TÉCNICO LISBOA

Fine Grained Transaction Scheduling In Replicated Databases Via Symbolic Execution

Miguel Cândido Viegas

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Paolo Romano
Prof. Miguel Matos

Examination Committee

Chairperson: Prof. Full Name
Supervisor: Prof. Full Name 1 (or 2)
Member of the Committee: Prof. Full Name 3

October 2018

For my parents and grandfather,

Acknowledgments

First, I would like to thank my two advisors, Professor Paolo Romano and Professor Miguel Matos, for their guidance during this long and stressful year. Thanks for every meeting, all the email exchanged and, most important, for their patience with me, it was no easy task. I would also like to thank Pedro Raminhas and Nuno Machado, for making a part of our team and for the huge support they given me during the implementation and also with the process of writing this thesis.

Secondly, I would like to thank all my family, in particular my parents. Eduarda, my mother for always supporting me no matter what and for helping me in every obstacle I faced. João, my father for always encouraging me to learn new things and for supporting my decision on which field I wanted to study. I have also to thank my grandfather, José, for being the best person I know and for working hard all his life to give my family what we have today. Lastly, in the family department, I need to thank my two older brothers: Margarida for being available during all life to help me, advise or just to talk; Vasco for all the advises on what is best for me and for trying to make me a better person... although sometimes giving me a headache.

Lastly, I would like to thank all my friends that made a big part of my life over the years and during the time doing this thesis. Special thanks to Raquel, João, David, Dharita, Eduardo and Paulo.

Resumo

Nos dias de hoje, a maioria dos serviços disponíveis na Internet dependem de base de dados para armazenar a sua informação. Estes serviços tendem a ter fortes requisitos de escalabilidade, disponibilidade e tolerância a faltas, o que requer com que sejam desenvolvidas técnicas eficientes de replicação de base de dados. No entanto, nestes sistemas a replicação introduz custos não negligenciáveis para garantir que o estado das replicas é mantido devidamente sincronizado.

Uma abordagem clássica, é a State Machine Replication (SMR), que é uma técnica usada para implementar soluções tolerantes a faltas. SMR tem algumas limitações no que conta ao paralelismo, pois requer que a execução seja determinística. Para resolver estas limitações, as soluções do estado da arte dependem que os acessos a ser feitos sejam determinados automaticamente ou pelos programadores. O último caso, não é perfeito pois identificar os acessos de transações complexas não é trivial e no primeiro caso ou os acessos determinados não são precisos ou é feito uma suposição otimista que aumenta a probabilidade de ocorrerem abortes. Isto tem um impacto no grau de paralelismo e no desempenho geral do sistema.

Esta tese resolve as limitações destas soluções usando a Execução Simbólica para determinar a priori e com precisão os acessos que as transações realizam. Com isto tornaremos o processo de escalonamento mais eficiente.

A nossa abordagem Symbolic-SMR, foi avaliada usando uma micro-benchmark e usando a benchmark TPC-C . Nestas experiências, a Symbolic -SMR superou o desempenho das soluções do estado da arte em 2 a 5 vezes.

Palavras-chave: Replicação de Base de Dados, Replicação Total, Execução Simbólica, Transações Distribuídas, Escalonamento de Transações, State Machine Replication

Abstract

Nowadays, most modern Internet services make large use of databases to store relevant data. These services tend to have strong scalability, high availability and fault tolerance requirements that create a strong urge for designing highly efficient database replication techniques. However, in environments, such as database systems, that offer strong consistency guarantees, replication introduces non-negligible costs in order to ensure that the state maintained by the various replicas is properly synchronized.

A typical approach is State Machine Replication (SMR), which is a technique to implement fault-tolerant solutions. SMR has limitations when it comes to parallelism because it requires deterministic execution. However, to solve these limitations, state of the art solutions rely either on automatic prediction or programmer input about the set of data items to be accessed. The latter is not optimal since complex workloads exhibit non-trivial storage accesses which are hard to predict while the former either relies on coarse-grained prediction or on an optimistic guess that increases the probability of aborts in case of misprediction. This impacts the solution parallelism degree as well as the overall system throughput, respectively.

This thesis addresses the aforementioned limitations, by the use of Symbolic Execution to determine a fine-grained a priori knowledge of the transactions' conflict classes to improve the efficiency of the scheduling process.

To evaluate Symb-SMR, we used a micro-benchmark and the TPC-C benchmark. In these experiments, our solution achieved a throughput 2 to 5 times higher than current state of the art solutions.

Keywords: Database Replication, Full Replication, Symbolic Execution, Distributed Transactions, Transaction Scheduling, State Machine Replication

Contents

List of Figures

Chapter 1

Introduction

Nowadays, most services available over the Internet make large use of databases to store relevant data, like an inventory of an online store. These services tend to have strong scalability, high availability and fault-tolerance requirements that are typically solved using replication techniques. This has created a strong urge for designing highly efficient database replication techniques. Replication allows to tolerate crashes of individual replicas while increasing the perceived availability of systems by placing multiple copies of applications' data across failure-independents machines. However, state of the art database replication introduces non-negligible costs in order to ensure that the state maintained by the various replicas is properly synchronized.

A typical approach to ensure these requirements in replicated systems is based on the State Machine Replication technique (SMR) [12]. In a nutshell, SMR is based on an order-then-execute approach that operates in rounds. In each round, replicas first reach an agreement using some consensus protocol, on a totally ordered set of (deterministic) operations to be executed at all replicas. Next, the set of operations are executed independently at each replica in an order that is consistent with the total order established during the agreement phase.

A key challenge of SMR approaches is how to ensure that transactions executing at different replicas are serialized in the same order. This is important to take advantage of multicore systems, where the order on which the transactions execute must be deterministic. Because the order in which transactions are executed could differ between replicas resulting in inconsistency. In order to remove the costs associated with the execution of the distributed agreement phase, state of the art solutions batch in each round, a large number of transactions. In these scenarios, the maximum throughput achievable by the system is typically bound by the speed at which replicas can process the set of transactions agreed upon using consensus.

Conventional concurrency control schemes, long studied in the literature on transactional

systems [11, 40], suffer from a main problem when employed with SMR-based replication techniques: they are not deterministic, i.e., they ensure equivalence to some serial execution, but provide no guarantee that the transaction serialization order at different replicas will coincide. In order to mitigate this issue, various techniques have been proposed in the literature, based on different approaches. Schemes such as NODO [32] assume a priori knowledge of the data that is going to be accessed by transactions, designated as transactions' conflict classes. An alternative is to estimate that data by doing a *reconnaissance phase* of the transaction before replicating it, as proposed by Calvin [39]. Afterwards, at commit-time, the conflict classes are compared to the previous *recognized* ones to infer if the transaction behaviour has deviated from the expected and if there are deviations the transaction must abort. This approach incurs serious drawbacks in geo-replicated scenarios where transaction submission and transaction execution are, on average, temporally separated in the order of tens of milliseconds since the vulnerability window of transactions to abort is delayed until commit-time.

Certification-based solutions [35, 25], take a different approach and execute transactions optimistically in non-deterministic orders at the different replicas. These solutions rely on Atomic Broadcast (AB) to establish the order in which transactions should be executed, by just one replica), and then validated by all replicas: a transaction is only committed if it is found not to have conflicted with any other transaction ordered before it according to the order specified in the AB. Unfortunately, in high conflict workloads, the performance of Certification-based solutions is highly affected by a large number of aborts. This is because a transaction is executed regardless of what transactions are also being executed (locally or in other replicas). It is only in the validation process that conflicts are detected after computing power and time has been wasted in the transaction execution.

SMR-based solutions are not affected by the limitations of Certifications, because execution is done after scheduling. The transactions are scheduled based on the a priori knowledge their conflict classes. However, approaches [32] that assume this a priori knowledge rely on coarse grain data access patterns to schedule the transactions, which can severely and unnecessarily restrict concurrency. Other solutions [39], that estimate transactions' accesses via a *reconnaissance phase*, are subject to large overheads. This is especially true, in high conflict workloads, as they require executing each transaction at least twice (for *reconnaissance* and for actual execution), and possibly more in case the *reconnaissance phase* is inaccurate.

1.1 Goals

The problem with the current state of the are solutions is on the difficulty to precisely determine a priori the data access patterns of transactions. We propose Symbolic-SMR (Symb-SMR) solution that uses Symbolic Execution (SE) to solve this problem. Symbolic Execution (SE) is a technique originally developed for software testing which allows to determine every possible execution branch of a code block. As it will be shown later in this document, Symb-SMR leverages the fact that SE provides correct and fine-grained transactions' data access pattern estimation to employ a highly concurrent, deterministic concurrency control that allows to maximize workloads' concurrency level, while maintaining consistency among replicas and reducing transactions' vulnerability window by depending solely on data accessed at server-side.

1.2 Thesis Outline

The remainder of the document is structured as follows. Section 2 discusses the background and related work, where it covers the subjects of Database Replication, Transaction Scheduling and Symbolic Execution. The design of Symb-SRM is presented in Section 3. Section 4 presents the evaluation of the Symb-SMR discussing the benchmarks used and comparing our system with state of the art approaches. Finally, Section 5 concludes the document and discusses future work.

Chapter 2

Background and Related Work

This chapter surveys the state of the art on the different topics covered in our work. The remainder of this chapter is organized as follows: Section 2.1, reviews the state of the art of Database Replication techniques. Section 2.2 focuses on Transaction Scheduling in Replicated Databases and in Transactional Memory. Finally, Section 2.3, provides some background on Symbolic Execution.

2.1 Database Replication

Data replication has been an increasing concern over the years, in order to increase the availability and performance of large-scale distributed database systems. For a system to be available it must be capable of withstanding multiple failures, i.e. be fault-tolerant. One way to accomplish this is by replicating the data on more than one site. State of the art Replicated Databases can be coarsely divided by: (1) whether replicas maintain a full state of data, named Full Replication, and typically employed in small clusters, or (2) whether they maintain only a subset of data, which is designated as Partial Replication, and typically employed in larger clusters. Since Database Replication has been a hot research topic for quite some time, the number of contributions to this area has been very large. Thus, since this thesis focus on Full Replication, this section will only overview the state of the art related to this area.

Figure 2.1 shows our taxonomy of database replication techniques, with the root on full replication. This taxonomy is inspired by another taxonomy proposed by Couceiro et al. [16].

Full replication can be achieved using one of two approaches: Single Master or Multi Master, depending on whether update transactions are executed only at a single node (called Master) or at any of the available nodes. In Section 2.1.1 and Section 2.1.2, we will briefly address Single Master and Multi Master, respectively.

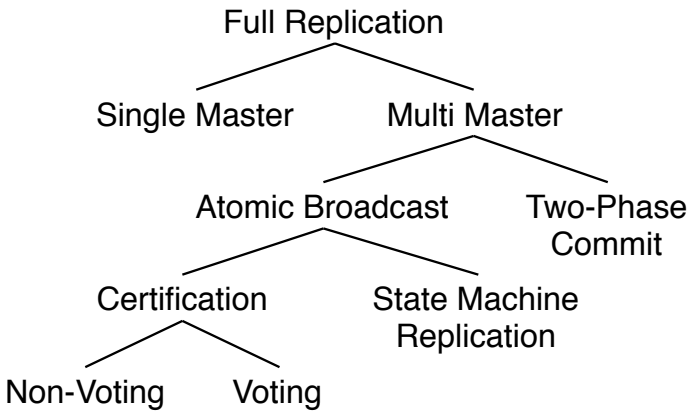


Figure 2.1: Taxonomy for full replication of databases, based on Couceiro et. al [16]

2.1.1 Single Master

A Single Master (SM) approach is constituted by an arbitrary number of nodes, where one of those nodes is appointed as *primary*, also known as master, and the rest are considered *backups*. Clients submit update transactions solely to the *primary* node, which first processes the transaction and then propagates the corresponding state changes to the backup nodes. Read-only transactions can be processed by any node. This allows to have high throughput in read intensive workloads compared to update workloads, because all update transactions need to be processed by a master which lowers the throughput on update workloads. Having a master also means that we have a single point of failure. When the master fails, the *backup* nodes need to:

- detect the failure;
- rollback transactions that have been processed by the master but not yet propagated to all replicas;
- nominate a new master between all replicas via consensus.

This process has been designated as *fail-over*, by Schneider et al. [12]. Faults that affect the master, thus have a big impact on the overall system performance. In case of failure of a *backup* node, the system maintains its normal behaviour and ignores the faulty node.

Many solutions have been proposed that implement a SM approach, such as MySQL Replication [4] and PostgreSQL (Single Master) [5]. These are mainly used to provide high availability, improve fault tolerance and scale out read workloads.

PILEUS

Pileus [38] is a recent solution that relies on SM approach. Pileus is a storage system that aims to relieve application developers from the burden of explicitly choosing a single ideal consistency. It

achieves this by providing a service level agreement (SLA) that allows developers to define quotas of the application's desirable consistency and latency. Depending on the SLA, Pileus chooses to which server (or set of servers) each read is directed. It also allows different applications to obtain different consistency guarantees while sharing the same data.

All update operations received by Pileus are performed and strictly ordered at a primary node. Secondary nodes eventually receive all updated objects via an asynchronous replication protocol. However, depending on the SLA, when guaranteeing strong consistency, the system will contain a mixture of strongly and eventually consistent nodes. The strongly consistent nodes are synchronously updated, whereas the others are asynchronously updated. This allows to have nodes that execute strongly consistent read operations, whereas others act solely as backups.

Terry et al. [38] identify two advantages of implementing Pileus with a SM approach. The first is that, the primary node act as an authoritative copy for answering strongly consistent reads. The second is that the system avoids conflicts caused by different clients concurrently updating (thanks to having a master node), which is beneficial in workloads with very high contention.

2.1.2 Multi Master

In a Multi Master (MM) approach, unlike SM, updates can be issued to any node. The failure handling mechanism of MM differs from the one of SM. In MM, when is detected that a node is faulty, the system can continue working without that node. Although, faults in the nodes do not impact MM system's performance, those nodes need to be later replaced. However, compared to SM, MM requires more expensive synchronization protocols between nodes, in order to guarantee consistency amongst all nodes. The order in which the corresponding transactions are serialized must be identical at every node, to avoid any deviation in the state between them. This has an impact in the throughput and the overall performance of the system.

Both approaches have complementary advantages and disadvantages. The key advantage of SM is simplicity: by executing transactions at a single site, in fact, SM approaches avoid a priori the problem of synchronizing the execution of multiple update transactions running at different replicas. However, SM has limited scalability in update-intensive workloads, due to having just one master. Although MM overcomes most of SM limitations, it does not have a good performance in high contention workloads. This is due to none of the state of the art solutions efficiently schedule the transactions to take advantages of the parallelism capabilities of current machines. This is precisely the motivation of this thesis and a workshop published based in this work by Raminhas et al. [34]

Multi Master approaches can be coarsely classified depending on whether they use Two-Phase Commit or Atomic Broadcast. These two subclasses will be reviewed, respectively, in Section 2.1.3 and Section 2.1.4.

2.1.3 Two-Phase Commit

Two-Phase Commit (2PC) is a well-known protocol to guarantee the atomicity of distributed transactions [11], whose execution is illustrated in Figure 2.2.

