Fine-grained Transaction Scheduling in Replicated Databases via Symbolic Execution

Miguel Viegas
miguel.viegas@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
Advisors: Prof. Paolo Romano
Prof. Miguel Matos

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 to ensure this is State Machine Replication (SMR), which requires that operations are deterministic. In order to ensure this, have been proposed schemes that assume an apriori knowledge of the data that is going to be accessed by transactions and accordingly pre-acquire the required locks deterministically. However, these schemes rely on coarse grain conflict classes to schedule transactions, which can severely restrict concurrency. This thesis addresses this limitation, via the use of Symbolic Execution to determine a fine-grained apriori knowledge of the transactions’ conflict classes to improve the efficiency of the scheduling process.

Keywords: Database Replication, Full Replication, Symbolic Execution, Distributed Transactions, Transaction Scheduling, State Machine Replication
# Table of Contents

1 Introduction ................................................................. 3  
2 Goals................................................................................. 4  
3 Related Work ................................................................. 4  
3.1 Database Replication .................................................... 5  
3.1.1 Single Master .......................................................... 5  
3.1.2 Multi Master ........................................................... 6  
3.1.3 Two-Phase Commit .................................................... 6  
3.1.4 Atomic Broadcast ...................................................... 7  
3.1.5 Summary ............................................................... 11  
3.2 Transaction Scheduling .................................................. 12  
3.2.1 Replicated Databases ............................................... 12  
3.2.2 Transactional Memory ............................................... 14  
3.2.3 Summary ............................................................... 15  
3.3 Symbolic Execution ..................................................... 15  
3.3.1 Limitations and Challenges ....................................... 16  
3.3.2 Improvements and Solutions ...................................... 17  
3.3.3 Concrete and Concolic Execution ................................. 19  
3.3.4 Use Cases ............................................................. 20  
3.3.5 Summary ............................................................. 20  
4 Approach ................................................................. 21  
4.1 Architecture ........................................................... 21  
4.2 Components ............................................................ 22  
4.3 Work Done ............................................................. 23  
5 Evaluation ................................................................. 23  
6 Work Plan ................................................................. 24  
7 Conclusion ................................................................. 24
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. Further, these services tend to have strong requirements for what concerns scalability, high availability and fault tolerance. These trends have created a strong urge for designing highly efficient database replication techniques. Replication allows to tolerate crashes of individual replicas and increase the perceived availability of systems. By placing multiple copies of applications’ data across failure-independents machines. However, 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 consistency in replicated systems is based on the State Machine Replication technique [1] (SMR). In a nutshell, SMR is based on an order-then-execute approach, also know in literature as Pessimistic 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.

Due to its generality, the SMR technique can be clearly applied to database systems. However, a key challenge is how to ensure that transactions executing at different replicas are serialized in the same order. In order to remove the costs associated with the execution of the distributed agreement phase, state of the art solutions, such as Calvin [2], 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.

On the other hand, conventional concurrency control schemes, long studied in the literature on transactional systems [3,4], 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. These include, schemes - such as NODO [5] that assume apriori knowledge of the data that is going to be accessed by transactions. An alternative is to estimate that data by simulating the execution of the transaction before replicating it, as proposed by Calvin [2]. Afterwards, the required locks are deterministically pre-acquired (using a single thread) before executing them.

Solutions, known as Certification [6,7] based, 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 with which transactions should be validated at each replica: 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, Certification solutions’ performance is highly affected by a large number of abort transactions occurrences. This is because, a transaction is executed regardless 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 been wasted in the transaction execution.

SMR-based solutions, on the other hand, are not affected by the limitations of Certifications, because execution is done after the required locks are acquired. These locks are acquired based on the knowledge of the data to be accessed by transactions. However, approaches [5] that assume this apriori knowledge rely on coarse grain conflict classes to
schedule transactions, which can severely and unnecessarily restrict concurrency. Other solutions [2], that estimate transactions’ accesses via simulation, are subject to large overheads. This is especially true, in high conflict workloads, as they require executing each transaction at least twice, and possibly more in case the simulated execution is inaccurate.

This thesis addresses the aforementioned limitations of SMR-based solutions, like NODO [5] and Calvin [2], by using a Symbolic Execution engine, such as JPF\textsuperscript{1} [8,9]. Symbolic Execution is a technique, originally developed for software testing [10] (but used also in other domains, like security [11] or privacy [12,13]), which allows for determining every possible execution branch of a code block, based on the value of its input parameters. With this, it is possible to determine a fine-grained estimation of transactions’ conflict classes that allows to schedule transactions in a more accurate and efficient way.

The remainder of the document is structured as follows. In Section 2 will be defined the goal of this work. Section 3 discusses the related work, where will be addressed all the properties mentioned before, Database Replication, Transaction Scheduling and Symbolic Execution. The proposed Approach is presented in Section 4 where is shown the architecture and all the components that compose the solution. In Section 5 is defined the evaluation process and the important metrics to measure. In Section 6 is presented the work schedule. Finally Section 7 presents the conclusions.

