fast atomic multicast. In *Proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 37–48. IEEE, 2017.
- [24] T. consortium. Tpc benchmark-w specification v. 1.8. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

- [25] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!’s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.
- [26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8, 2013.
- [27] M. Couceiro, P. Romano, and L. Rodrigues. Polycert: Polymorphic self-optimizing replication for in-memory transactional grids. In *Proceedings of the 12th International Middleware Conference*, pages 300–319. International Federation for Information Processing, 2011.
- [28] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):2942–2955, Nov 2015.
- [29] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.
- [30] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 715–726. VLDB Endowment, 2006.
- [31] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.
- [32] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.
- [33] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. Rcdc: a relaxed consistency deterministic computer. In *ACM SIGPLAN Notices*, volume 46, pages 67–78. ACM, 2011.
- [34] D. J. Dewitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *ACM SIGMOD Record*, 19(4):104–112, 1990.
- [35] D. Didona, K. Spirovska, and W. Zwaenepoel. Okapi: Causally consistent geo-replication made faster, cheaper and more available. *CoRR*, abs/1702.04263, 2017.
- [36] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 32nd IEEE International Symposium on Reliable Distributed Systems*, pages 173–184. IEEE, 2013.

- [37] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.
- [38] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 343–354. IEEE, 2014.
- [39] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.
- [40] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
- [41] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84. IEEE, 2005.
- [42] A. Fekete, D. Liarakapis, E. O’Neil, P. O’Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.
- [43] P. Felber, C. Fetzer, R. Guerraoui, and T. Harris. Transactions are back—but are they the same? *ACM SIGACT News*, 39(1):48–58, 2008.
- [44] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [45] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 233–242, Aug 1997.
- [46] S. Frølund and R. Guerraoui. e-transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering*, 28(4):378–395, Apr. 2002.
- [47] D. K. Gifford. Information storage in a decentralized computer system. Technical report, Xerox Res. Cent., 1982.
- [48] A. Gotsman, A. Lefort, and G. Chockler. White-box atomic multicast. In *Proceedings of the 49th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 176–187. IEEE, 2019.
- [49] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 85–96. ACM, 2013.

- [50] J. Gray, P. Helland, P. O’Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Record*, 25(2):173–182, June 1996.
- [51] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.
- [52] Grid’5000. <https://www.grid5000.fr/>, 2018.
- [53] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.
- [54] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental consistency guarantees for replicated objects. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 169–184. USENIX Association, 2016.
- [55] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316, 2001.
- [56] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 9th European Conference on Computer Systems*, page 11. ACM, 2014.
- [57] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. *Proceedings of the VLDB Endowment*, 7(12):1259–1270, 2014.
- [58] V. Hadzilacos. Fault-tolerant broadcasts and related problems. *Distributed systems*, 1993.
- [59] J. R. Haritsa, K. Ramamritham, and R. Gupta. The prompt real-time commit protocol. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):160–181, Feb. 2000.
- [60] P. Helland and D. Campbell. Building on quicksand. *arXiv preprint arXiv:0909.1788*, 2009.
- [61] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [62] S. Hirve, R. Palmieri, and B. Ravindran. Archie: a speculative replicated transactional system. In *Proceedings of the 15th International Middleware Conference*, pages 265–276. ACM, 2014.
- [63] T. Hoff. Latency is everywhere and it costs you sales - how to crush it. High Scalability, july, 25, 2009.

- [64] Intel. Threading building blocks. <https://www.threadingbuildingblocks.org/>, 2018.
- [65] M. Javidi Kishi, S. Peluso, H. F. Korth, and R. Palmieri. SSS: Scalable key-value store with external consistent and abort-free read-only transactions. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, pages 589–600. IEEE, July 2019.
- [66] R. Jiménez-Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 477–, Washington, DC, USA, 2002. IEEE Computer Society.
- [67] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 603–614. ACM, 2010.
- [68] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.
- [69] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 237–250, 2012.
- [70] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1018–1032, 2003.
- [71] T. Kobus, M. Kokocinski, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems*, pages 286–296, July 2013.
- [72] R. Kotla, M. Balakrishnan, D. Terry, and M. K. Aguilera. Transactions with consistency choices on geo-replicated cloud storage. Technical report, Microsoft Research, September 2013.
- [73] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.
- [74] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