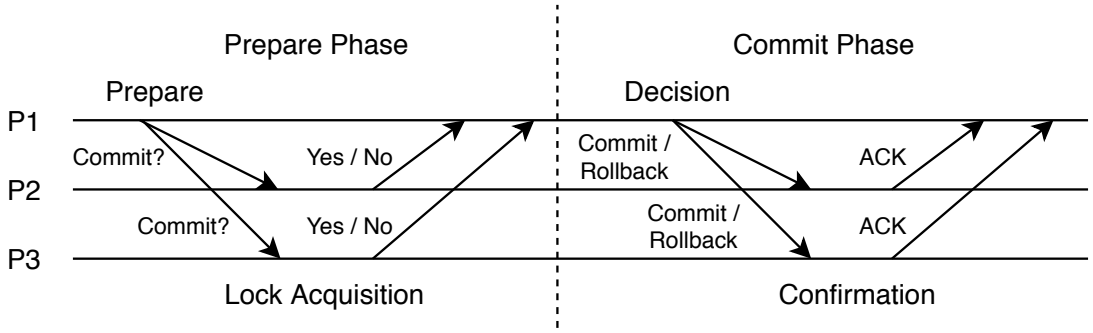


Figure 2.2: Two-Phase Commit Protocol, based on Couceiro et. al [16]

The protocol can be split into two phases, Prepare and Commit. In the Prepare phase, the coordinator (P1) sends a prepare message, requesting the participant nodes (P2 and P3) to acquire locks on the data items accessed during transaction execution. Then all participants send back a reply (Yes or No) depending on whether they succeeded in acquiring the requested locks. Next, in the Commit phase, the coordinator sends a final decision (i.e., commit or abort) to all participants: in case even a single participant voted negatively, the transaction is aborted; otherwise, it is committed. Finally, the participants confirm to the coordinator the transaction successful commit.

2PC can be straightforwardly used to deal with fully replicated data. In this case, the concurrent execution of two conflicting transactions issued at different replicas is detected during the prepare phase. However, in some scenarios, 2PC can cause distributed deadlocks, which are very hard to debug. Another down-side of 2PC is that, in conflict intensive workloads, the likelihood of incurring in transaction aborts grows cubically with the number of replicas [23]. Despite these limitations, recent systems use variants of 2PC. One example is Sinfonia [9], a system that exploits a priori knowledge of the data items to be accessed by transactions (called mini-transactions) that enables efficient and consistent access to data in distributed systems.

2.1.4 Atomic Broadcast

Atomic Broadcas (AB) is a protocol to perform broadcasts between a group of processes where the correct processes deliver the same set of messages in the same order. It is termed as atomic, because either all processes eventually deliver the message correctly, or none of them deliver the message. AB must satisfy the following properties, originally defined by Chandra and Toueg [14]:

- *Validity* - if a correct process broadcasts a message, then all correct processes eventually delivery that message.
- *Uniform Integrity* - one message is delivered at most once.
- *Uniform Agreement* - if a message is delivered by a correct process, then all correct processes will eventually deliver that message.
- *Uniform Total Order* - messages are totally ordered; i.e., if a correct process delivers m_1 first and m_2 afterwards, then every correct process must deliver m_1 before m_2.

Next, we will address two categories of replicated systems that make use of the AB primitive, namely State Machine Replication, in Section 2.1.5, and Certification, in Section 2.1.6.

2.1.5 State Machine Replication

SMR is a well-known technique for implementing a fault-tolerant service, proposed by Schneider et. al [36]. In the SMR protocol, replicas reach a consensus on a total order of transactions to be executed. This eliminates the need of having distributed transactions since they are only executed by one replica. First SMR approaches were single threaded and the execution order followed complied with the total order agreed by all replicas. However, with the introduction of multi-core processors it is now possible to execute more than one transaction concurrently. However, the order of transactions executed must be the same through all replicas, so their states do not diverge. To achieve this, conflicting transactions must be ordered equally by all replicas whereas non-conflicting transactions can be executed in parallel [28, 29].

This ensures that all replicas start with the same state and keep an equal state after each transaction execution, without the need of replicas exchanging messages.

Figure 2.3 shows an example of parallel execution of two transactions in SMR with two replicas. In Parallel SMR approaches, transactions are first disseminated using AB. Upon delivery, any locks protecting the data items to be accessed by the transactions are acquired before executing the transaction. In order to ensure that locks are acquired in an order compliant

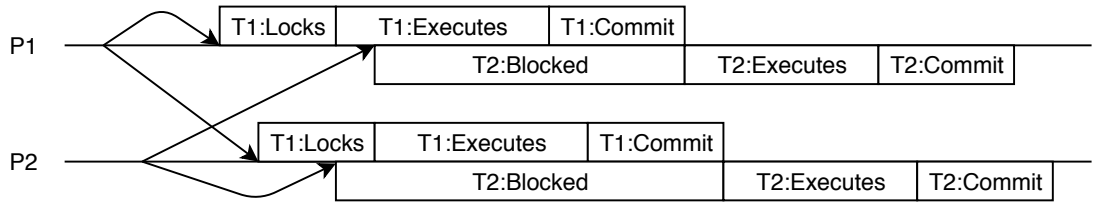


Figure 2.3: Two Parallel Transactions in SMR, presented in Couceiro et. al [16]

with the one established by the AB, the lock acquisition phase is executed by a single thread at each replica. As soon as all the necessary locks are acquired, the transaction can be executed locally in parallel and committed without the need for any remote synchronization. Also, Figure 2.3 shows the interactions between parallel transactions, i.e. a transaction that happens in the same time frame of other transaction. Transaction T1 is delivered to both replicas, and both acquire the necessary locks to execute T1. Afterwards, transaction T2 is delivered but it cannot acquire the locks because of T1, therefore it blocks until the locks are free again. The locks are free when T1 is committed (or aborted), only then T2 can acquire the locks and start execution. The reliance on AB to establish the transaction serialization order makes this solution way more effective than approaches based on 2PC in contended workloads.

NODO

NODO [32] or NOn-Disjoint conflict classes and Optimistic multicast uses a transaction reordering technique to avoid aborts. NODO executes transactions at only one site (no distributed transactions) and allows transactions to access more than one conflict class. A conflict class is the set of data items accessed by a transaction. NODO assumes that conflict classes are identified and determined a priori by the developer. It uses the given conflict class to establish a queue, where each conflict class has a respective queue. Transactions are then inserted in the corresponding queue to its conflict classes. For instance, consider conflict classes Cx and Cy and transactions T1, T2 and T3 with conflict classes, C_{T1}={Cx, Cy}, C_{T2}={Cx} and C_{T3}={Cy}. Knowing this, NODO will queue these transactions, following the order of delivery, as follows: Cx={T1, T2} Cy={T1, T3}. Since T1 is at the head in both queues can be executed while T2 and T3 must wait. When T1 is finished, T2 and T3 can be executed concurrently because both have different conflict classes. The problem with NODO is that requires developers to provide the conflict classes of transactions. This requirement expects that the conflict classes provided by the developers are correct which is a very optimistic expectation. Other problem is that the conflict classes that NODO uses only consider the table of the storage access, which is too coarse-grained of an information to efficiently control the concurrency of transactions.

Calvin

Calvin [39] is a transaction scheduling and data replication layer that orders transactions' execution deterministically to reduce the contention costs associated with distributed transactions. Calvin is designed to run in a SMR like system, where replicas do not have to share information between them. One goal of Calvin is to avoid the problem that holding locks bring, where some lock agreement protocols, like two-phase commit, require multiple message exchanges between replicas to work. These agreement protocols have an impact on the time required to execute a transaction. Calvin's approach to achieve inexpensive agreement is to do this outside of the transactional boundaries. Once the agreement on how to handle a transaction is defined, the rest of the execution must be done accordingly to the plan. This plan needs to be established deterministically so no replicas' state diverges.

To perform the concurrency control of transaction execution, Calvin uses a *sequencer* and a *scheduler*. The *sequencer* is responsible for collecting transaction requests every 10 milliseconds and then compiling all the collected transactions into a batch. The batch is then sent to the *scheduler* containing a unique replica ID, a batch number (that is incremented every 10ms) and all transactions' inputs. When the *scheduler* receives the batch from the *sequencer*, it goes through each transaction (following the order set by the *sequencer*) requesting all locks that the transaction will need. The lock requests are granted strictly following the order in which the requests are made by the transactions and are released only when the transaction is executed to completion. To achieve this scheduling procedure, it is required that all transactions declare their full read/write sets in advance. This information needs to be explicitly provided by the client when issuing a request. This puts the burden of analysing the transaction and determining the conflict classes on the developer. The client is also responsible for performing all remote reads that could be needed to determine the complete conflict classes of a transaction. In scenarios where the scheduling process requires remote reads, Calvin may throw arbitrary aborts. This is due to the fact that remote reads, executed before submitting the transaction, may no longer be accurate. This could originate conflicts during execution that will result in aborts. Upon aborts, the client is responsible for re-submitting the transactions to be sequenced, scheduled and, finally executed again. This retry process, in a high contention workload, results in a non-negligible overhead.

2.1.6 Certification

In Certification-based approaches, replicas execute one transaction each and then propagate the changes through all replicas. This allows, in low contention scenarios, to achieve better

throughput. However, the decrease in execution time is negligible in high contention workloads due to the high number of aborts. An abort occurs when conflicting transactions are concurrently executed in different replicas. This is the main limitation of these approaches.

Next, we describe two Certification-based approaches: Voting and Non-Voting.

Voting

The Voting protocol consists of two phases, a broadcast phase, where the transaction write set are delivered to all replicas, and a voting phase, where is decided if a transaction is committed or aborted.

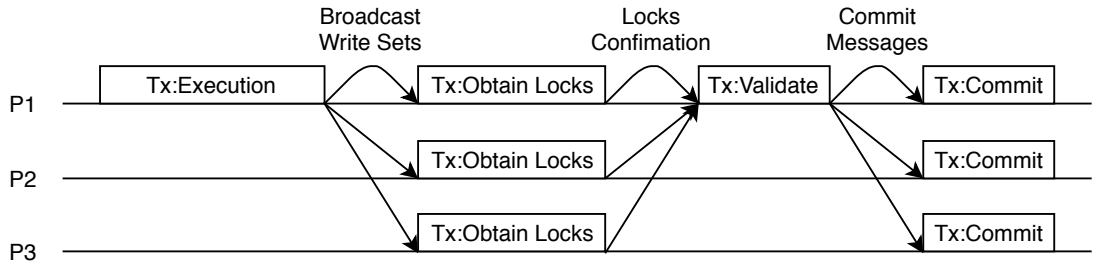


Figure 2.4: Voting Protocol, based on the algorithm described by Rodrigues et al. [35]

Figure 2.4 shows an execution example of the Voting protocol. This figure is based on the general outlines of the algorithm described in Rodrigues et al. [35].

As mentioned before, a transaction, in a Certification approach, is executed in just one replica, designated as the *delegate* node. When executing a transaction, the read locks on the read objects are acquired (because in order to perform a write operation, the object must be previously read). When the transaction is ready to be committed, the write set of the transaction is sent to all replicas though AB. When a replica receives the transaction's write set, it tries to obtain the write locks needed. On one hand, if there is a transaction holding a write lock on any object, the transaction is placed on hold until the locks are released. On the other hand, if a transaction holds a read lock on an object belonging to the write set, the transaction is aborted. When the *delegate* receives the write locks from all replicas, it sends a commit message through AB. Then every replica applies the transaction's writes and releases the write locks. If the *delegate* receives an abort message, the transaction is aborted in all replicas and every lock acquired is released.

The Certification protocol suffers from the limitation of incurring in a considerable number of aborts due to read locks conflicts. Rodrigues et al. [35] suggests some optimization to minimize this problem. Instead of aborting the transaction immediately when a read lock is encountered, the authors suggest, that the transaction could be placed in an alternative state, called *executing*

abort. Consider a transaction *T'* in the *executing abort* state blocked due to transaction *T*. If *T* ends up being aborted, transaction *T'* could resume execution. On the other hand, if transaction *T* ends up executing normally and commits, then transaction *T'* will still be aborted as before. This optimization slightly reduces the number of aborts, but the issue still stands.

Non-Voting

The Non-Voting protocol is very similar to the Voting protocol. After transaction execution in the *delegate* node, the profile of the transaction is sent to every replica. This profile includes the set of objects accessed (read or written) and their version number. The difference between Non-Voting and Voting protocols is that, in Non-Voting, as the name suggests, there is no voting phase. Each replica validates and takes the decision to commit or abort by themselves. At commit time, the transaction profile is broadcasted to all replicas. If a replica has read an object and meanwhile receives the validation of other replicas, it only aborts if the version number of the arriving transaction is smaller than the version number of the local transaction.

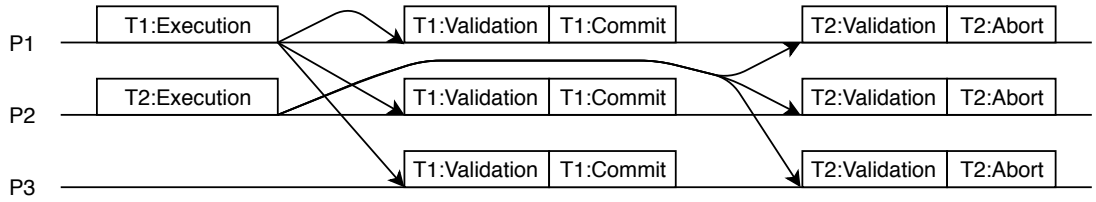


Figure 2.5: Non-Voting Protocol, based on the algorithm described by Rodrigues et al. [35]

Figure 2.5 illustrates a possible scenario of Non-Voting protocol that is described in Rodrigues et al. [35]. This is similar to a scenario presented in Couceiro et al. [16].

As Figure 2.5 shows, transactions T1 and T2 are firstly executed in their respective *delegate* node. When the execution finishes and the transactions are ready to be committed, both delegate nodes send the transaction profiles to all replicas. In this scenario, transaction T1 is first delivered to the replicas via AB. Each replica, having in consideration the transaction profile, validates and commits (or aborts) transaction T1. A transaction can be committed, only if there are no conflicts with any other local transactions. There is no conflict if the version of the objects read by the transaction being validated are greater or equal to the versions of the objects stored locally. If this condition is true the transaction is committed, otherwise is aborted. Seeing that transaction T1 and T2 have the same version number, T1 will commit and T2 will abort because when T2 is validated, the version number will be lower than the one that the replicas have. Since this process is deterministic and all replicas, including the *delegate* nodes, receive transaction by the same order, all (non-faulty) replicas will achieve the

same decision about the outcome of both transactions.

This protocol suffers from the same weakness of Voting. To reduce the impact of this, Rodrigues et al. [35] suggested an improvement: when validating a transaction, if it ends up being validated and committed, local running transactions that conflicted with it, are aborted. This way, it is spared AB messages and computational power.

2.1.7 Summary

In the previous section, we discussed some of the main approaches for fully replicating a database. Table 2.6 summarizes and compares each approach with the following criteria:

- *Example implementations* - example of implementations of the approaches presented.
- *Weak points* - conditions (e.g. workloads) where the limitations of each approach have the biggest impact.
- *Strong points* - points where the solutions excels comparing to the rest of the state of the art.
- *A priori knowledge of transactions' read and write sets* - if the approach needs a priori knowledge of the transactions conflict classes to execute properly.