2 Goals

This thesis aims to investigate an idea, to the best of our knowledge, still unexplored in the literature: to use symbolic execution techniques to predict the object accessed by a transaction and build a fine-grained, deterministic, transaction scheduler for SMR-based database replication. With this, we aim to increase the parallelism of replicated databases.

In this thesis, we plan to apply symbolic execution to transactional programs in order to identify, in an accurate and autonomous way, the access patterns that will be generated by transactions during their execution - but without the need for executing them (unlike Calvin [2]) or any a priori knowledge on the transactions conflict classes (unlike NODO [5]).

The information extracted, offline, via symbolic execution will be used to build an online deterministic transaction scheduler, which can be exploited in SMR-based replication schemes to ensure that replicas execute transactions according to the same serialization order.

With this thesis, we intend to overcome the results of similar state of the art solutions, such as Calvin [2] and NODO [5], in real-world workloads.

3 Related Work

This section will go through the state of the art and will present solutions and techniques related to the research direction proposed in this thesis. The remainder of this section is organized as follows. Section 3.1, reviews database replication techniques. Section 3.2 focuses on transaction scheduling. Finally, Section 3.3, provides some background information on Symbolic Execution.

\textsuperscript{1} We are using an extension of JPF, called Symbolic JPF, that integrates a Symbolic Execution engine to the core model of JPF
3.1 Database Replication

Data replication has been an increasing concern over the years, in order to increase availability in 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 through more than one site. State-of-the-art Replicated Databases can be coarsely divided by whether replicas maintain a full state of data, named Full Replication, or whether they maintain only a subset of data, which is designated as Partial Replication. In this thesis, we are only interested in Full Replication.

Figure 1 shows our taxonomy of database replication techniques, with the root on full replication. This taxonomy is based on another presented by Couceiro et al. [14].

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 replicas. In Section 3.1.1 and Section 3.1.2, we will briefly address Single Master and Multi Master, respectively.

3.1.1 Single Master

A Single Master (SM) approach is constituted by an N number of nodes, where one of those nodes is appointed as primary 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. On the other hand, read-only transactions can be processed by any node, primary or backup node. This results in a superior throughput in intensive read workloads compared to update workloads. This means that the master represents a potential bottleneck for the processing of update transactions, as well as a possible single point of failure. In order to deal with crashes of the primary node, a fail-over process is executed, as designated by Schneider et al. [1]. Upon this process, backup nodes vote to select the new primary. In case of a failure on a backup node, the system maintains the normal behavior and ignores the faulty node.

Many solutions have been proposed that implement a SM approach, such as MySQL Replication [15] and PostgreSQL (Single Master) [16]. These are mainly used to provide high availability, improve fault tolerance and scale out read workloads.
PILEUS Pileus [17] is a recent solution that relies on a 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 to developers a service level agreement (SLA) that specifies the application’s consistency/latency desires. 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 available nodes to execute strongly consistent read operations, whereas the other nodes act solely as backups.

Terry et al. [17] 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, which can be beneficial in very high contention.

3.1.2 Multi Master

In a Multi Master (MM) approach, unlike SM, clients’ updates can be issued to every node. The failure handling mechanism of MM is different from the one of SM. In this approach, there is no need to replace faulty nodes, in order to resume the processing of update transactions. However, compared to SM, MM solutions require more expensive synchronization protocols between nodes, in order to guarantee the correctness of 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 can affect the throughput and the overall performance of the system.

Both approaches, SM and MM, have complementary advantages and disadvantages. The key advantage of SM approaches lies in their simplicity: by executing transactions at a single site, in fact, these approaches avoid apriori 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 conflict workloads. This is due to the fact that, no good MM solution exists that identifies apriori, in a fine-grained manner, if transactions, concurrently executing, will conflict. Because of this, this work will focus on MM approaches.

Multi Master approach can be coarsely classified depending on whether they use Two-Phase Commit or Atomic Broadcast. These two subclasses will be reviewed, respectively, in Section 3.1.3 and Section 3.1.4.

3.1.3 Two-Phase Commit

The Two-Phase Commit (2PC) is a well-known protocol to guarantee the atomicity of distributed transactions [3], whose execution is illustrated in Figure 2.

The protocol can be split into two phases, Prepare and Commit phases. 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) for the transaction: in case even a single participant voted negatively, the transaction is aborted; otherwise, it is committed. And lastly, the participants confirm to the coordinator the conclusion of the transaction.

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. The main down-side of 2PC is that, in conflict intensive workloads, the likelihood of incurring in transaction aborts grows cubically with the number of replicas [18]. Despite these limitations, recent systems use variants of 2PC. One example is Sinfonia [19], a system that exploits a priori knowledge on the data items to be accessed by transactions (called mini-transactions) to enables efficient and consistent access to data in distributed systems.

3.1.4 Atomic Broadcast

Atomic Broadcast (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 complete execution correctly, or all abort execution without affecting the previous state of the system. AB must satisfy the following properties, originally defined by Chandra and Toueg [20]:

- **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 and Certification.