- [75] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [76] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
- [77] J. Li, E. Michael, and D. R. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120. ACM, 2017.
- [78] Z. Li. Sparkle codebase. https://github.com/marsleezm/spec_calvin.
- [79] Z. Li, P. Romano, and P. Van Roy. Sparkle: Speculative deterministic concurrency control for partially replicated transactional stores. In *Proceedings of the 49th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 164–175. IEEE, 2019.
- [80] Z. Li, P. Van Roy, and P. Romano. Enhancing throughput of partially replicated state machines via multi-partition operation scheduling. In *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications*, pages 1–10. IEEE, 2017.
- [81] Z. Li, P. Van Roy, and P. Romano. Exploiting speculation in partially replicated transactional data stores. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 640–640, New York, NY, USA, 2017. ACM.
- [82] Z. Li, P. Van Roy, and P. Romano. Transparent speculation in geo-replicated transactional data stores. In *Proceedings of the 27th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018.
- [83] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM, 2005.
- [84] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don’t settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.
- [85] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*, pages 368–377. IEEE, 2014.
- [86] P. J. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 454–465. IEEE, 2011.

- [87] P. J. Marandi, M. Primi, and F. Pedone. Multi-ring paxos. In *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2012.
- [88] R. G. Martínez, Z. Li, A. Lopes, and L. Rodrigues. Augure: Proactive reconfiguration of cloud applications using heterogeneous resources. In *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications*, pages 1–8. IEEE, 2017.
- [89] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems*, 9(4):19, 2015.
- [90] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting STM replication via aggressively optimistic transaction processing. In *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications*, pages 20–27. IEEE, 2010.
- [91] R. Palmieri, F. Quaglia, and P. Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems*, pages 59–64. IEEE, 2011.
- [92] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, pages 1–8. IEEE, 2010.
- [93] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. Planet: making progress with commit processing in unpredictable environments. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 3–14. ACM, 2014.
- [94] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.
- [95] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.
- [96] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proceedings of the VLDB Endowment*, 5(2):85–96, 2011.
- [97] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.
- [98] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

- [99] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. SPECULA: speculative replication of software transactional memory. In *Proceedings of the 31st IEEE Symposium on Reliable Distributed Systems*, pages 91–100. IEEE, 2012.
- [100] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, pages 456–475. Springer-Verlag New York, Inc., 2012.
- [101] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems*, pages 455–465. IEEE, 2012.
- [102] D. R. Ports and K. Grittnr. Serializable snapshot isolation in PostgreSQL. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.
- [103] F. Quaglia and P. Romano. Ensuring e-transaction with asynchronous and uncoordinated application server replicas. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):364–378, March 2007.
- [104] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment*, 7(10):821–832, 2014.
- [105] P. Romano and M. Leonetti. Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In *Proceedings of the 2012 International Conference on Computing, Networking and Communications*, pages 786–792. IEEE, 2012.
- [106] M. M. Saad, M. J. Kishi, S. Jing, S. Hans, and R. Palmieri. Processing transactions in a predefined order. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 120–132, New York, NY, USA, 2019. ACM.
- [107] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems*, pages 266–275, July 2013.
- [108] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*, pages 214–224. IEEE, 2010.
- [109] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
- [110] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3, NSDI’06*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

- [111] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 290–297. IEEE, 2007.
- [112] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, et al. F1: the fault-tolerant distributed rdbms supporting google’s ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM, 2012.
- [113] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.
- [114] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 385–400, New York, NY, USA, 2011. ACM.
- [115] I. L. Stats. Google statistics. <http://www.internetlivestats.com/google-search-statistics/>.
- [116] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
- [117] R. S. Sutton, A. G. Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.
- [118] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.
- [119] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–182, New York, NY, USA, 1995. ACM.
- [120] A. Thomson. Calvin codebase. <https://github.com/yaledb/calvin>.
- [121] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.
- [122] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

- [123] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, 1989.
- [124] P. T. Wojciechowski, T. Kobus, and M. Kokocinski. State-machine and deferred-update replication: Analysis and comparison. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2016.
- [125] D. Yang. *Java Persistence with JPA*. Outskirts Press, 2010.
- [126] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balesgas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, pages 75–87. ACM, 2015.
- [127] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 263–278. ACM, 2015.