Fine grained transaction scheduling in replicated databases via symbolic execution

Miguel Cândido Viegas
miguel.viegas@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2018

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, replication introduces non-negligible costs in order to ensure that the state maintained by the various replicas is properly synchronized. Database replication approaches, such as State Machine Replication (SMR), have limitations when it comes to parallelism. The state of the art solutions that try to solve these limitations rely either on automatic predictions or programmer input about the set of data items to be accessed. The predicted item accesses by these solutions are either too coarse-grained or are too optimistic, which 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 items accessed by transactions to improve the efficiency of the scheduling process.
Keywords: Database Replication, Full Replication, Symbolic Execution, Distributed Transactions, Transaction Scheduling, State Machine Replication

1. Introduction

Nowadays, most services available over the Internet make large use of databases to store relevant data. 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) [3]. 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 [2, 11], 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 [8] 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 [10]. 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.

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. Document Structure

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.

2. Background and Related Work

This section provides some background about the state of the art of Database Replication techniques and an overview 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. 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 this is based on the State Machine Replication (SMR).

2.1.1 State Machine Replication

SMR is a well-known technique for implementing a fault-tolerant service, proposed by Schneider et. al [9]. 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 [7, 6]. 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.

NODO [8] 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 which is an impossible task with complex transactions. Another problem is that NODO's conflict classes only consider the table of the items accessed, which is too coarse-grained of an information to efficiently control the concurrency of transactions.

Calvin [10] is a transaction scheduling and data replication layer that orders transactions' execution deterministically. To do this, 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.

2.2. Symbolic Execution

Symbolic Execution (SE) is a program analysis technique first introduced by King in [4]. 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. The root of the tree is the initial state of the program. During the analysis, if SE finds a path constraint, splits the execution 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. This tree can also include 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.

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 [8], determine the transactions' conflict classes with a large granularity, which limits the degree of parallelism. Other solutions, such as Calvin [10], 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 2.2, 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. In Section 3.2, we discuss some arguments related to the correctness and determinism of the presented solution.

3.1. Overview

Symb-SMR has two main modules, depicted in Figure 1. One module is the client application and the

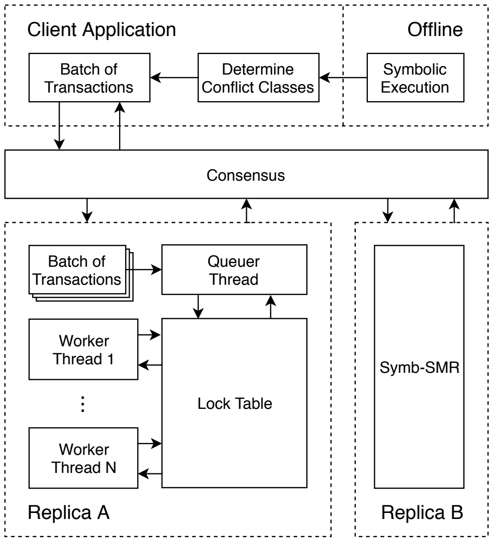


Figure 1: Symb-SMR Overview

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 runtime, 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. 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.

4. Results

This section presents the experimental evaluation of the Symb-SMR. The solution will be evaluated with two different workloads, one generated from the No Contention micro-benchmark, Section 4.1, and the other with the TPC-C benchmark [1], Section 4.2. With the No Contention micro-benchmark, we evaluate the scalability of the system and identify possible bottlenecks, in particular in the Queuer and Workers threads. With TPC-C we compare Symb-SMR with other state of the art solutions, Nodo [8] and Calvin [10]. In particular, we will compare Calvin's handling mechanism for transactions that fail against the algorithm of Symb-SMR.

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. 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 [5]. 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.

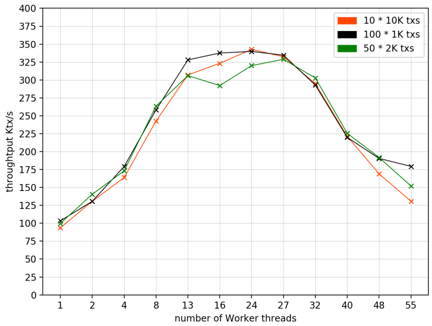
4.1. No Contention Micro-Benchmark

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.