**State Machine Replication**
State Machine Replication (SMR), also called Pessimistic approach, is a well-known technique for implementing a fault-tolerant service, proposed by Schneider et. al [21]. In the SMR protocol, replicas receive, from consensus, a batch of transactions to be executed in a deterministic order. This ensures that all replicas start with the same state and keep an equal state after each transaction execution. Numerous systems have been proposed in the literature, like BFT-SMaRT [22,23] and UpRight-BFT [24], that leverage SMR-based approaches to achieve Byzantine Fault Tolerance (BFT).

**Fig. 3: Two Parallel Transactions in SMR, presented by Couceiro et. al [14]**

Figure 3 shows an example of State Machine Replication with two replicas. In 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 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 and committed without the need for any remote synchronization. Also, Figure 3 shows the interactions between parallel transactions, i.e. a transaction that happens in the same time frame of another 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 [5] or NOn-Disjoint conflict classes and Optimistic multicast, uses a transaction reordering technique to avoid aborts. NODO does this even when the optimistic and the total message orderings are not the same - i.e. the real order that transactions were executed is different from the order established by AB. NODO executes transactions at only one site and allows transactions to access more than one conflict class. A Conflicts class is a set of data items that are accessed by a transaction. NODO assumes that the 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 queue corresponding 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 wait. When T1 is finished, T2 and T3 can be executed concurrently because both have different conflict classes. Each time a trans-
action finish executions, the order of execution is compared with the order established by
the delivery of the transactions. If both orders are equal, the transactions are committed.
If not, NODO analyzes the conflict classes of the transactions and decides if the optimistic
order still guarantees the correctness ensured by the order established upon delivery of the
transactions. If correctness is still guaranteed, the transactions are committed, if not they
are aborted.

Certification

Certification, also known as Optimistic solutions, execute a transaction in just one replica
and then propagate the changes to all replicas. The approach of Certification solutions al-
 lows, in low conflict workloads, to the execution time be lower compared to other approaches,
such as SMR. However, the decrease in execution time is nonexistent in high contention
workloads. In these scenarios, Certification approaches, when propagating to all replicas
the execution changes to be committed. It will incur with abort events due to conflicts with
transactions being executed in other replicas. This is the main limitations of this approaches.

Next, will be presented two systems categorized as Certification solutions, namely 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.

![Voting Protocol](image)

Fig. 4: Voting Protocol, based on the algorithm described by Rodrigues et al. [6]

Figure 4 shows an example of the Voting protocol. This figure is based on the general
outlines of the algorithm described in Rodrigues et al. [6].

As mentioned before, a transaction, in a Certification approach, is executed in just one
replica, this replica is designated as the delegate node. When a transaction is executed in
the delegate node, it is obtained the read locks on the read objects (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.
If there is already a transaction holding a write lock on any object intended, the transaction
is placed on hold until the locks are released. On the other hand, if a transaction holds
a read lock on any 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 any abort message, the transaction is aborted in all replicas and every lock obtained 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. [6] 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 will slightly reduce the number of aborts but the issue will still stand.

**Non-Voting**

The Non-Voting protocol is very similar to the Voting protocol. After a 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, like the name suggests, there is no voting phase. Each replica validates and takes the decision of 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 5 illustrates a possible case of the Non-Voting protocol described in Rodrigues et al. [6]. This figure is similar to a figure presented in Couceiro et al. [14].

![Fig. 5: Non-Voting Protocol, based on the algorithm described by Rodrigues et al. [6]](image)

As shown in Figure 5, transactions $T_1$ and $T_2$ are firstly executed in their respective delegate node. When the execution finishes and the transaction is ready to be committed, is sent the transaction profiles to all replicas. In this scenario, transaction $T_1$ is first delivered to the replicas via AB. Each replica, having in consideration the transaction profile, validates and commits (or aborts) transaction $T_1$. 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 $T_1$ and $T_2$ have the same version number, $T_1$ will commit and $T_2$
will abort, because when T2 is validated, the version number will be lower than the one in the replicas. 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. [6] suggest 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 AB messages and computational power are spared.

### 3.1.5 Summary

The solutions presented are all of them for full replication, because that is the focus of this thesis. Now will be done a summarization and comparison of all solutions presented. The comparison is represented in Table 6 below, where each approach is compared between:

- **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 this solution excels comparing to the rest of the state of the art.
- **A Priori Knowledge of Conflict Classes** - if the approach needs a priori knowledge of the transactions conflict classes to execute properly.