				Example Implementations	Weak Points	Strong Points	A Priori Knowledge of Conflict Classes
Single Master				[5], [6], [39]	Limited scalability in update intensive workloads (with low transactions' conflicts)	Simple and effective in low % update transactions or very high conflict	No
Multi Master	Two-Phase Commit			[10]	Subject to deadlocks in high contention	Simple	No
	Atomic Broadcast	State Machine Replication		[33], [40]	Poor parallelism with coarse-grained conflict classes	Scalable even with high conflict workloads	Yes
		Certif.	Voting	[38]	Subject to high abort rate in high contention, higher commit latency vs non-voting	Scalable with low conflict workloads	No
			Non-Voting	[36]	Subject to high abort rate in high contention	Scalable with low conflict workloads, lower commit latency vs non-voting	No

Figure 2.6: Comparison between the approaches presented

Single Master

The simplicity of Single Master approaches is one of its strong points. However, its simplicity is the reason why SM approaches have limited scalability. This is especially noticeable in update-intensive workloads because the master is the solely responsible for processing update transactions. SM shines in workloads with a low percentage of updates, i.e. in read-intensive workloads, where any replica can answer. It is also suitable to handle very high contention workloads, due to the fact of just having one master. This eases the conflict handling and also the

scheduling of transactions. SM does not require any a priori knowledge of transaction's conflict classes. This approach is simple and effective in read-intensive and high contention workloads; however, has a low throughput in update-intensive workloads.

Two-Phase Commit

One of Two-Phase Commit main weaknesses is scalability, due to the fact the number of messages exchanged becoming excessive as the number of nodes increases. It also causes distributed deadlocks in high contention workloads. This happens when mutual blocking between transactions occur, and execution of those transactions is interrupted, and no completion can be reached. This problem may be solved using the knowledge of conflict classes, e.g. for implementing a scheduling system. However, that would add more messages exchange in a protocol that already has a high rate of messages.

State Machine Replication

State Machine Replication has some scalability limitations, like most solutions presented, due to the fact that any replica added to the system must execute all transactions, and so throughput does not increase with the number of replicas. Also, SMR parallelism can be affected when transactions have coarse-grained conflict classes that complicate the decision to execute transactions concurrently. So, in scenarios with high conflict workloads, SMR needs a fine-grained conflict class of transactions to be able to schedule them efficiently. This means that SMR requires a priori knowledge of conflict classes to work well. However, most solutions available require the developer to specify the transactions' conflict classes [32] or require a reconnaissance phase to determine the storage accesses [39]. This could result in faulty predictions, affecting the performance and even correctness of the system. We will address this issue in the current work, thus enabling significant improvements.

Voting and Non-Voting

The main problem of Certification solutions, including Voting and Non-Voting, is when transactions are later found to conflict and are required to abort. That is why both approaches are weak when subject to high conflict workloads. On the other hand, Voting and Non-Voting, scale very well when subject to low conflict workloads where the abort rate has a small impact on the execution. Both solutions behave similarly, however, the difference in implementations results in Voting solutions having a higher commit latency compared to Non-Voting. This is because the number of messages exchanged is higher in Voting approaches. Finally, Certification approaches

do not require a priori knowledge of transaction's conflict classes.

2.2 Transaction Scheduling

The goal of a Transaction Scheduler (TS), is to schedule transactions in a way that minimizes the occurrence of transaction aborts. The scheduling process can be done by taking into consideration the transactions' conflict classes, due to the fact that all transaction aborts, except induced by the application, occur when transactions have the same conflict classes. By judiciously scheduling transactions, the number of aborts will decrease, resulting in an increase of the processing rate.

The way transactions are scheduled can vary per solution. In the following sections, we will present implementations of TS included in Replicated Databases solutions and in Transactional Memory solutions, in Section 2.2.1 and Section 2.2.2, respectively.

2.2.1 Replicated Databases

Very little work has been done on TS on Replicated Databases. One weak point of all database replication approaches mentioned previously is that in some scenarios occurs a large number of aborts that highly impacts the system's throughput. Using TS techniques can drastically mitigate this issue. Next, we describe two approaches of TS in replicated databases, AKARA and AJITTS.

AKARA

AKARA [15] is a database replication protocol based on group communication. The goal is to maximize resource usage by scheduling sufficient concurrent executions. AKARA is constituted by 4 queues (Q0, Q2a, Q2b, Q2c). Transactions that arrive in queue Q0, are first classified in terms of its type (active or passive) and its conflict classes before being sent to Q2a. The type determined establishes if a transaction is actively or passively executed, i.e. execute with high priority or not. In the next queue Q2a, transactions wait to be scheduled. The scheduler analyses the queued transactions starting at the head and compares each transaction's conflict classes to the conflict classes of previous transactions. If a conflicting transaction is found, it waits for its turn. This allows to schedulle transactions to execute concurrently without occurring conflicts. Transactions that are ready to be executed first move from Q2a to Q2b, before starting execution. In queue Q2b, transactions may be aborted due to conflicts with a transaction in Q2b or Q2c. However, due to interleaving inside the database, a transaction t' ordered before a transaction t may be blocked by t. To overcome this, AKARA allows t to

overtake *t'* in the total order established by consensus, when both have the same conflict classes and belong to the same replica. Otherwise, it aborts *t*. After being executed, transactions are moved to Q2c, where they wait to be committed (or aborted). When ready to be committed, the transaction's modifications are sent to all other replicas and they are committed. If an abort occurs, the transaction is re-executed conservatively by imposing its priority on any locally running transaction.

Adding this scheduling process to a replicated database, drastically decrease the number of conflict occurrences while providing a satisfactory throughput.

AJITTS

AJITTS [31] is an adaptive just-in-time transaction scheduler. AJITTS is similar to AKARA but employs a different scheduling technique by having just one queue. AJITTS's goal is to decrease the number of aborts while increasing transaction throughput by computing the appropriate start time for each transaction. The idea behind AJITTS is that transactions are vulnerable to being aborted from the time execution starts until certification. So, in order to minimize the number of aborts, execution should start as late as possible. However, this can result in certification going idle due to the transaction in the head of the queue still being executed. Certification goes idle in these cases because the certification must occur in the order previously established by consensus. To mitigate this issue, AJITTS introduces a mark in the queue that determines which transactions should start execution - all transactions before this line are not eligible to start executing, while all transactions between the line and the head of the queue that are not yet being executed, are sent to be executed. The marker is changed every time a transaction leaves the queue, i.e. it is committed or aborted. The mark is determined by correlating the estimated execution time and the input size of transactions. This results in, transactions with a higher value of input being executed earlier than transactions with lower values. The number of transactions executed and not yet executed is also taken into consideration when determining the position of the mark. In other words, AJITTS algorithm determines the mark position, depending on the workload of the system. AJITTS keeps a record of all estimated and real values of transactions executed and certified to adapt the calculation of the mark position.

AJJITS, in the beginning, could have a poor performance related to having low throughput. This low throughput is due to the occurrence of aborts or due to having too much idle time between executions. But has the system estimation becomes more accurate, it starts reducing the number of aborts and improving the peak throughput, even if it throttles transaction execution. AJJITS could be improved by providing some specifications of the expected workload, to avoid

the poor performance in the beginning.

2.2.2 Transactional Memory

TS techniques have been widely used in Transaction Memory [19, 20, 18], with many proposed implementations over the years. Next, we present three solutions implemented in Transactional Memory, CAR-STM [19], Shrink [20] and Seer [18]. Due to the fact that the scope of this thesis is Replicated Databases, Transactional Memory falls outside of this thesis scope. Because of this, we will not go through the specifications of each solution implementation, we will only consider the TS techniques that each solution uses.

CAR-STM

CAR-STM [19] is a scheduling-based mechanism for software transactional memory (STM) which avoids transactional conflicts. CAR-STM is implemented in a centralized configuration, as this goes outside the scope of this work, we will analyse only the idea behind the scheduler. CAR-STM utilizes its scheduling capability in two different ways, with a contention manager called *serializing contention management* and a technique designated *proactive collision avoidance*. The first, *serializing contention management*, detects conflicts between transactions, originating from different queues, and aborts one transaction and moves it to the queue of the other transaction. This way, the scheduler avoids repeated collision of transactions. The second, *proactive collision avoidance*, allows CAR-STM to pre-assign transactions that are more likely to collide. Applications can provide information about transaction's collision-probability and CAR-STM can use this information to decide on which queue to put the new transaction. When a transaction arrives, it extracts the corresponding information, including the collision probability. Afterwards, a dispatcher, using the *proactive collision avoidance* with the transaction information, chooses which queue to send the transaction to. During execution, if a conflict occurs in one of the queues, the *serializing contention management* extracts one of the conflicting transaction and designates to a new queue with no conflicting transactions.

Incorporating CAR-STM into a transactional system greatly reduces the probability that a pair of colliding transactions would collide again. It also improves execution time and increases throughput while, at the same time, providing a more stable performance.

SHRINK

Shrink [20] is a scheduler that bases its prediction on the access patterns of past transactions from the same thread. It uses a novel heuristic, called *serialization affinity*, to schedule transactions

with a probability proportional to the current amount of contention. Shrink is based on two ideas: locality of reference and serialization affinity. In the first one is used a notion of *temporal locality* to predict transactional read sets. *Temporal locality* provides the frequently accesses of past transactions. Shrink uses this information to predict whether the same read accesses will occur in future transactions. To predict transactional write sets, Shrink uses a similar technique as to predict the read sets. However, instead of considering the past transactions, for the write sets, Shrink considers the transaction repetition. Shrink uses the predicted access sets (read and write), in conjunction with the information of the currently executing transactions, to prevent conflicts. The second idea, serialization affinity, allows to serialize threads only if contention is high. This means that Shrink only activates the prediction and serialization techniques when the success rate falls below a certain threshold. It does this by maintaining a success rate parameter for every execution queue. When a transaction begins, Shrink predicts its read and write sets and, if the success rate is high, executes transaction normally. When the success rate is low, the transaction is first sent to a scheduler where it waits until the locks corresponding to the predicted read and write sets are free. Afterwards, the transaction waiting can be executed, without incurring any conflicts.

Conflicts can still happen, however Shrink obtains roughly 70% accurate read and write accesses predictions, the rest is tolarated by the system.

SEER

Seer [18] is a scheduler that addresses Hardware Transaction Memory (HTM) restrictions by leveraging an inference technique that identifies the most likely conflict relations. With this, Seer establishes a dynamic locking scheme to serialize transactions in a fine-grained manner. This means that the scheduler works with imprecise information about the conflict causes due to limitations of HTM, whereas STM can give precise information about aborts, pinpointing which transaction caused the abort. With this in mind, the key idea of Seer is to: gather statistics to detect the set of concurrently active transactions upon abort and commit events. The statistics gathered are then used as input for an on-line inference technique that uses probabilistic arguments to identify conflict patterns between different atomic blocks of the program in a reliable way. The final step exploits the probabilistic knowledge of conflict relations to synthesize a fine-grained dynamic locking scheme that serializes transactions to avoid the occurrence of conflicts. Seer keeps a lock table with every transaction locks. With this table, Seer identifies which transactions can be executed concurrently. Every time a transaction finishes executioning, either because it committed or aborted, the list of active transactions is analysed and stored in

two matrices one for commits and one for aborts. The commit matrix tracks the frequency of commit events for a transaction and list which transactions were active. The abort matrix is equal but instead tracks information about abort events. These matrices are merged and are used to calculate and update the locking scheme to reduce aborts of transactions. The challenge is identifying, among all captured conflicts, which ones occur frequently enough to benefit from throttling down concurrency.

2.2.3 Summary

Transaction Scheduling can have a positive impact on a transactional system, by allowing better concurrency and reducing the chance of aborts. However, as seen in the section 2.1, the implementation of a TS can have a negative impacts on the system's scalability and throughput.

All the schedulers analysed in this document require or would benefit from having knowledge of the transactions' conflict classes. This description needs to be done a priori by the developer or at runtime by predicting the transaction's conflict classes. In the former, the description given could suffer from errors by the developer. In the latter, the description obtained ends up being too coarse-grained to schedule transactions efficiently. This thesis adresses this problem by determining a priori and in a fine-grained manner the transactions' conflict classes via Symbolic Execution. This greatly improves the efficiency of transaction schedulers.

2.3 Symbolic Execution

Symbolic Execution (SE) is a program analysis technique first introduced by King in [26]. It is traditionally used for software testing and debugging, as it allows to check whether a program has errors, such as null pointers, memory leaks, or if some property can be violated, e.g. unauthorized acquisition of privileges.

SE uses symbolic variables, i.e. variables that abstract their concrete value, to construct a path condition, i.e. boolean expressions that unequivocally identify the constraints associated with each path. SE achieves this by constructing a tree that represents the program execution. Figure 2.7 represents a SE tree based in the *foobar* method shown in Listing 2.1. The tree's root is the first condition of the method. Then the root splits into two paths, one where the condition verifies and other where it does not. Afterwards, each of these two paths are explored and whenever another conditional statement is found, the execution is splited again.

```
void foobar(int a, int b) {
    if (a != 0)
        return 1;
    else if (b == 0)
        return 0;
    else
        return -1;
}
```

Listing 2.1: Simple Java method

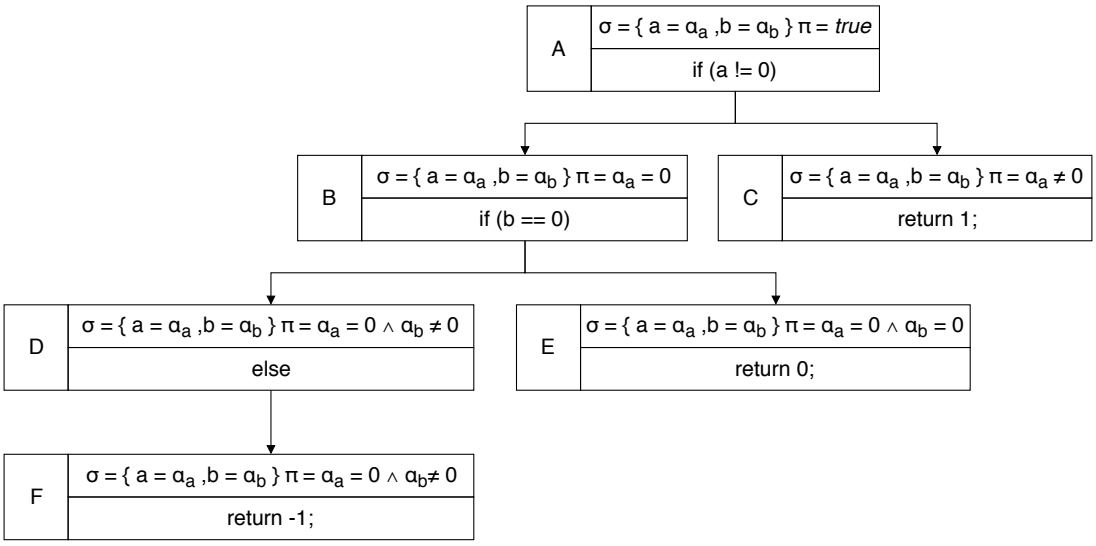


Figure 2.7: Symbolic execution tree based on the method given in Listing 2.1

These conditional statements are represented as (*smt*, σ, π), as shown in Baldoni et al. [10], where:

- *stmt* - is the statement to evaluate, e.g. *if (a != 0)* the root's statement in Figure 2.7.
- σ - represents the current state of the program variables. This can include expressions over concrete values or symbolic values, represented as α_i where i symbolizes the variable.
- π - denotes the path constraints. Path constraint is an expression of a set of symbolic variables states to reach *stmt*, e.g. to satisfy a condition.