<table>
<thead>
<tr>
<th></th>
<th>Example Implementations</th>
<th>Weak Points</th>
<th>Strong Points</th>
<th>A Priori Knowledge of Conflict Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Master</td>
<td>[9], [16], [17]</td>
<td>Limited scalability in update intensive workloads</td>
<td>Simple and effective in low % update transactions or very high conflict</td>
<td>No</td>
</tr>
<tr>
<td>Two-Phase Commit</td>
<td>[10]</td>
<td>Subject to deadlocks in high contention</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Multi-Master</td>
<td>State Machine Replication [22], [24]</td>
<td>Poor parallelism with coarse-grained conflict classes</td>
<td>Scalable even with high conflict workloads</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Voting [6]</td>
<td>Subject to high abort rate in high contention, higher commit latency vs non-voting</td>
<td>Scalable with low conflict workloads</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Non-Voting [6]</td>
<td>Subject to high abort rate in high contention</td>
<td>Scalable with low conflict workloads, lower commit latency vs non-voting</td>
<td>No</td>
</tr>
</tbody>
</table>

Fig. 6: Comparison between the approaches presented

**Single Master** Single-Master is the most straightforward solution presented, being this one of its strong points. But, this same reason is the cause of the system having limited scalability. This is especially noticeable in update-intensive workloads, due to the reason of having just one master and the need of having to update the back-ups. Single-Master 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 high contention workloads, due to the fact of just having one master. This eases the conflict handling and also the scheduling of transactions. Single-Master 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 it is weak responding in update-intensive workloads.

**Two-Phase Commit** Two-Phase Commit main weakness is scalability, due to the fact the number of messages exchanged becoming excessive as the number of nodes increase. It
is also subject to deadlocks in high contention workloads. This can happen when mutual blocking between transactions occur, and execution of these 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 with an already high rate of message exchange.

**State Machine Replication** State Machine Replication has some scalability limitations, like most solutions presented, due to any replica added to the system executing 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. It scales well in high conflict workloads, with fine-grained conflict class of transactions. As seen before, SMR require a priori knowledge of conflict classes to work well. However, most solutions available require the developer to specify the transactions’ conflict classes [5] or require simulating an execution to determine the objects accessed [2]. This could result in faulty predictions, affecting the performance and even correctness of the system. Due to this reason, this approach would gain a considerable boost in performance with this thesis.

**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 be aborted. 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 result in Voting solutions having a higher commit latency compared to Non-Voting. This is because of the number of messages exchanged being higher in Voting solutions. Certification approaches do not require a priori knowledge of transaction’s conflict classes.

### 3.2 Transaction Scheduling

The goal of a Transaction Scheduler (TS), is to schedule transactions in a way to minimize 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 scheduling transactions, so conflicting transactions are not concurrently executed, the abort rate will decrease. Resulting in an increase of the processing rate of transactions.

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 3.2.1 and Section 3.2.2, respectively.

#### 3.2.1 Replicated Databases

Very little works have been done on TS on Replicated Databases. As seen previously, one weak point of all of database replication approaches mentioned is the fact that, in some scenarios, could result in a large number of aborts. Negatively affecting the execution and throughput of a system. Using TS techniques can drastically mitigate this issue. Next, we describe two approaches of TS in replicated databases, AKARA and AJITTS.

**AKARA** AKARA [25] is a database replication protocol based on group communication. The goal is to maximize resource usage by scheduling sufficient concurrent executions while
at the same time keeping queuing outside the danger zones. This ensures fairness and allows seamless active execution. 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 analyzes the queued transactions starting at the head and compares each transaction’s conflict classes to the ones before. If a conflicting transaction is found, it waits for its turn. This allows to send transactions to execute concurrently without occurring conflicts. Transactions move from Q2a to Q2b where are executed. 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 [26] 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 the 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 execution. The line is changed every time a transaction leaves the queue, i.e. it is committed or aborted. The mark is determined using the estimated execution time of the transactions and their input size, transactions with a higher value of input are executed earlier than with a lower value. The number of transactions executed and not yet executed are 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.

AJITTS reduces the number of aborts and also improves peak throughput even if it throttles transaction execution. This is achieved without requiring any workload specific configuration.
3.2.2 Transactional Memory

TS techniques have been widely used in Transaction Memory [27,28,29], with many proposed implementations over the years. Next, will be presented three solutions implemented in Transactional Memory, CAR-STM [27], Shrink [28] and Seer [29]. Due to the fact that the scope of this thesis is Replicated Databases, Transactional Memory goes outside of this scope. Because of this, we will not go through the specifications of the implementation of each solution, we will only consider the main idea of each solution.

CAR-STM CAR-STM [27] 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 analyze 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, is first withdrawn 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.

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

SHRINK Shrink [28] 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 the locality across repeated transactions. 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 queue of execution. 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 predicted read and write sets are free. Afterwards, the transaction waiting can be executed, without incurring any conflicts.

Conflicts can still happen, Shrink obtains roughly 70% accurate read and write accesses predictions, the rest is tolerable.
SEER 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’s acquired locks. With this table, Seer identifies which transactions could be executed concurrently. Every time a transaction exits execution, either because it committed or aborted, the list of active transactions is analyzed 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.

3.2.3 Summary

Transaction Scheduling can have a positive impact in a transactional system, by allowing better concurrency and reducing the chance of aborts. However, as seen in the previous section of Database Replication, the implementation of a Transaction Scheduling can have a negative impact in the approach, where the scalability and throughput of the solution can be negatively affected.