The execution tree can include, in addition to the execution paths and the corresponding conditional statements, the state changes of symbolic variables, e.g. when is assigned a different symbolic value to a variable. With this, is possible to get a very fine-grained information about every execution path of the program.

Section 2.3.1 identify some known limitations and challenges of SE. Section 2.3.2 presents some solutions to these limitations. Section 2.3.3 briefly describes Concrete and Concolic execution and their advantages. Finally, Section 2.3.4, goes through some use cases of SE.

2.3.1 Limitations and Challenges

SE was designed following a number of performance-related design principles, the most notable were depicted by Baldoni et al. [10]:

- *Progress* - the executor must be able to make forward progress for an arbitrarily long time without exceeding the given resources.
- *Work repetition* - no execution work should be repeated.
- *Analysis reuse* - analysis results from previous runs should be reused as much as possible.

The *progress* principle is related to hardware limitations, that could affect the program's analysis time. *Work repetition* and *analysis reuse* principles are simple optimizations to spare resources (time and hardware). The survey of Baldoni et al. [10] listed some challenges inherent from the SE state of the art:

- *Memory* - The amount of memory required to analyse a program is a challenge by itself. Another big challenge is to handle and simulate memory behaviour, such as pointers, arrays, or other complex objects present in the programs being analysed.
- *Loops* - The existence of loops in the program makes the symbolic engine fork in multiple branches to accommodate the number of paths in the loop. This leads to an explosion of paths that the SE engine has to explore, incurring in performance penalties, as well as increasing the amount of memory needed. This could result in an incomplete program analysis.
- *Constraint Solver* - Constraint solvers suffer from many limitations. The more prevalent is execution time, that increases exponentially as the number of constraints increases, or when complex constraints are added to be solved.

The following section will address some solutions to these challenges.

2.3.2 Improvements and Solutions

This section describes some solutions to the challenges previously described, related to *memory*, *loops*, *path explosion* and *constraint solvers*. Afterwards, will be presented two SE tools, KLEE [13] and JPF [2, 33, 7], that brought many of the improvements listed next.

Memory

One of the main challenges of memory is related to the way pointers are handled by the symbolic engine. One possibility is to consider the memory addresses as *fully symbolic memory*, where a symbolic address could reference any position in memory. This approach in complex programs can end up being intractable because of the sizable number of positions available. For this reason, *fully symbolic memory* is best suited for the cases where the set of possible memory addresses to be referenced is small. Another option is to use *address concretization*, that consists of a symbolic address referencing a single specific address. This reduces the number of states, but in exchange, some execution information could be missed, such as execution paths. *Partial memory modeling* mitigates the scalability problems of *fully symbolic memory* and the loss of soundness of *address concretization*. The main idea of this technique is to use the best parts of both previously described techniques, where write addresses are always concrete and read addresses are modelled symbolically.

Handling complex objects is another critical challenge related to memory. Because these objects must be allocated in such a way that is possible to take advantage of the object properties. A possible approach is to only initialize them when the objects are accessed. An object can be initialized with various values: (1) null, (2) a reference to a new object with all symbolic attributes, and (3) a previously introduced concrete object of the desired type. This approach, does not require an a priori bound on the number of input objects. They are bound when accessed, avoiding acquiring resources for useless objects. This can be extended using *method preconditions*, where input objects states are characterized externally by their intended behaviour. This is especially useful in custom objects not supported by the SE engine.

Loops

The common solution to counteract the loops challenges is to only compute part of the loop, by limiting the number of iterations. This increases speed in exchange for soundness, as a lot of information can be lost. Other approaches infer loop invariants through static analysis and use them to merge equivalent states. This technique allows to have a more complete analyse in exchange of speed, as this would take more time but spare some hardware resources.

Path Explosion

A program that contains loops increases exponentially the number of execution states. This typically results in a Path Explosion. Path Explosion can also occur in complex programs, regardless of loops. To mitigate this problem, for both cases, the only solution is to restrict the

SE engine to explore just a fraction of all possible execution paths. There are several approaches that perform this, where they try to identify and explore all the important execution paths. Two implementations that do this are: *depth-first search* (DFS) and *breadth-first search* (BFS). DFS continuously expands a path as much as possible, before backtracking to the deepest unexplored branch. BFS explores all unexplored paths in parallel, repeatedly expanding each of them by a fixed slice. DFS is more suited to programs that do not have loops, due to the fact that, paths containing loops and recursive calls can easily stall the execution. BFS is best for programs containing loops, where we end up with a larger path coverage and overcomes the staleness problem.

An alternative technique is to use a method called *preconditioned symbolic execution* that drives a symbolic execution towards a subset of inputs space. The subset is determined by some predefined conditions. *Preconditioned symbolic execution* allows to narrow the exploration space, resulting in a more efficient exploration of the execution paths. These preconditions need to be carefully selected because if it is too specific, the subset could be too small or even empty. On the other hand, if the preconditions are too general, almost the entire paths will need to be explored.

There are many other techniques to mitigate the Path Explosion problem. All the ones mentioned previously have a common goal, of only exploring a fraction of all execution paths. What varies between techniques is the exploration length. Some try to explore more paths, but the information gathered from each path might not be enough. Others analyse thoroughly some paths, gathering a more complete information of the path, however, the number of paths explored is lower.

Constraint Solver

Constraint solvers are one of the main obstacles to the scalability of SE engines. There are many approaches that try to minimize this problem such as Z3 [17] and Choco [24]. The more prominent solutions are the *constraint reduction* and the *reuse of constraint solutions*. The first one consists of reducing the size and complexity of the constraints to evaluate. This is achieved by dividing a complex constraint into simpler ones, and running these simpler constraints concurrently. In some cases, it is possible to rewrite a new constraint in an already existing one. Regarding *reuse of constraint solutions*, the idea is speeding up the computation of results by reusing previously computed results. The results are stored in a cache to avoid calling the solver unnecessarily.

The best results would be achieved by combining both approaches, e.g. when dividing a complex constraint into simpler constraints, or rewriting it to a similar simpler constraint, these simpler constraints could be cached. With this combination the solver will end up being called fewer times, resulting in a lower execution time.

KLEE

KLEE [13] is a symbolic execution tool proposed by Cadar et al., that is capable of automatically generating tests that achieve high coverage on a diverse set of complex programs. KLEE was designed with C/C++ programs in mind and it has been highly tested. KLEE introduced many of the innovations and improvements mentioned previously. KLEE is considered an *online executor* because it executes multiple paths simultaneously in a single run. This requires a careful attention to memory consumption because the number of active states could be too high. KLEE uses caching techniques that allow to never re-execute instructions, this information improves the time of analyse of the following states. To mitigate the memory consumption problem, KLEE uses the *fully symbolic memory* approach, mentioned previously, where symbolic addresses are referenced in any position of memory.

KLEE brought many improvements to the path explosion problem. The main one is an implementation of *breadth-first search*, where it assignes a probability to each path based on the length and on the branch variety. With this information, it is possible to decide which paths to prioritize the exploration first. Regarding the constraint solver improvements, KLEE implemented some techniques that were mentioned previously. One is an implementation of *reuse of constraint solutions* where the constraint processing is reduced by simplifying the expressions using previously obtained results. For example, an equality constraint form x := 5 could be simplified by other constraints that use x, such as x<0.

KLEE improved and optimized many of SE challenges by introducing new techniques and approaches.

Java PathFinder

JPF [2] is an open-source runtime environment for verifying Java bytecode that was developed by NASA. JPF is composed by a core, designated as *jpf-core*, that can be easily extended. *Jpf-core* is a program which receives Java programs to find possible errors in them. One of JPF extensions, and the one that we will use, is Symbolic JPF [33, 7], also designated as *jpf-symbc*. *Jpf-symbc* is a framework that integrates SE with the existing model checking of *jpf-core*. *Jpf-symbc* generates test cases that obtain a high code coverage. Programs are executed using

symbolic inputs that represent all possible concrete inputs. Values of variables are represented as numeric constraints, generated from analysis of the code structure. These constraints are then solved to generate inputs that are guaranteed to reach that part of the code. JPF supports a big number of constraints solvers, including Z3 [17] and Choco [24] that were mentioned previously. This makes JPF, in specific *jpf-symbc*, a very complete SE tool to analyse Java programs.

2.3.3 Concrete and Concolic Execution

To avoid some of SE's limitations, there are alternative approaches such as Concrete and Concolic executions. These work similarly as SE, with the main difference being in the way that variables are handled. Concrete Execution uses only concrete values in the variables' states. SE uses symbolic values that represent the state of the variable, this state can be just the variable itself, an expression, such as α+2, or a path constraint to satisfy a given condition, like α>0. Finally, Concolic Execution can use both concrete and symbolic variables.

Next, we will briefly discuss these two execution techniques, Concrete and Concolic Execution [10], and then we will present CUTE [37], a unit testing tool that introduced many Concolic techniques now present in many other tools.

Concrete Execution

Concrete Execution uses concrete values to explore the execution flow of a program, concrete values are switched and tested until it is found a value that generates a new execution flow. These values are then stored in the path constraint (π) of the corresponding statement (*stmt*). In complex programs, this type of execution technique can result in a large number of executions due to the huge amount of possible input values. This means that Concrete Execution has space (hardware) and time limitations, resulting in execution paths not being explored. Because of this, Concrete Execution allows to set properties, in order to prioritize the exploration of a specific path flow. This kind of execution is best suited for test cases, where the amount of inputs possible are known and limited.

Concolic Execution

Concolic Execution, stands for cooperative Concrete and Symbolic execution. It was first introduced by Godefroid and Sen with DART [22] and was extended afterwards by Sen and Marinov with CUTE [37]. Concolic Execution combines the advantages of Concrete and Symbolic execution. These advantages are: (1) the benefit of using concrete values to specify a variable state and (2) the extensibility of symbolic values that allow a better and more complete exploration

of a program. Concolic Execution is also affected by some of SE's limitations. However, it is not affected by the weaknesses of Concrete Execution because they are complemented by the SE.

CUTE

CUTE [37] is a unit testing engine that use Concolic Execution techniques. CUTE is available in C and in Java with jCUTE [1]. The combination of Symbolic and Concrete execution generates test inputs that allow to explore all feasible execution paths. Before, such an extensive exploration would require a long time and powerful hardware. CUTE also introduced a technique to avoid redundant input testing, i.e. input that would give a previously seen result. With this technique, it is more likely to test critical inputs that could cause incorrect behaviour of the program.

CUTE provides a method for representing and solving approximate pointer constraints to generate test inputs. By using a logical input map, with all inputs, it is possible to build constraints on these inputs, by symbolically executing the code being tested. CUTE also makes the expressions on pointers simpler, by having only one value, instead of one field for each value, allowing to execute a larger number of unit tests.

With CUTE, nearly every unit test will cover every branch of a program in an efficient way. However, it cannot test concurrent programs and programs using algebraic functions, such as cryptographic protocols, due to the large number of possible solutions.

2.3.4 Use Cases

As seen in previous sections SE is mostly used for testing and debugging software. Below we present some interesting use cases of symbolic execution.

SAFELI

SAFELI [21] is a tool for detecting SQL Injection vulnerabilities in Web applications. SAFELI instruments the bytecode of Java Web applications and utilizes symbolic execution to statically inspect security vulnerabilities. SAFELI detects SQL injection attacks by symbolic executing the code to find for critical spots which submit SQL query. When one of these critical spots are found, SAFELI constructs a hybrid string equation to explore values that might be used to apply a SQL injection attack. Once the equation is successfully solved, the solution of the equation is used to construct a test case. An attack pattern library is used to apply attacks in these tests. This library is the weak point of SAFELI because it needs to be manually updated. The authors want, in the future, to not have an attack pattern library at all, making SAFELI

completely autonomous. This can be made possible by tackling the string constraints issues discussed in the paper.

Other implementations

Marcozzi et. al [30] proposes an algorithm that use SE, to generate tests for simple Java methods. These tests execute reads and writes via SQL transactions to a relational database, subject to integrity constraints. In this approach, the authors use a Relational SE technique that allows the algorithm to generate a set of relational constraints for any finite path to test. The solutions to these constraints constitute a test case, that exercise the selected path. To achieve this, the algorithm receives as input a SQL dynamic-link library (or DLL) file that describes the database scheme and receives a Java method to test. The algorithm outputs the relational constraints generated, which are then sent to an analyser that solves the constraints in order to find structures that satisfy them. The algorithm transforms the given relational constraints into a set of boolean constraints. The advantages of this approach compared to other approaches, is that there is no need to transform the original program code.

2.3.5 Summary

Despite some limitations, SE is a very powerful testing tool. Although SE is used for testing purposes, there are other interesting use cases in the security and conflict handling domains, such as SAFELY [21] and Marcozzi et. al [30], respectively. Similarly to these approaches, which used SE in domains other than testing, we will do the same in this dissertation. We use SE to obtain a priori knowledge of the transactions reads and writes sets and consequently optimize the way that concurrent transactions are handled in a fully replicated database systems.

Chapter 3

Symbolic-SMR

The analysis of the state of the art conducted in chapter 2 highlighted that replicated databases still incur several limitations. The main limitation that replicated databases have is related to their parallelism capabilities. SMR approaches require determinism to take advantage of multi-core machines, i.e. to execute transactions in parallel. One way to solve this limitation is to determine which transactions can be executed concurrently without incurring in conflicts. State of the art solutions achieve this by classifying the conflict classes of transactions, allowing to determine which transactions conflicts. However, state of the art solutions, such as Nodo [32], determine the transactions' conflict classes with a large granularity, which limits the degree of parallelism. Other solutions, such as Calvin [39], require the developers to provide the transactions' conflict classes which is a notoriously hard task with complex programs. However, Calvin has an alternative method to determine the transactions' conflict classes, it does this by performing a reconnaissance phase. With this, Calvin pre-executes (without acquiring locks and without performing write operations) the transaction to identify the items that the transaction accesses. However, this has a large cost in the system performance and if the conflict classes are mispredicted the transaction is then aborted and this process needs to be done again.

Based on this analysis, this dissertation proposes the idea of using SE, presented in Section 2.3, to analyse the transactions' logic and extract information about the items that are accessed. We do not only extract the items accessed but also, the path conditions to reach those accesses. This information would then be used to build a solid and fine-grained scheduler that allows to have concurrent executions without any conflicts and therefore, no aborts.

Next, in Section 3.1, we introduce an overview of the overall solution where each of the components are presented. Afterwards, in Section 3.2, we describe in great detail the two main components of the solution, the SE engine and the Symbolic-SMR. Finally, in Section 3.3, we discuss some arguments related to the correctness and determinism of the presented solution.