All the schedulers analyzed in this document require the knowledge of the transactions conflict classes. This description needs to be done a priori by the developer or at run time 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 is due to, the prediction process using the knowledge of transactions previously executed and by analyzing the transaction’s input. Our work solves this problem, by determining a fine-grained a priori knowledge of transaction’s conflict classes via Symbolic Execution. We expect this to improve transaction scheduling.

3.3 Symbolic Execution

Symbolic Execution (SE) is a program analysis technique first introduced by King in [10]. It is mostly 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, like for example, gaining privileges improperly.

SE abstracts the concrete value of variables, called symbolic variables, to construct a path condition, i.e. a boolean expression that unequivocally identifies the constraints associated with each path. SE achieves this by constructing a tree that represents the program
execution. Whenever the execution finds a conditional statement, the tree splits to explore all possible execution paths. These conditional statements are represented as \((smt, \sigma, \pi)\), as shown in Baldoni et al. [30], where:

- \(smt\) - is the statement to evaluate, e.g., the condition to satisfy.
- \(\sigma\) - represents the current state of the program variables. This can include expressions over concrete values or symbolic values. These values are represented as the concrete number itself or as \(\alpha_i\), respectively.
- \(\pi\) - denotes the path constraints. Path constraint is an expression of a set of symbolic variables states to reach \(smt\), 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 a different symbolic value is assigned to a variable. With this, it is possible to get a very fine-grained information about every execution path of the program.

The following sections identify some known limitations and challenges of SE, Section 3.3.1. Afterwards, Section 3.3.2 presents some solutions to counteract those limitations. Section 3.3.3 briefly describes Concrete and Concolic execution. Finally, Section 3.3.4, goes through some use cases of SE.

### 3.3.1 Limitations and Challenges

SE was designed following a number of performance-related design principles, the most notable were depicted by Baldoni et al. [30]:

- **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 analyze time of a program. **Work repetition** and **analysis reuse** principles are for improving performance, but in some cases could raise some challenges, related to caching and loop handling.

The survey of Baldoni et al. [30] listed some challenges inherent from the SE state of the art:

- **Memory** - Memory limitations are already a challenge on its own, having to manage the amount of memory needed for the execution. But is also a big challenge, having to handle and simulate memory behavior, such as pointers, arrays, or other complex objects present in programs.
- **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 symbolic executor has to explore, incurring performance penalties, as well increasing the amount of memory needed for the execution. This results in paths not being explored.
- **Constraint Solver** - Constraint solvers suffer from many limitations. The more prevalent is handling time, that increases exponentially as the number of constraints increases, or if complex constrains are added to be solved.

The following section will address some solutions for these challenges.
3.3.2 Improvements and Solutions

This section describes some solutions to the challenges described before.

**Memory** One of the main challenges 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. For other cases, can be used *address concretization*, which 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, e.g. execution paths. To mitigate the scalability problems of *fully symbolic memory* and the loss of soundness of *address concretization*, is introduced *partial memory modeling*. The main idea of this approach is to use the best parts of both previously approaches, where write addresses are always concretized and read addresses are modeled symbolically. However, if in a read address the range size of references exceeds a given threshold, the address is concretized.

The handling of complex objects is another challenge, because these objects must be allocated in such a way that take advantage of the object properties. A possible approach is to only initialize them when they are accessed. An object can be initialized with: (1) null, (2) a reference to a new object with all symbolic attributes, and (3) a previously introduced concrete object of the desired type. With this approach, is not required an a priori bound on the number of input objects, they are bound when accessed. This can be extended using *method preconditions*, where input objects states are characterized externally by their intended behavior.

**Loops** The common solution to counteract the loops challenges is to compute part of the loop, by limiting the number of iterations. This increases speed in exchange for soundness, as lot information can be lost. Other approaches infer loop invariants through static analysis and use them to merge equivalent states.

**Path Explosion** Due to the use of loops, the number of execution states could exponentially increase, that would cause a Path Explosion. Path Explosion can also occur in complex programs, without the use of loops. To mitigate this problem, for both cases, the only solution is to restrict the symbolic execution engine to explore just a fraction of the possible paths. There are several approaches to this problem where the goal is to identify and explore all the important execution paths. Two implementations that have this concern 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 complex programs, due to the fact that, paths containing loops and recursive calls can easily stall the execution, this is especially noticeable in DFS. BFS is the best solutions for programs with loops, ends up with a larger path coverage and overcomes the staleness problem.

An alternative technique is to use a method called *preconditioned symbolic execution* that drive a symbolic execution towards a subset of inputs space. The subset is determined by defining some preconditions. With this method, the exploration space gets narrower, allowing for a more efficient exploration of the paths. These preconditions need to be carefully selected, if one 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 are variations to achieve the same goal of not having to explore all execution paths. What varies between implementations is the grade of exploration. Some try to explore more paths but with poorly precise information, others analyze thoroughly each path, but explore fewer paths.