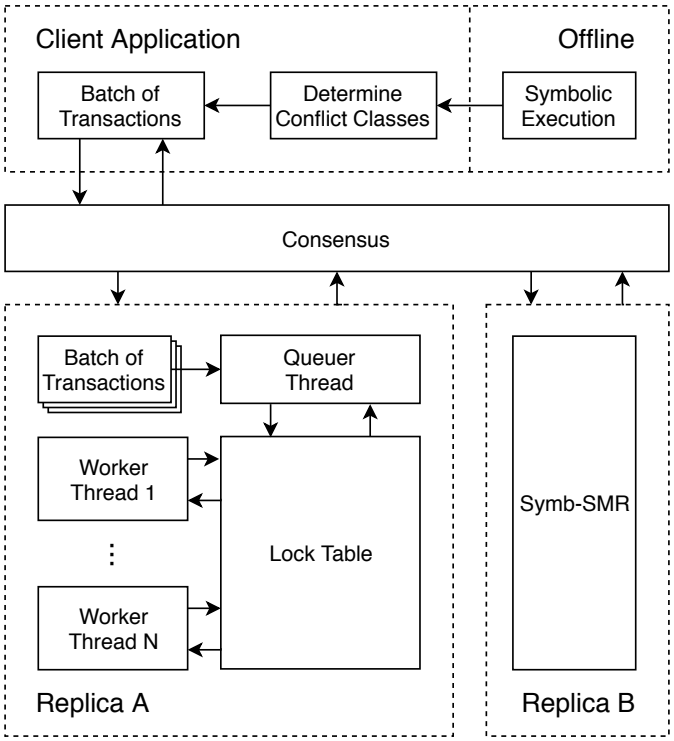


Figure 3.1: Symb-SMR Overview

3.1 Overview

Symb-SMR has two main modules, depicted in Figure 3.1. One of the modules is the client application and the other is the system replicas.

The client application is responsible for generating transaction requests and send them to the replicas where then, the replicas reach a consensus on the order to execute the transactions. The Client also attaches the conflict classes of each transaction it generates. It does this based on the output given by the SE after analysing the transaction code. However, the code is analysed by SE offline. Thus, the output produced by SE is not dependent on the concrete values of the transaction's inputs, as the values are symbolically annotated during the analysis performed by the SE engine. The generated output encompasses both: (1) boolean restrictions upon every path condition, which is typically the functionality employed by state of the art SE engines to give the set of restrictions associated to each of the programs control flow; (2) symbolic variables manipulated by the transactions, which when instantiated represent a fine-grained representation of the transaction conflict classes. Thus, at run-time, the Client only needs to replace the corresponding input's symbolic variable for its concrete value to obtain the conflict classes of that transaction.

The replicas process every batch of transactions that is delivered. The replicas must process each batch in a way that their state is maintained consistent. In single-core system, where we

would only have one Worker thread, i.e. thread that executes transactions, determinism would not be a concern. However, in a multi-core system, with various Workers, it must exist a control on which transactions the Workers can execute. This control is important due to conflicts between transactions, but also to avoid replicas having distinct commit orders which results in different replicas' states. To do this, without exchanging any messages between replicas, we need to schedule the transactions deterministically. So, the scheduling process is done just by one thread, the Queuer Thread, it extracts a batch of transactions and schedules each transaction of the batch by placing them in the Lock Table according to the conflict classes, so that two transactions belonging to the same conflict classes are serialized. The Lock Table is the structure that dictates the order that transactions will execute, to avoid conflicting transactions being executed concurrently. The Queuer, while scheduling of a transaction, might need to perform read operations in the database. This occurs when a transaction has accesses that depend on read accesses done during its execution. Similarly to the existing literature, we designate this type of transactions as Indirect Transactions (IT). Transactions without any accesses dependencies are called Direct Transactions (DT). We have also categorized another type of transactions, the Read-Only Transactions (ROT), that only have read accesses. DT and IT are allocated in the Lock Table during scheduling, but ROT are not. This is because the order by which ROT are executed is not relevant. Because of this, we decided to not place ROT in the Lock Table and allow them to be executed concurrently with each other when there are no update transactions being executed. The Workers only execute the transactions that have successfully acquired all the locks. After executing a transaction, the Worker removes the finished transaction from the Lock Table and updates the list of transactions that are ready to be executed. Before executing IT, the Worker must verify if the values read during scheduling did not change. It does this by performing the same reads that were done during the transaction scheduling and comparing them with the values read previously. If the values are equal, then the transaction may execute as normal. But if the values differ, then the transaction is removed from the Lock Table and placed in a queue of failed transactions without being executed. This is necessary because, if the values read are different, this means that the scheduling process is incorrect. This could result in conflicting transactions being executed concurrently or breaking the execution order and thus leading to replica divergence. Therefore, after all transactions are executed, the failed transactions are re-scheduled again by the Queuer Thread and re-executed by the Worker Threads. This process is repeated until all failed transactions are executed successfully committed.

3.2 Detailed Description

In this section, we provide a detailed description of the solution, explaining, in detail how each component works. First, in Section 3.2.1, we will detail how we obtain the conflict classes of transactions using Symbolic Execution. Then, in Section 3.2.2, we will describe the Symbolic-SMR that include the previously mentioned Lock Table, Queuer Thread and Worker Threads.

3.2.1 Symbolic Execution

Symbolic Execution is a program analysis tool, as described in Section 2.3. The number of SE engines available is quite large and each offers different capabilities. The one we choose to use was JPF [33] with the Symbolic JPF extension [7] that was previously presented. The reasons behind this choice were: (1) it is a SE engine for Java, (2) it is open-source, has constant updates and has a large number of users and (3) offers developers a complete API to use and modify the SE engine internals to accommodate their needs. The latter is a very important reason because we needed to customize the SE analysis to give the information about the items accessed by transactions. To achieve this, we used one of the API objects of JPF, the Listeners.

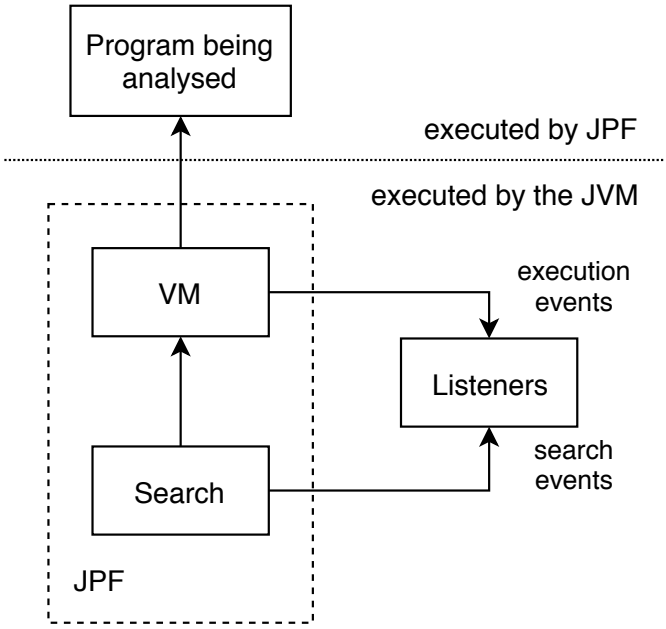


Figure 3.2: Scheme of JPF listener based on [3]

Listeners allow to execute code when specific events occur during program analysis. Figure 3.2 shows an overview of how Listeners work and is based on a scheme presented in [3]. During a program analysis, JPF notifies the Listeners of Java execution events, such as instruction executed, object created and method invoked, and also notifies of JPF search events, such as new execution path and assignments to symbolic variables. This allows to identify specific points

of a transaction execution to gather crucial information. More specifically, we gather information about the items accessed by a transaction and the path that leads to each access. To achieve this we implemented a custom Listener, the *SymbolicDBListener*, which will be described next.

SymbolicDBListener Logic

The main role of *SymbolicDBListener* is to identify and collect the storage accesses in a transaction. To do this, we implemented a method that every time it receives an instruction executed event, checks if the instruction was a method invocation, more specifically a *put* or *get* method invocation. If it is, we extract from the method arguments, the table and key accessed. As the inside analysis of the put and get methods' are not relevant, we decided to skip the execution of these methods, avoid iterating with the database. This is done by jumping directly to the return call of the method. Then we add the arguments extracted from the method to the corresponding execution path, which describes the conditional statement to reach this path. The key value, and possibly, the table value gathered from each access could include symbolic variables that make up the table or key value, for example:

$$table\_x = 1 \ ; \ key\_y = input\_1 + input\_2 \tag{3.1}$$

This means that an item in table 1 with the key *input_1* + *input_2*, where *input_1* and *input_2* are symbolic variables, was accessed. At the end of the analysis, we end up with an output like the one represented in Listing 3.1. This output is later analysed by the Client to determine the conflict classes of a transaction, by exchanging the inputs' symbolic variable for the concrete values.

```
Transaction : <transaction_name>

Path : <path_condition>
        Read Set  : [<table_value> <key_value>, ...]
        Write Set : [<table_value> <key_value>, ...]
...
```

Listing 3.1: Example of the output given by JPF

As described previously, Indirect transactions have item accesses that depend on read operations done before. These types of accesses are also identified by the *SymbolicDBListener*, by generating a new symbolic variable, called read variable, every time a read operation (*get* method) is analysed. The read variable is then stored together with the access values (table and

key) of the corresponding read operation. Afterwards, every time the access values of a item access are identified, we verify if any of the symbolic variables included in the access values are read variables. If yes, then the table and key values of the read variable are included in the values of the access being analysed, for example:

$$table\_x = 1 \ ; \ key\_y = input\_1 + read\_variable\ (read\_variable\_table, read\_variable\_key). \quad (3.2)$$

This means that this item is in table 1 and its key depends on the sum of the symbolic variable *input_1* and a previous read operation done in table *read_variable_table* with the key *read_variable_key*. This kind of item accesses, designated as indirect accesses, must be handled differently by the client. When the client determines the conflict classes of a transaction, it cannot determine the conflict classes corresponding to indirect accesses. In other approaches, such as Calvin [39], the client resolves these problems by performing remote reads to solve the indirect accesses. However, in high-latency scenarios, i.e. scenarios where transaction submission and transaction execution are sufficiently spaced, this greatly augments the probability that such reads are stale by the time the transaction executes, hence increasing the abort rate. To avoid this, we store the indirect reads separately and execute them during the scheduling process at each replica. This way, in our approach, the vulnerability window is only from the scheduling phase to the execution phase, wheres in other approaches it is from client submission until the execution phase. As the latter includes the round-trip between the client and the server, together with the latency induced by consensus, the vulnerability window in our approach is significantly smaller.

During the implementation we encounter some problems. One of the problems was related to transactions with loops. We were getting two results when analysing these transactions: (1) the analysis running indefinitely and (2) obtaining repeated read and write sets values resulted from different execution paths. The latter was not incorrect, but it difficulted the output analysis. To mitigate this issue we changed the state exploration of JPF to breadth-first instead of depth-first, so when analysing a given if-statement, if we identify that no symbolic variable is accessed in that path then we add this if-statement to an *irrelevantBranch* set. This set is then used in the next iteration of the loop to identify the branches that were determined as irrelevant in previous iterations of the loop. If a branch is identified as irrelevant then, we would say to JPF to only take one of the branches of the if-statement without adding a new path constraint to the analysis. With this, we end up with a more quick and efficient analysis of the transactions. However with big loops the analysis still runs indefinitely.

3.2.2 Symbolic-SMR

After the client has generated the transactions and determined the transactions' conflict classes, the transactions and the respective conflict classes, are compiled in a batch. This batch is then ordered by consensus and then deterministically processed by the replicas following the consensus order. Determinism is only a concern in parallel systems because single-threaded systems return always the same result for the same workloads. Replicas with parallel systems need to be deterministic to maintain an equal state between them.

3.2.3 Overview

Executing transactions deterministically requires scheduling the transactions in a way that the resulting state after executing the transactions is equal throughout all replicas. To achieve this, it is necessary to take special attention to conflicting transactions, because if replicas execute these transactions in different orders, the end result for each replica will be different. To prevent this from happening, we implemented a Lock Table to perform the concurrency control of transaction execution. The transactions are inserted in the Lock Table by the Queuer Thread, during scheduling. The Lock Table controls the concurrency between transactions by each transaction's conflict classes. Figure 3.3 shows an example of how the Lock Table controls the concurrency of transactions Tx1, Tx2 and Tx3. The transactions are ordered in the batch as Tx1, Tx2 and Tx3 and their conflict classes are the following:

- Tx1 is table X - key 1, and table Y - key 1;
- Tx2 is table X - keys 1 and 6, and table Y - key 3
- Tx3 is table Y - key 1, and table Y - key 3;

The Queue thread will process the batch and populate the Lock Table as depicted in Figure 3.3. For a transaction to start execution, it needs to be in the head of all queues where it was inserted. As an example, following Figure 3.3, Tx1 can be executed, because is the first in both queue [T=X,K=1] and queue [T=Y,K=1]. This means that none of Tx1's conflicting transactions, Tx2 and Tx3, can be executed until Tx1 has finished. This concurrency control mechanism is very similar to the one of NODO [32], presented previously in Section 2.1. After Tx1 finishes, the transaction is removed from the Lock Table and Tx2 can start executing, while Tx3 still has to wait. At any given instant there are several instances of the Lock Table, identified by a unique ID. This allows the scheduling of future batches and the execution of the current batch to be made in different Lock Table instances. Using only one Lock Table would result in contention between the Queuer and Worker threads thus reducing performance. So, the Workers are only

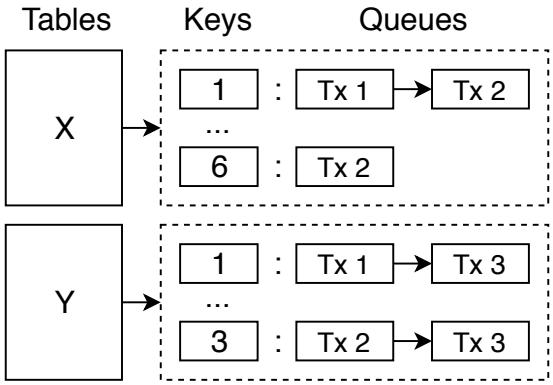


Figure 3.3: Example of the Lock Table organization

allowed to start execution in a Lock Table when the Queuer as finished scheduling in that Lock Table. Lastly, the Lock Table also includes some data structures that allow:

- to store the batch to be scheduled by the Queuer Thread;
- to store the transactions that are ready to be executed by the Worker Threads;
- and to control what transactions are being executed.

Next, we will present and describe the pseudocodes of the Queuer and the Worker Threads. Algorithm 1 depicts the data structure that both the Queuer and the Worker use. *BatchedToQueue* is a queue containing the batches delivered from consensus. *LockTable* is a data structure that controls the concurrency of transactions. *TransactionsToExecute* and *ROTransactionsToExecute* are queues that contain update and read-only transactions, respectively, that are ready to be executed. *TransactionsExecuting* is a list that contains the transactions that are currently being executed by the Workers Threads. FailedTransactions is a queue of transactions that failed to execute. Lastly, KVStore is the underlying key-value store.

```
1 initially :
2     BatchesToQueue = ... // Queue of batches to schedule
3     LockTable = ... // Lock Table data structure
4     TransactionsToExecute = ... // Queue of update transactions to execute
5     ROTransactionsToExecute = ... // Queue of read-only transactions to execute
6     TransactionsExecuting = ... // List of transactions executing
7     FailedTransactions = ... // Queue of transactions that failed
8     KVStore = ... // Key-Value store
```

Algorithm 1: Initialization of Lock Table and other data structures used by the Queuer and the Workers threads.