**Constraint Solver** Constraint solvers are still one of the main obstacles to the scalability of symbolic execution engines. There are many approaches that try to minimize this problem such as Z3 [31] and Choco [32]. The more prominent solutions are the *constraint reduction* and the *reuse of constraint solutions*. The first one consists in reducing the size and complexity of the constraints to evaluate. Some optimizations were done for cases where a complex constraint could be divided into simpler constraints, being possible to run these simpler constraints concurrently. In some cases could even be possible to, when a new constraint is introduced, to rewrite it as an already existing constraint. 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 in order to avoid calling the solver.

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 approach, the solver will be called less times, optimizing the execution time.

**KLEE** KLEE [33] is a symbolic execution tool proposed by Cadar, that is capable of automatically generating tests that achieve high coverage on a diverse set of complex programs. 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, which requires careful attention to memory consumption because of the many active states. KLEE use caching techniques that allow to never re-executes previous instructions. To mitigate the memory consumption problem, KLEE uses the *fully symbolic memory* approach. The improvements that KLEE brought were mostly related to the path explosion problem. One of them is assigning probabilities to paths based on their 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 uses some approaches mentioned previously. It attempts to reduce the constraints by simplifying the expressions computed. It does this by using previously obtained results in future constraints. For example, if an equality constraint form $x := 5$ enables a simplification of other constraints over the same variable $x$, using something like $x < 0$ instead of the exact number.

Although KLEE is not the ideal symbolic execution tool for every program, it improved and optimized many of symbolic execution challenges by introducing new techniques and approaches, that are used by many other tools.

**JPF** Java PathFinder (JPF) [9] is an open-source runtime environment for verifying Java bytecode developed by NASA. JPF is composed of a core, called *jpf core*, that can be easily extended depending on the set objective. The JPF core is a program which receives Java programs to find defects in the programs. One of JPF extensions, and the one that we will use, is Symbolic JPF [8,34], a framework that integrates SE with model checking of *jpf core*. Symbolic JPF generates test cases that obtain a high code coverage. Programs are executed on 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 test inputs guaranteed to reach that part of the code.
Symbolic JPF is always receiving updates, because of being an open-source project. This allows for the improvements developed over the years to be also implemented and available in Symbolic JPF.

3.3.3 Concrete and Concolic Execution

To address some of the limitations of SE, 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 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 \( \alpha + 2 \), or a path constraint to satisfy a given condition, like \( \alpha > 0 \). Concolic Execution can use both concrete and symbolic variables.

Next, we will briefly discuss these two execution techniques, Concrete and Concolic Execution [30], and then we will present CUTE [35], 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 is found a value that generates a new execution flow. These values are then stored in the path constraint (\( \pi \)) 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. Due to space (memory) and time limitations, execution paths can end up not being explored. Because of this, Concrete Execution allows to set properties, in order to prioritize the exploration of a specific path flow. Concrete 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 [36] and was extended afterwards by Sen and Marinov with CUTE [35]. 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 the weaknesses of SE. Is not affected by the weaknesses of Concrete Execution because they are complemented by the SE.

**CUTE** CUTE [35] is a unit testing engine that use Concolic Execution techniques. CUTE is available in C and in Java with jCUTE [37]. 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 introduced a technique to avoid redundant input testing, i.e. input that would give a previously seen result. With this technique, is more probable to test critical input that could cause incorrect behavior 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 programs with concurrency and programs using algebraic functions, such as cryptographic protocols, due to the huge number of possible paths.

### 3.3.4 Use Cases

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

**SAFELI** SAFELI [38] 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. When symbolic executing the application, a critical spot which submits SQL query is encountered, a hybrid string equation is constructed to explore the initial values of Web controls which might be used to apply 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 attack patterns in these tests. This library is the weak point of SAFELI, because of the reason for needing to be manually updated. The authors want, in the future, to not have an attack pattern library at all, making SAFELI completely autonomous. This could be possible by tackling the string constraints issues, an issue that the authors leave open to solve in the paper.

**Other implementations** Marcozzi et. al [39] 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 allow the algorithm to generate a set of relational constraints for any finite path to test in the control-flow graph of the method. The solutions of these constraints are data which constitute a test case, which exercise the selected path in the method. To achieve this goal, the algorithm receives as input a SQL DLL file, that describes the database scheme, receives the JAVA method to test and the execution path of the method. The algorithm outputs the relational constraints generated that are expressed using a widely used and well-documented language called Alloy. The outputs are then sent to an Alloy analyzer that solves Alloy constraints in order to find structures that satisfy them. Basically, transforms the given relational constraints into a set of boolean constraints. The advantages of this approach compared to other similar solutions, is that with this approach does not need to transform the original program code, supports more SQL statements, transactions management primitives (Commit and Rollback) and database integrity constraints (e.g. primary keys or foreign keys). The downside is that this approach only supports SQL statements.