Queuer Thread

The Queuer Thread role is to schedule the transactions by inserting them in the Lock Table. Then, the Queuer determines if the transaction scheduled is ready for execution. Algorithm 2 describes the main logic behind the Queuer. The Queuer is always working while there are transactions to schedule. In every cycle, a new batch is polled to be scheduled. When there are no transactions to schedule (batch or failed transactions), the Queuer goes idle while it waits for work. When there is, the Queuer first checks if there are any failed transactions to schedule. This verification is done throughout the Queuer implementation to prioritize the scheduling of failed transactions. This is because the Workers cannot start executing a new batch before completing the current one. So, we want to schedule the failed transactions with the highest priority to quickly complete executing the current batch. The schedule of transactions is done by the function *QueueTransactions*, described in Algorithm 4 that will be described below. After checking and, possibly, scheduling the failed transactions it is time to schedule the batches. The Queuer polls a batch stored in the Lock Table to be scheduled. If there is no batch to schedule, it returns the value *null*. If the batch polled is not *null*, the batch is scheduled in the method *QueueBatch*, that receives a batch. This is described in Algorithm 3 which we analyse next.

```
1  upon Queuer start :
2      while not stop do
3          if no batches or failed transactions to schedule then
4              Wait()
5          if exist failed transactions to schedule then
6              failed_lock_table_id = FailedTransactions.getLockTableID()
7              QueueTransactions(FailedTransactions, failed_lock_table_id)
8          batch_to_queue = BatchesToQueue.poll()
9          if batch_to_queue not null then
10             QueueBatch(batch_to_queue)
11     end
```

Algorithm 2: Queuer's logic

```
1  Funtion QueueBatch(batch_to_queue) :
2      lock_table_id = batch_to_queue.getLockTableID()
3      while has_batch_to_queue do
4          if batch_to_queue has Read-Only transactions then
5              transactions_queue = batch_to_queue.pollReadOnlyTransactions()
6          else if batch_to_queue has Direct transactions then
7              transactions_queue = batch_to_queue.pollDirectTransactions()
8          else if batch_to_queue has Indirect transactions then
9              WaitForWorkers()
10             transactions_queue = batch_to_queue.pollIndirectTransactions()
11         else
12             has_batch_to_queue = false // exit loop
13         end
14         if exist failed transactions to schedule then
15             failed_lock_table_id = FailedTransactions.getLockTableID()
16             QueueTransactions(FailedTransactions, failed_lock_table_id)
17         QueueTransactions(transactions_queue, lock_table_id)
18         if transactions_queue are Read-Only transactions then
19             SignalWorkersToExecuteReadOnlyTransactions()
20     end
21     SignalWorkersToExecuteUpdateTransactions()
```

Algorithm 3: Function that schedules a batch of transactions

The *QueueBatch* function receives a batch to schedule. The batch, has mentioned before, includes the transactions to execute and their conflict classes. The transactions are subdivided into three subqueues, a queue of Direct transactions (DT), a queue for Indirect Transactions (IT) and a queue for Read-Only Transactions (ROT). When the Lock Table receives a batch, it is assigned a Lock Table ID to the batch. This ID is extracted in the function *QueueBatch*, and identifies the Lock Table where transactions will be scheduled. The order that the batches are scheduled is always the same. First is the ROT, then the IT and finally the DT. Regarding the IT, before these transactions can be scheduled, the Queuer must wait for the Workers to finish executing. This is because these transactions need to perform read operations to solve their conflict classes. Thus, the Queuer has to guarantee that these reads are not done concurrently with write operations to get the most recent values. The batch is then completely scheduled when the three queues are all empty. In each cycle of function *QueueBatch*, it is extracted

a different subqueue of the transaction to schedule. Before proceeding with the scheduling of the transaction's subqueue (via the function *QueueTransactions*) is done a verification if there are any failed transactions to be scheduled. The Queuer has the responsibility to notify the Workers that they have transactions to execute. This is done in two phases: first the Workers are notified to execute ROT and second, when the batch is completely scheduled the Queuer notifies the Workers to start executing the update transactions (DT and IT). This is done because the ROT can start being executed as soon as they are scheduled because they do not need concurrency control. The only concurrency control between transactions in this situation is between ROT and update transactions. Next, will be analysed the pseudocode of the function *QueueTransactions* that is described in Algorithm 4.

```
1  Funtion QueueTransactions(transaction_queue, lock_table_id) :
2      foreach transaction ∈ transaction_queue do
3          if transaction is Read-Only then
4              ROTransactionsToExecute.add(transaction)
5              continue
6          if transaction is Indirect then
7              foreach indirect_entry ∈ transaction.getIndirectEntries() do
8                  foreach entry_to_read ∈ indirect_entry.getEntriesToRead() do
9                      entry_table = entry_to_read.getTable()
10                     entry_key = entry_to_read.getKey()
11                     entry_row = entry_to_read.getRow()
12                     value_read = KeyValueStore.get(entry_table, entry_key, entry_row)
13                     indirect_entry.addValueRead(entry_to_read, value_read)
14                 end
15                 transaction.addNewLockTableEntry(indirect_entry)
16             end
17         foreach lock_table_entries ∈ transaction.getLockTableEntries() do
18             table = lock_table_entries.getTable()
19             key = lock_table_entries.getKey()
20             queue = LockTable[lock_table_id][table].get(key)
21             queue.add(transaction)
22             if transaction is first in queue then
23                 transaction.acquireLock()
24         end
25         number_of_locks_to_acquire = transaction.getLocksToAcquire()
26         if number_of_locks_to_acquire equal to 0 then
27             TransactionsToExecute[lock_table_id].add(transaction)
28     end
```

Algorithm 4: Function that schedules the transactions, populates the Lock Table and determines what transactions can be executed

Algorithm 4 describes the pseudocode of function *QueueTransactions*. This function receives as inputs the queue of transactions to schedule and the *lock_table_id* where the transactions will

be scheduled. In each cycle, a new transaction *Tx* is polled from the transaction_queue. If *Tx* is a ROT, it does not need to be inserted in the Lock Table and it is only added to the list of ROT ready to be executed, *ROTransactionsToExecute*. If *Tx* is an update transaction, then we verify if *Tx* needs to perform some reads before being scheduled, i.e. if *Tx* is an IT. If it is, then the Queuer goes through all indirect variables that *Tx* has. For each indirect variable, we perform a read for every read variable included. We extract the table, the key and the row of the read variable. Then the value is read and stored in the indirect variable. After having all the values for every read variable, we determine the conflict classes (or Lock Table entries) of the indirect variable. Next, the conflict classes are inserted in the Lock Table. For each conflict classes, we extract the corresponding table and key. Then we add the transaction to the queue corresponding to the extracted table and key of the conflict class. Next, we check if the transaction inserted is the first in queue. If it is, then the number of locks to acquire is decremented. A lock for a transaction is considered acquired when that transaction is the first in queue. The number of locks to acquire is represented by a counter that is decremented until it reaches 0, this means that all locks for a transaction are acquired and the transaction is ready to be executed. This verification is done after having inserted all the conflict classes, where the number of locks to acquired is checked. If its equal to zero, then the transaction is ready to be executed and is added to the queue *TransactionsToExecute*. These steps are performed until every transaction is scheduled.

Worker Threads

The Worker Threads are responsible for executing the transactions that are in the queues *TransactionsToExecute* and *ROTransactionsToExecute*. As described previously, the ROT can start being executed before the update transactions. This allows the Workers to execute ROT while they wait for the Queuer to finish scheduling the rest of the update transactions.

Algorithm 5 describes the overall logic of a Worker Thread. The Worker threads are kept busy as long as there are transactions to execute. After executing a batch, the Workers check if any IT failed during execution. An IT fails when, the values read during scheduling and the current values differ. This means the values changed since scheduling and the transaction cannot be executed safely. This transaction is then added to the queue of *FailedTransactions*, to be later scheduled and executed again. So, if there are failed transactions, one Worker waits for all Workers to finish executing and then signals the Queuer to schedule the failed transactions. While the Workers wait for the failed transactions to be scheduled, they can execute ROT that have been already scheduled for the next batch.. When there are no failed transactions to

schedule, the Workers go for the next batch. If there is no batch to execute, the Workers wait and execute ROT. If there is a batch to execute, then the Workers get the Lock Table ID that was assigned to the batch and start executing. Before executing any (update) transaction, the Workers must wait for any Worker that is still executing ROT. The Workers are allowed to start executing update transactions only when every Worker has stopped executing ROT. The transactions to execute are polled from the queue *TransactionsToExecute* and are executed in function *ExecuteTransaction*. This function, described in Algorithm 6, executes the transaction and removes it from the Lock Table (with the corresponding Lock Table ID) after the transaction has completed executing. This process is repeated until all transactions of the batch are executed. Then the Worker finishes executing the rest of the ROT. Before executing ROT, is necessary to verify if there are Workers still executing update transactions, to avoid concurrents reads and writes. Next, we analyse function *ExecuteTransaction*, described in Algorithm 6.

```
1  upon Worker start :
2      while not stop do
3          if the batch is completely executed then
4              if the batch has failed transactions then
5                  if lock is acquired by a Worker then
6                      WaitForAllWorkers()
7                      SignalQueuerToScheduleFailedTransactions()
8                  else
9                      WaitAndExecuteReadOnlyTransactions()
10                 end
11             else
12                 if no batch to execute then
13                     WaitAndExecuteReadOnlyTransactions()
14                 lock_table_id = LockTable.getLockTableIDOfBatchToExecute()
15             end
16         WaitForReaders()
17         foreach transaction ∈ TransactionsToExecute[lock_table_id]) do
18             ExecuteTransaction(transaction, lock_table_id)
19         end
20         ExecuteReadTransactions()
21     end
```

Algorithm 5: Worker overall logic

The function *ExecuteTransaction*, Algorithm 6, receives as input, a transaction and a Lock Table ID. The task of this function is to execute the transaction and remove it from the Lock Table. But before starting executing, if the transaction is IT then the values read during scheduling must be verified. This is done by performing the same reads and comparing the new values with the previous ones. If any value differs, then the transaction is considered as failed and it is not executed. Afterwards, the transactions that have not failed are executed. When the transaction has completed executing, it is removed from the Lock Table. In this process, the Worker polls the transaction from the Lock Table queues corresponding to the transaction's conflict classes. After the transaction is polled from a queue, the Worker decrements the lock counter of the next transaction in queue. If all locks of the next transaction are acquired, then that transaction is added to the queue *TransactionsToExecute*. ROT do not need to do this process because those transactions are not included in the Lock Table. Lastly, the transaction

is removed from the list *TransactionsExecuting*.

```
1  Funtion ExecuteTransaction(transaction, lock_table_id) :
2      if transaction is Indirect then
3          failure = VerifyIndirectValuesRead(transaction)
4          if failure then
5              transaction.failed()
6      if transaction not failed then
7          ExecuteTransaction(transaction)
8      if transaction not Read-Only then
9          foreach lock_table_entries ∈ transaction.getLockTableEntries() do
10             table = lock_table_entries.getTable()
11             key = lock_table_entries.getKey()
12             queue = LockTable[lock_table_id][table].get(key)
13             queue.poll()
14             if queue not empty then
15                 next_transaction = queue.getHead()
16                 number_of_locks_to_acquire = next_transaction.acquireLock()
17                 if number_of_locks_to_acquire equal to 0 then
18                     TransactionsToExecute[lock_table_id].add(next_transaction)
19         end
20     TransactionsExecuting.remove(transaction)
```

Algorithm 6: Transaction execution and removal from the Lock Table

3.3 Correctness Argument

In this section, we sketch a correctness argument for the proposed solution.

The common concern behind the design of every algorithm of this solution is if the implementation is deterministic. The Queuer primary task is to schedule the transactions deterministically across all replicas because the batch of transactions is ordered by consensus and there is only one thread processing the batch. This means that after processing a batch, and before starting execution, the Lock Table of all replicas is the same. The other big concern was, winding up with the same state after concurrently executing a batch. This is addressed by the Lock Table concurrency control mechanism. The Lock Table controls the order by which transactions are executed and which transactions can be concurrently executed. So, the final state of each replica will be the same. Finally, the last big concern is the IT. These transactions, when in

large number, are very likely to fail. The re-scheduling of these transactions needs to be done following the same order. However, the order that the transactions fail is not always the same when being concurrently executed. So, this order cannot be used for scheduling these transactions, due to the fact of not being deterministic. To solve this problem, we order the failed transactions by their ID. As transactions reach a consensus on the order of transactions in a batch, it will be given the same IDs to the transactions. Also, for each batch, we know that the set of failed transaction is the same, even if they don't fail in the same order. So, by re-ordering the failed transactions by their ID, we guarantee that they are re-schedule in the same order in all replicas.

Chapter 4

Results

This chapter presents the experimental evaluation of the Symb-SMR. The experiments done focus on evaluating the performance of the overall system, including the Queuer and Worker threads, but also to identify potential bottlenecks that the system may have.

To evaluate the Symb-SMR's performance we use a No Contention micro-benchmark and the TPC-C benchmark [8]. The No Contention micro-benchmark, Section 4.2, generates a non-conflicting workload of transactions. The goal is to evaluate the scalability of Symb-SMR and to identify possible bottlenecks that affect the solution. The TPC-C benchmark, Section 4.3, generates real-world workloads to evaluate the overall performance of Symb-SMR. Symb-SMR will be compared with two state of the art approaches, Nodo [32] and Calvin [39]. This comparison will be done using the TPC-C benchmarks where each solution will process the same workload.

In the next section, we will present the platform and evaluation metrics used in the experimental evaluation.

4.1 Platform and Evaluation Metrics

Symb-SMR will be evaluated via the following metrics: (1) throughput of the system and (2) number of times transactions fail. With the throughput, we measure the number of transactions that are processed per second. In the fail rate, we will measure the number of times transactions fail until being executed successfully. This measurement will be dependent on the workload being processed, in particular to the percentage of indirect transactions included in the workload. The solution will be evaluated with two different workloads, one generated from the no contention micro-benchmark, and the other with the TPC-C benchmark [8]. We will also use TPC-C to compare Symb-SMR with other previously presented state of the art solutions, Nodo [32] and

Calvin [39]. In particular, we will compare Calvin's handling mechanism for transactions that fail against the algorithm of Symb-SMR. With the No Contention micro-benchmark, we want to evaluate the scalability of the system and identify possible bottlenecks, in particular in the Queuer and Workers threads.

All the experiment results presented were obtained on a machine with the following specs: an Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz processor with 2 sockets, connected with UMA, 14 physical cores per socket, where each core can execute 2 hardware threads with hyperthreading [27]. The operating system is Ubuntu 16.04.3 and uses Java version 1.8.0_171.

In the experiments, we did various measures of the solutions with different numbers of Workers, between 1 and 55. Some of these values have some important particularities:

- 1 worker - demonstrates the overhead of Symb-SMR versus the baseline of processing the workload sequentially;
- 4 and 8 workers - are the typical number of Workers which state of the art solutions can scale;
- 13 workers - the same number of threads (Queuer + Workers) as the number of physical cores in a socket;
- 27 workers - the same number of threads (Queuer + Workers) as the number of cores in both sockets or with hyperthreading in just one socket;
- 55 workers - the same number of threads (Queuer + Workers) as the maximum number of cores with hyperthreading in both sockets

The underlying database that we used is RocksDB [6], a persistent key-value store developed by Facebook. RocksDB's library is implemented in C++ but it provides a Java API to interact with the database. Although, RocksDB supports transactions operations, for these experiments we disabled these features to increase performance but also, because Symb-SMR already works as a concurrency control system, so there is no need for RocksDB's own concurrency control.