### 3.3.5 Summary

SE is a very powerful testing tool, despite its limitations. Although SE is used for testing purposes, there are other interesting use cases, such as SAFELI [38] and Marcozi et. al [39]. These use cases tried to extend the useful range of SE and with this work, we also intend too. SE is used as an asset to enable having a priori knowledge of the transactions reads and writes sets and use these results for optimizing the way that concurrent transactions are handled in fully replicated database systems.
4 Approach

Current approaches of database replication have several problems as discussed in Section 3.1. In particular, the SMR approach offers an interesting trade-off but it is limited by the granularity of the conflict classes. The main idea of our approach is to use Symbolic Execute to achieve a fine-grained conflict classes of all transactions in order to improve the scheduling process.

In the rest of this section, we will present the Architecture for the proposed solution, in Section 4.1. Afterwards, in Section 4.2 will describe every component of the Architecture. Finally, in Section 4.3, will be addressed the work already done.

4.1 Architecture

Figure 7 presents the Architecture of our proposed solution. The architecture is composed by two different modules: one that works at compile-time, which include the SE Engine, and another one executed at run-time, which encompasses the Scheduler, a waiting queue and the Solver. We first describe the overall approach and then in Section 4.2, we describe each component in more detail.

At Compile-Time Compile-time execution is provided by a SE Engine, which outputs the conflicts classes of all transactions in an application. This is accomplished by symbolic executing the application code. The SE Engine explores the paths of each transaction and obtains all possible read and write sets. The output lists every transaction in the application. Each transaction is sub-divided by path constraints identified inside that transaction. And finally, each path constraint has the corresponding read and write sets. This output is

Fig. 7: Proposed Architecture
then used at runtime by the Solver (0) to identify, based on a transaction inputs, its conflict classes.

**At Run-Time** At run-time, the system receives a batch of transactions (1) previously agreed by a consensus algorithm, such as Paxos [40]. The transactions, following the order set by consensus, are handled by the Scheduler where each transaction’s inputs are extracted and sent to the Solver (2). The Solver then checks if the arriving transaction conflicts with transactions in the system, i.e. transactions in queue or already executing (3). The Solver, using the output given by the SE Engine (0), checks what are the conflict classes of the arriving transaction. Then, the Solver compares those conflict classes with the ones of the transactions already executing and queued. It is necessary to also check the queued transactions, because a new transaction cannot execute first than an already queued transaction with the same conflict classes. This is because, a queued transaction comes first in the order set by consensus, so it has priority over an arriving transaction. Then the Solver delivers the results to the Scheduler indicating whether the transaction conflicts or not (4). Based on this, the Scheduler decides where to send the transaction (5): if there is no conflict, the transaction can be executed (6) and added to the list of Transactions Executing, if it results in a conflict, the transaction is queued. Upon every commit (7), the Solver updates its model and checks if any queued transactions can be executed. Transactions that have no conflicts, with the transactions executing and with the ones ahead in the queue, can be sent to execution.

4.2 Components

We will now describe each component in detail.

**Symbolic Execution Engine** The SE Engine that will be used is the JavaPathFinder (JPF) [8,9]. JPF offers many mechanisms to the developer. One that will be used is listeners. More specifically, a PropertyListener, that allows to execute code before and after an instruction execution. Using this listener in methods that perform read and writes operations (e.g. put and get methods), we are able to extract which objects are accessed. This information in conjunction with the path conditions given by the SE, gives a fine-grained information of the relation between inputs and what objects are accessed. The output of this process is then used by the Solver.

**Scheduler** The Scheduler pops from a batch of transactions, given by AB, one transaction at a time. Each transaction comes with its corresponding inputs. These are then sent to the Solver to check if the transaction has conflicts or not. The Solver then sends back to the Scheduler the results if the transaction conflicts or not. The Scheduler, based on these results, decides if a transaction is placed in execution or if it is queued. The Scheduler is executed in a single thread, because a multi-thread Scheduler could result in deviation of the state of each replica. Thus, violating the order set by consensus.

**Solver** The Solver will be implemented recurring to the Z3 Solver [31], a high-performance constraint solver developed by Microsoft. The Solver is composed of two threads: one that receives transactions inputs from the Scheduler and other that receives information of committed transactions from execution.

The first one, after receiving the inputs from the Scheduler, uses the output of the SE Engine to identify what objects will be accessed by this transaction based on its inputs. Then it must check if this transaction conflicts with transactions already in the system, i.e. in Queue or in Transactions Executing. It does this by searching for a match of the
transaction’s conflict classes, with the ones queued and currently executing. If it is found a match in a transaction already executing, the transaction is placed in the Queue. However, if it is not found a match, the transaction cannot yet go to execution. The Queue must be checked before that. This is to ensure that a new transaction does not overtake an older transaction already in queue with the same conflict classes. Only if there is no match in both, Queue and Transactions Executing, the transaction can be sent to execution.

The second thread receives information about transactions that finished its execution. Each time this occurs, the Solver checks if there is a waiting transaction that can be executed. It does this by repeating the same process described previously, of checking the Queue to ensure that correctness is guaranteed. Afterwards, the transaction is removed from the Transactions Executing.

**Queue and Transactions Executing** All transactions already processed by Scheduler and Solver are placed in the Queue or in Transactions Executing. Transactions being executed are placed in Transaction Executing and the ones waiting are placed in the Queue. In both cases, the transaction has its input and its conflict classes, that were determined by the Solver. The Queue follows a FIFO (first in, first out) policy to guarantee that transactions are analyzed following the order given by consensus. So, new transactions placed in the Queue go to the end and searches done by the Solver start in the beginning.

### 4.3 Work Done

The choice of which Symbolic Execution tool to use was an important and pressing issue. The choice of JPF had in consideration all the goals that have been established. To familiarize with the tool, we started by identifying the path constraints in a simple example, with arrays serving as a data-store. Then we did the same process but with a more complex example, that had three distinct transactions, but still using arrays. In this step, we implemented a listener of the type `PropertyListener`, to obtain the inputs of the objects accessed. To achieve this, we need to identify the methods that perform read and write operations. For now, these methods are identified a priori by stating in a file which methods are. Because we are only interested in symbolically executing the portions that access the database. However, when implementing this, we faced a problem related to the arrays. Because, when JPF symbolic executed a program, every position of the array would be tested until an exception was thrown. To mitigate this problem, we used the implemented listener and added a functionality of skipping the instructions of the body of the, previously mentioned, methods. In other words, we only execute the invocation of the method (to extract the inputs) and the return statement. Any instruction between this is not executed. At this stage, our solution already identified the path constraints and corresponding read and write sets. Then, we changed the example to use a more sophisticated data-store, such as hash-maps. But, thanks to the added functionality of skipping the instruction of the method’s body, the underlying data-store ends up being irrelevant. Currently, JPF in conjunction with the implemented listener can exactly determine the read and writes sets of all transactions in a variation of the TPC-C benchmark. It outputs every transaction in the application, their path-constraint and the read and write sets of each path constraints.

### 5 Evaluation

The proposed solution is a complex system that includes various components. Because of this, the solution will be evaluated in two ways. We will evaluate the performance of
Scheduler and the Solver alone, by using micro-benchmarks. To evaluate the performance of all the system will be used macro-benchmarks.

In both evaluation processes will be measured a number of key metrics. These metrics are throughput, abort rate and latency. The throughput and abort rate show the efficiency of the proposed solution on decreasing abort events. The latency measures the performance of the solution by having in consideration the execution time of the of the system. This is especially important to compare the execution time of the Scheduler + Solver to the execution time of the system as a whole. This allows to identify where a possible bottleneck of the system may be. The evaluation of the system scalability by increasing the number of replicas, it is also important to measure the impact that this has on the system.

The proposed solution will also be compared to existing solutions that have similar goals to ours. Possible candidates are Calvin [2] and NODO [5]. Both work similar, the difference is that Calvin performs an execution simulation to obtain the transactions’ conflict classes. NODO, on the other hand, requires a description of the transactions by the developers. Comparing our solution to these two solutions, in equal scenarios, will allow to show the benefits of our approach.

To measure the solution implementation will be used well-known benchmarks, such as TPC-C [41]. TPC-C is widely used to evaluate transactional systems. It measures the transaction rate and the associated price per transaction. With this, we can confirm if the impact of the Scheduler and Solver is low and also that the abort rate is low.

6 Work Plan

The work plan is the following:

- January 5th - March 1st: Implementation of the Solver and integration with the Symbolic Execution engine.
- March 1st - May 1st: Implementation of the full system. Implementation of the Scheduler and integration with the Solver. Implementation of the queue and transaction execution process and integration with both the Solver and Scheduler.
- May 1st - June 15th: Porting of benchmarks, test of the implemented system via micro-benchmarks and adapt existing solutions that will be used in the evaluation process.
- June 15th - August 1st: Evaluation of the Solver and scheduling process via micro-benchmarks and of the all system using macro-benchmarks. Comparison of the proposed solution with existing solutions.
- August 1st - September 1st: Preparation of a paper for submission to a scientific conference.
- September 1st - October 1st: Preparation of Masters’ thesis

7 Conclusion

In this document, we reviewed the state of the art on Database Replication, focusing on the full replication approach, where we identified the limitations of the existing approaches. For SMR in particular, the main limitations are identifying transactions’ conflict classes. The existing approaches require either the programmer to define the program conflict classes, a notoriously hard task, or to define conflict classes automatically, but in a coarse-grained way. The review of the state of the art highlights that no good solution exists. Symbolic Execution
is a promising approach to tackle this limitation. The ability to explore all branches of execution allows to determine in which conditions objects are accessed. We intend to develop a Scheduler that leverages the information obtained by the Symbolic Execution to disallow potentially conflicting transactions to execute, while optimizing throughput.

During the next phase of the dissertation, we aim to continue addressing and expand the limitations identified. In this document, we have already present a solution to address these issues. In the end, we will have an implemented solution to fine-grained schedule transactions in a Replicated Database via the usage of Symbolic Execution.

References