For these experiments we do not determine at runtime the transactions' conflict classes. Instead, we use a pre-determined structure (based on the output given by JPF) that contains the transactions conflict classes. We do this, to specifically evaluate the performance of Symb-SMR without any external overheads.

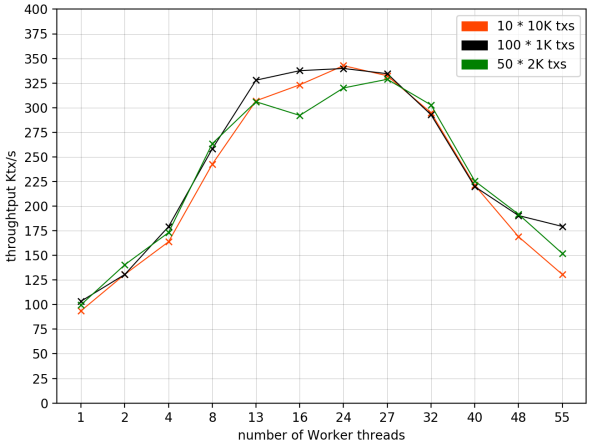
4.2 No Contention Micro-Benchmark

This micro-benchmark generates workloads with no conflicting transactions. Each transaction generated by the micro-benchmark will access a different line of a table resulting in transactions not conflicting. With this we want to test the performance of the solution in scenarios with no concurrency constraints, i.e. without conflicts between transactions, meaning that every transaction can be executed concurrently. With this, we will evaluate the scalability of the solution and identify possible bottlenecks that might affect Symb-SMR's performance.

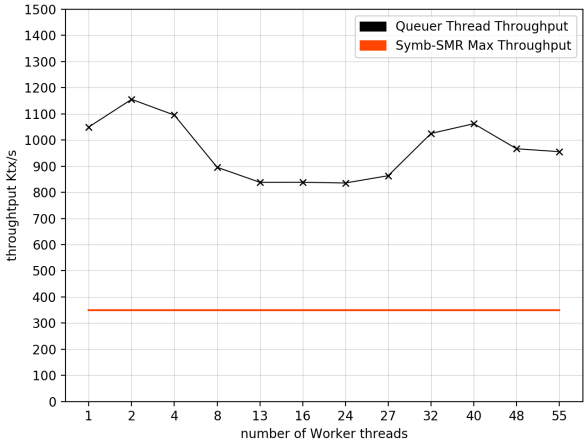
4.2.1 Experiment Results

In this experiment, we generate various No Contention workloads with 100 000 transactions. We generate 10 batches of 10 000 transactions, 100 batches of 1 000 transactions and 50 batches of 2 000 transactions. Theoretically, in scenarios without conflicting transactions Symb-SMR's throughput should scale relative to the number of Workers threads. Figure 4.1 (a) shows the results for the above configurations with a varying number of Worker threads. Symb-SMR scales well up to 13 Workers. Then between 13 and 27 Workers the throughput is roughly constant at approximately 340 000 transactions per second. Afterwards the throughput decreases to approximately 175 000 transactions per second. There are 2 possible causes for this scalability limitation: (1) the Queuer throughput is not high enough and (2) contention between Worker threads. The first possibility is disproved by Figure 4.1 (b) which it shows the Queuer Thread throughput and the max throughput achieved by Symb-SMR. Although, the Queuer thread throughput is not constant, it is much higher than the max throughput of Symb-SMR. The second possibility is analysed in Figure 4.2 that profiles the Worker threads execution by the percentage of time: processing transactions, waiting for the Queuer thread to schedule transactions and extracting a new transaction to execute from the *TransactionsToExecute* queue. Figure 4.2 (a) shows the results for 10 batches of 10 000 transactions, Figure 4.2 (b) shows the results for 100 batches of 1 000 transactions and Figure 4.2 (c) shows the results for 50 batches of 2 000 transactions. As we can see, the percentage of processing time decreases in all scenarios when the number of Workers increases. Whereas the processing percentage decreases the percentage of time where transactions are extracting a transaction to execute increases. This is because all Workers extract transactions to execute from a single *TransactionsToExecute* queue. This impacts the performance because the queue controls concurrent accesses, resulting in Workers having to wait to extract a transaction from this queue. Thus the performance of the system is limited by the contention on this queue. However, as we will see in the TPC-C benchmark, which contains more complex transactional logic this limit is not reached.

The Symb-SMR's throughput achieved in these experiments were very similar. However, we observed a slightly higher throughput in the workload with the lowest batch size, the workload with 100 batches of 1 000 transactions.

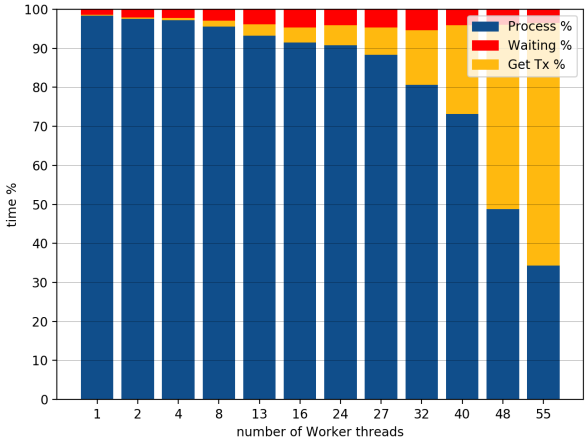


(a) Symb-SMR's throughput

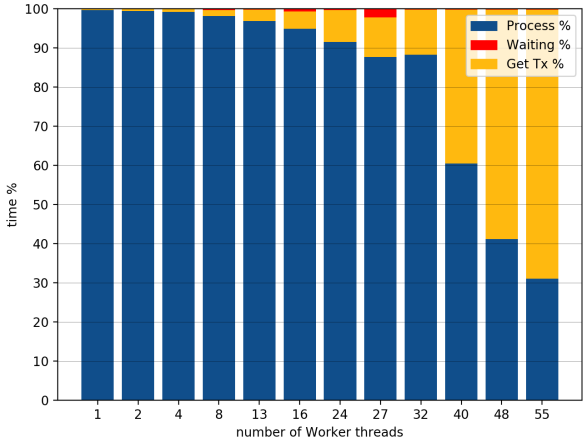


(b) Queuer throughput and Symb-SMR's max throughput

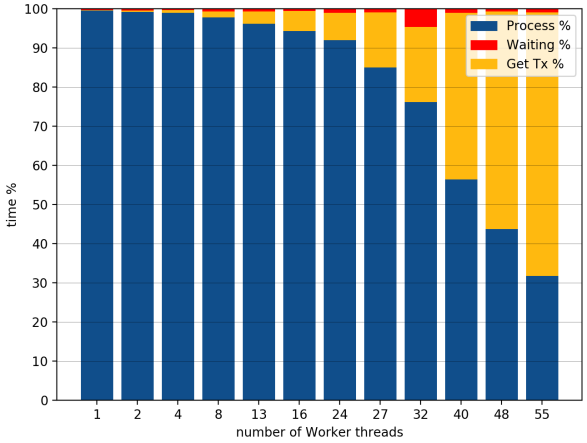
Figure 4.1: No contention workload of 100 000 transactions



(a) 10 batches of 10 000 transactions



(b) 100 batches of 1 000 transactions



(c) 50 batches of 2 000 transactions

Figure 4.2: Breakdown of Symb-SMR's Workers when processing batches with different sizes

4.3 TPC-C Benchmark

TPC-C [8] is a well-known benchmark, that is widely used to evaluate transactional systems. TPC-C is composed of five transactions, three update transactions, New Order, Payment and Delivery, and two read-only transactions, Status Order and Stock Level. Two of the updates transactions, New Order and Delivery, are Indirect transactions (IT) because they contain item accesses that depend on other previous accesses. These two transactions will allow to evaluate the performance of the mechanism for handling failed transactions of Symb-SMR and Calvin. The TPC-C benchmark accesses 9 tables. However, the more critical table is the Warehouse table, that is accessed by every transaction. The TPC-C standard size of the Warehouse table is 10, where each warehouse has 10 districts and each district has 3 000 customers. Since every TPC-C transaction depends on a Warehouse the amount of conflicting transaction will depend on the size of the Warehouse table. Thus, we did experiments with 55 warehouses, equal to the max number of Worker threads used in these experiments. With this, we will evaluate the scalability of Symb-SMR when faced with real-world workloads and the amount of failed transactions occurrences.

Before presenting the results we will describe the key implementational differences of Nodo and Calvin compared to Symb-SMR

4.3.1 Nodo

Nodo [32] is a concurrency control system, that was described in Section 2.1.5. Symb-SMR implementation was largely based on the Nodo idea of controlling the concurrency between transactions based on their conflict classes. However, the big difference between both solutions is that: (1) Nodo requires developers to provide the conflict classes of the transactions, whereas we use Symbolic Execution to obtain that and (2) Nodo's scheduling is very coarse-grained because it only considers the tables that are accessed in a transaction. This limits the level of parallelism of the system, where only transactions that access different tables are allowed to be concurrently executed. Symb-SMR controls the concurrency of transactions based on the tables and keys that are accessed. This results in more transactions being allowed to execute concurrently increasing the system parallelism.

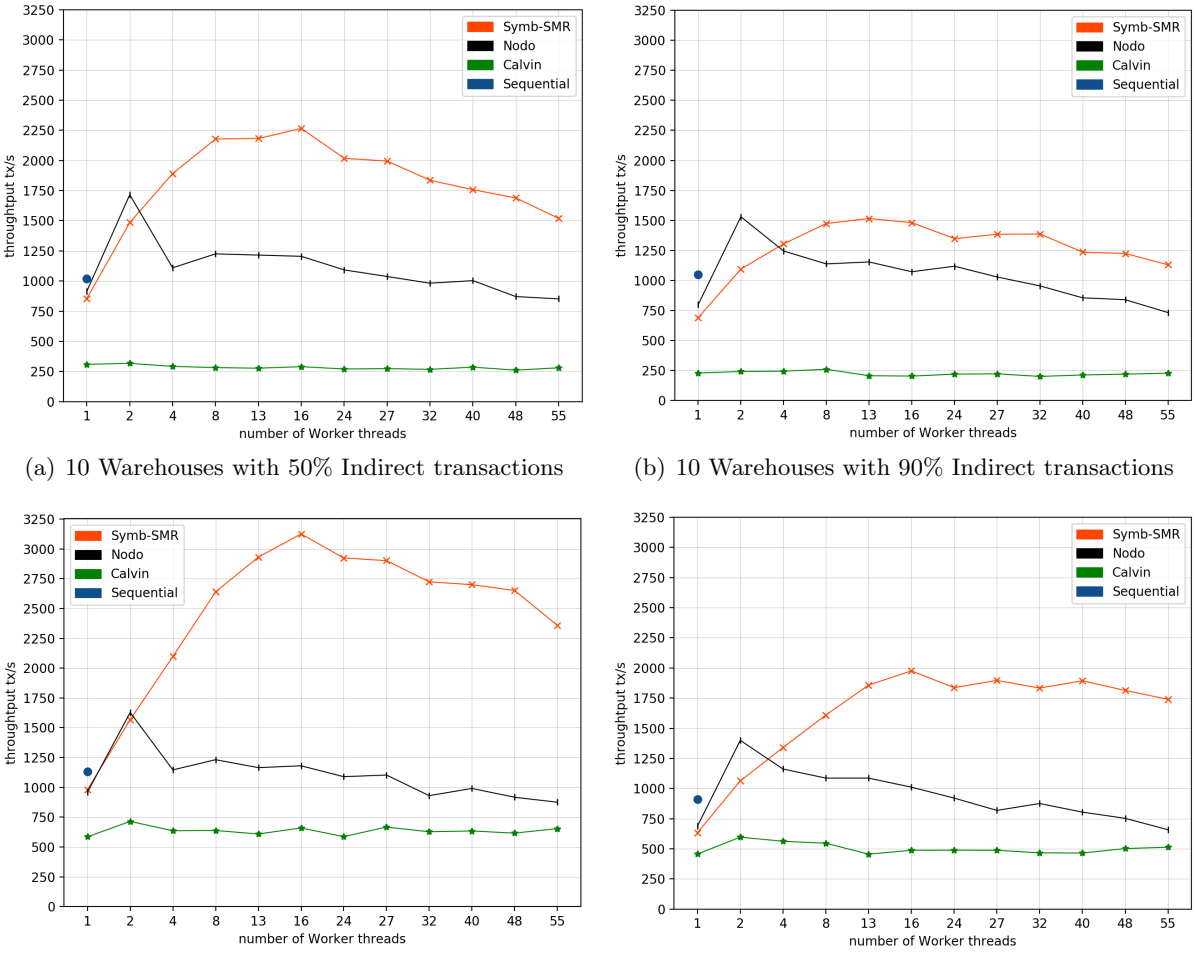
4.3.2 Calvin

Calvin [39] is a transaction scheduler, that was presented in Section 2.1.5. The way Calvin and Symb-SMR work is very similar. However, the main differences are: (1) the way conflict classes of transactions are determined, (2) how failed transactions are handled and (3) the scheduling of

Read-Only transactions (ROT). Calvin has two ways for determining the transactions' conflict classes. One way, similar to Nodo, requires developers to provide the conflict classes of transactions. However, this can be too complex because Calvin requires a fine-grained description of the conflict classes (table and key) which is unrealistic for large complex applications. As a result, Calvin has an alternative way of determining the conflict classes. It does a reconnaissance phase to obtain the transactions' read and write set. The reconnaissance phase of transactions is done by executing the transactions without acquiring any locks and without executing any write operations. The problem with this is that the reconnaissance phase is done in the client resulting in a considerable interval of time where the read and write set can be changed due to concurrent updates in the database. As a result, if a transaction's conflict classes change before the transaction is committed, then the transaction is incorrectly scheduled and will end up aborting. This occurs especially with IT, where transactions' read and write set depend on other read and write set. Symb-SMR mitigates this problem by solving the Indirect transactions during scheduling and when there are no write operations in the database. This reduces the chance of the determined conflict classes to be changed. When transactions abort, Calvin sends these transactions back to the client. It is then, the client responsibility to re-do the reconnaissance phase and to re-send the transactions. Symb-SMR does not abort any transactions that fail, instead, we aggregate these failed transactions and re-schedule them to be executed right away. In the end, this will result in a lower number of failed transactions. The last main implementation difference is that Calvin does not differentiates ROT from the other transactions and schedule them as regular update transactions. We choose to separate ROT because they can all be executed concurrently due to these transactions only performing read operations. However, ROT and update transactions cannot be executed in parallel due to possible conflicts in concurrent read and write operations. To avoid this, we do not allow to execute ROT concurrently with update transactions. The advantage of not scheduling ROT is that this allows to execute these transactions when the Workers are waiting for update transactions to be scheduled. This way, we avoid the costs of scheduling ROT and we reduce the chance of the Workers threads going idle.

4.3.3 Experiment Results

For these experiments, we generated TPC-C workloads of 100 batches with 100 transactions. Figure 4.3 depicts the results of the solutions throughput for processing these workloads. In these plots, we compare Symb-SMR with three other solutions, Sequential, Nodo and Calvin. The Sequential implementation works as a baseline to measure the impact of each solution. In



(a) 10 Warehouses with 50% Indirect transactions

(b) 10 Warehouses with 90% Indirect transactions

(c) 55 Warehouses with 50% Indirect transactions

(d) 55 Warehouses with 90% Indirect transactions

Figure 4.3: Comparison of Symb-SMR vs Sequential vs Nodo vs Calvin with a TPC-C workload with 10 and 55 warehouses with 50% and 90% of Indirect Transactions

Figure 4.3 (a) and Figure 4.3 (b) are the solutions' throughputs resulted from processing a TPC-C workload when the size of Warehouse table is 10. The difference between these two plots is in the percentage of IT wherein Figure 4.3 (a) approximately 50% of the transactions are Indirect transactions and in Figure 4.3 (b) 90% are IT. We experimented with different percentages of IT to measure the impact that these transactions have in the system performance. Symb-SMR, with 50% Indirect transactions scales until 16 Worker threads and then deeps where with 90% of Indirect transactions the solution does not scale, not surpassing the 1500 transactions per second. As we can see, the percentage of Indirect transactions have a big impact in Symb-SMR's performance, that was expected. Since Nodo does not suffer from Indirect transactions, its throughput is not affected by the change in percentages. Figure 4.3 (c) and Figure 4.3 (d) have the same respective changes in the number of Indirect transactions with the difference being the size of the Warehouse table, that in these cases is 55. As we can, Symb-SMR scalability, in Figure 4.3 (c), increased compared to Figure 4.3 (a). This is due to TPC-C transactions

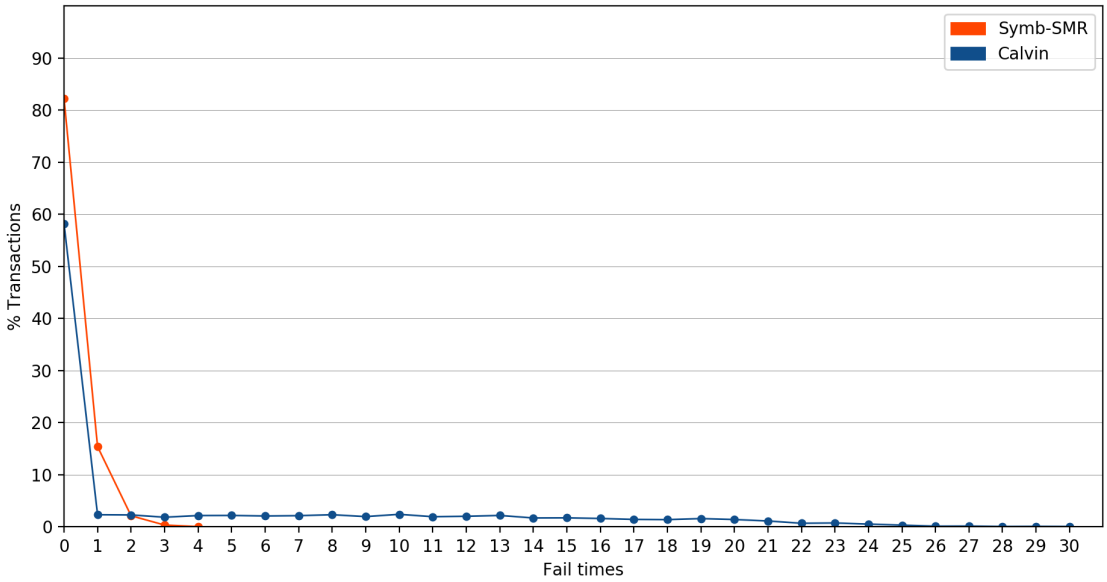
conflicting less when the size of the Warehouses table increases. Since Nodo only considers the tables to control the concurrency, an increase in the table size will not impact the performance as we can observe in all plots. Also, the number of Workers do not benefit Nodo, in fact, increasing the number of Worker threads slightly decreases the throughput of Nodo. Calvin's performance in all these scenarios is poor comparing to the others. The main bottleneck of Calvin is the reconnaissance phase and the vulnerability window of the transactions' conflict classes. Performing the reconnaissance phase has already huge costs in the system throughput but the costs increase even more due to transactions aborting and requiring to do the reconnaissance phase again. Especially in these scenarios, where we have 50% and 90% of Indirect transactions, the number of abort transactions will be very high. This will be further analysed next.

Failed Transactions

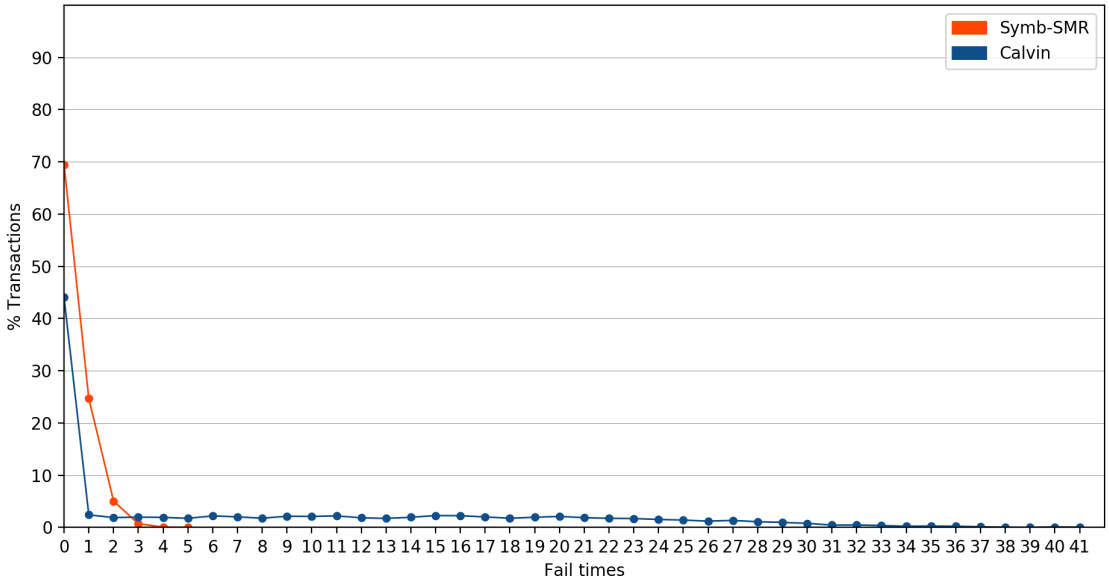
In these experiments we compare Symb-SMR with Calvin by measuring the number of times transactions fail until they are successfully executed. Nodo is not included in these experiments because is not impacted by IT resulting in transactions not failing due to changes in conflict classes. Figure 4.4 show the number of times transactions failed during the processing of a TPC-C workload. Each point represents the percentage of transactions that have failed x times. The scenarios are the same than previously, Figure 4.4 (a) and Figure 4.4 (b) are with 10 Warehouses, 50% and 90% of Indirect transactions respectively. Figure 4.4 (c) and Figure 4.4 (d) are with 55 Warehouses with the same changes in Indirect transactions respectively. In Figure 4.4 (a) and Figure 4.4 (b), we can see the main advantage of Symb-SMR compared to Calvin. Symb-SMR achieves a higher percentage of transactions that executed successfully without failures and the overall number of failures is lower compared to Calvin. Symb-SMR by solving Indirect transactions during scheduling and when there are no update operations being made in the database, allows to lower the chance of Indirect transactions to fail due to changes in the database state. Calvin using the reconnaissance phase in the client, with possible concurrent operations in the database, results on the determined conflict classes being possible already incorrect in the moment of scheduling. As expected, with a higher number of warehouses both solutions will have less failed transactions but Symb-SMR successfully executes over 90% of all transactions without failures whereas Calvin achieves 70% at best.

4.4 Summary

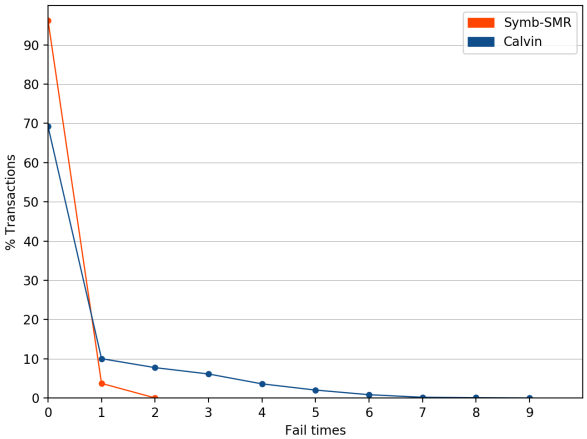
In the first experiment we analysed Symb-SMR's scalability and max throughput using the no contention micro-benchmark. The workloads generated did not have conflicting transactions



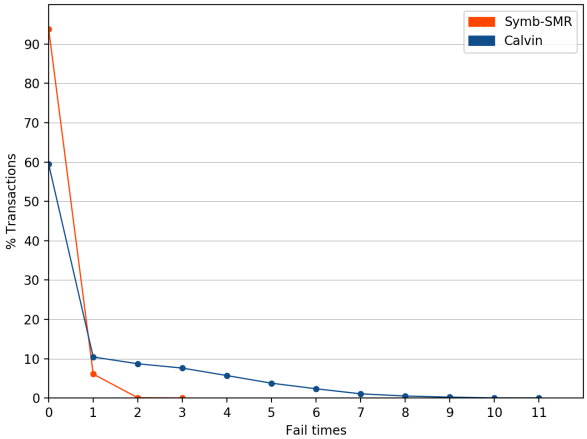
(a) 10 Warehouses with 50% of Indirect transactions



(b) 10 Warehouses with 90% of Indirect transactions



(c) 55 Warehouses with 50% of Indirect transactions



(d) 55 Warehouses with 90% of Indirect transactions

Figure 4.4: Percentage of failures per transaction for a TPC-C workload with 10 and 55 Warehouses and with 50% and 90% of Indirect transactions: Symb-SMR vs Calvin

resulting in all transactions being allowed to execute concurrently. The results obtained showed that with a high number of Worker threads (from 27 Workers) Symb-SMR throughput begins to decrease. The performance of Symb-SMR, in these scenarios, is limited due to the contention in the *TransactionsToExecute* queue, that every Worker accesses. In the experiments with the TPC-C benchmark we observed that this bottleneck was not that impactful. However, in these scenarios Symb-SMR only scaled until 16 Workers and then the throughput begins to slightly decrease. Although these scalability limitations, Symb-SMR overall throughput was 2 times higher than Nodo and approximately 7 times higher than Calvin. In the last experiments, with the failed transactions, Symb-SMR executes successfully over 80% of transactions without failures whereas Calvin, in the same scenario, only executes successfully at first time less than 60% of transactions. Overall, Symb-SMR's performance surpasses the performance of current state of the art approaches in every scenario that was analysed. This shows that Symb-SMR is a viable option to control the concurrency of transactions in SMR-based systems.

Chapter 5

Conclusions

The main goal of this dissertation is to increase the level of parallelism of a SMR-based system and to mitigate the limitations that impact these type of systems. We do this by using Symbolic Execution to determine a priori and in a fine-grained manner the items that transactions access to control the concurrency between them. This is all done autonomously without requiring anything to the developers, unlike other state of the art solutions that require developers to provide the items accessed by the transactions. Symb-SMR uses the item accesses determined by SE to build an efficient and deterministic concurrency control system that provides a higher level of parallelism comparing to other state of the art solutions, like Nodo and Calvin. This is proven in the experiments done where Symb-SMR throughput results surpass the results of the other state of the art solutions by 2 and 7 times.

5.1 Future Work

Symbolic Execution and JPF Symbolic Execution has an immense power which we only used a portion. There is much room for improvement in this part. First, we need to improve the way we handle loops because transactions with big loops, that have many iterations, are not yet supported. All the experiments done were using benchmarks that only used primitive variables, e.g. integer, long and boolean, due to the current limitations with strings support. Databases that do not use strings is not realistic, so it important to include the support of strings.

JPF Output Analysis The determination of transactions conflict classes, based on the output given by JPF, is not efficient. This is due to using strings to identify and determine the transactions' conflict. To improve this, we need to implement a system that generates in real time objects that contain the transactions conflict classes, this must be done efficiently to have as much low impact in the performance.

Optimizations to Symb-SMR The experiments done showed that Symb-SMR's performance varies significantly when processing various batches with different number of transactions. The solution scalability also has some room for improvement, by solving the contention problem when with a large number of Worker threads.

Bibliography

[1] jCUTE. http://osl.cs.illinois.edu/software/jcute/. Accessed: 2018-10-15.

[2] JPF - Java Path Finder. https://github.com/javapathfinder/jpf-core. Accessed: 2018-10-15.

[3] JPF Listeners. https://github.com/javapathfinder/jpf-core/wiki/Listeners. Accessed: 2018-10-15.

[4] MySQL Replication. https://dev.mysql.com/doc/refman/5.7/en/replication.html. Accessed: 2018-10-15.

[5] PostgreSQL. https://www.postgresql.org/. Accessed: 2018-10-15.

[6] RocksDB. https://rocksdb.org/. Accessed: 2018-10-15.

[7] Symbolic JPF. https://github.com/SymbolicPathFinder/jpf-symbc. Accessed: 2018-10-15.

[8] TPC-C. http://www.tpc.org/tpcc/default.asp. Accessed: 2018-10-15.

[9] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):159–174, October 2007.

[10] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[11] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[12] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Chapter 8: The primary backup approach, 1993.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[14] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[15] A. Correia, J. Pereira, and Rui Oliveira. AKARA: A flexible clustering protocol for demanding transactional workloads. In *On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*, pages 691–708, 2008.

[16] Maria Couceiro, Paolo Romano, and Luís Rodrigues. Chapter 2: Towards autonomic transactional replication for cloud environments, 2012.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Nuno Diegues, Paolo Romano, and Stoyan Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 224–233, New York, NY, USA, 2015. ACM.

[19] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 125–134, New York, NY, USA, 2008. ACM.

[20] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 7–16, New York, NY, USA, 2009. ACM.

[21] Xiang Fu and Kai Qian. Safeli: Sql injection scanner using symbolic execution. In *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*, TAV-WEB '08, pages 34–39, New York, NY, USA, 2008. ACM.

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[23] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, June 1996.

[24] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, Paris, France, France, 2008.

[25] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, Hollywood, CA, 2012. USENIX.

[26] James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233, New York, NY, USA, 1975. ACM.

[27] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[28] P. J. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 454–465, June 2011.

[29] Parisa Jalili Marandi and Fernando Pedone. Optimistic parallel state-machine replication. *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 57–66, 2014.

[30] M. Marcozzi, W. Vanhoof, and J. L. Hainaut. A relational symbolic execution algorithm for constraint-based testing of database programs. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 179–188, Sept 2013.

[31] Ana Nunes, Rui Oliveira, and José Pereira. *AJITTS: Adaptive Just-In-Time Transaction Scheduling*, pages 57–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[32] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. *Scalable Replication in Database Clusters*, pages 315–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[33] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and

system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[34] Pedro Raminhas, Miguel Matos, and Paolo Romano. Fine-grained transaction scheduling in replicated databases via symbolic execution, 04 2018.

[35] Luís Rodrigues, Hugo Miranda, Ricardo Almeida, João Martins, and Pedro Vicente. *The GlobData Fault-Tolerant Replicated Distributed Object Database*, pages 426–433. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

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

[37] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[38] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[39] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[40] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.