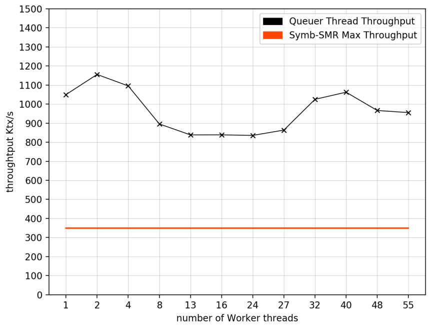
4.1.1 Experiment Results

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 2 (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 2 (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 3 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 a queue containing the transactions that are ready to execute. Figure 3 (a) shows the results for 10 batches of 10 000 transactions, Figure 3 (b) shows the results for 100 batches of 1 000 transactions and Figure 3 (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 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.

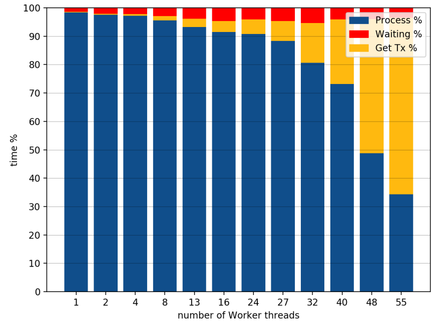


(a) Symb-SMR's throughput

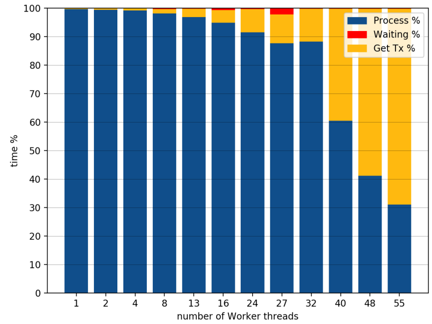


(b) Queuer throughput and Symb-SMR's max throughput

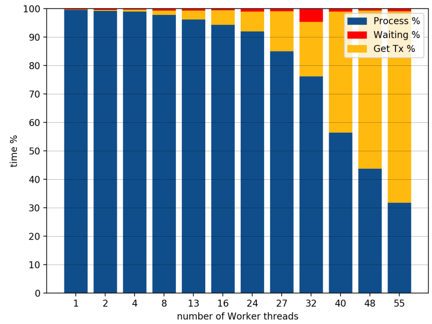
Figure 2: No contention workload of 100 000 transactions



(a) 10 batches of 10 000 transactions



(b) 100 batches of 1 000 transactions



(c) 50 batches of 2 000 transactions

Figure 3: Breakdown of Symb-SMR's Workers when processing batches with different sizes

4.2. TPC-C Benchmark

TPC-C [1] 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

4.2.1 Nodo

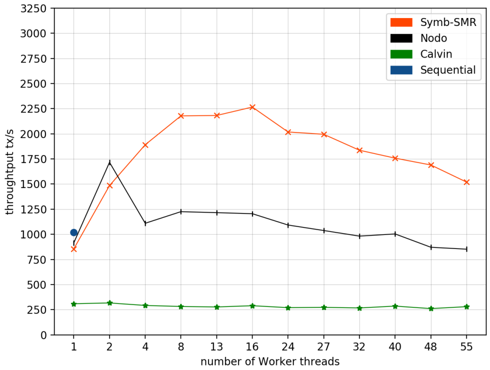
Nodo [8] is a concurrency control system, that was described in Section 2.1.1. Symb-SMR implementation was largely based on the Nodo idea of controlling the concurrency between transactions based on their conflict classes. However, the big difference between both solutions is that: (1) Nodo requires 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.

4.2.2 Calvin

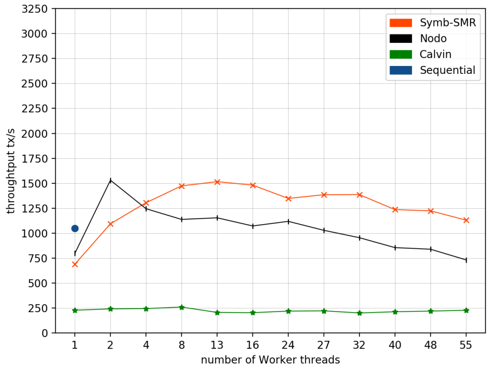
Calvin [10] is a transaction scheduler, that was presented in Section 2.1.1. The way Calvin and Symb-SMR work is very similar. However, the main differences are: (1) the way conflict classes of transactions are determined, (2) how failed transactions are handled and (3) the scheduling of Read-Only transactions (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.

4.2.3 Experiment Results

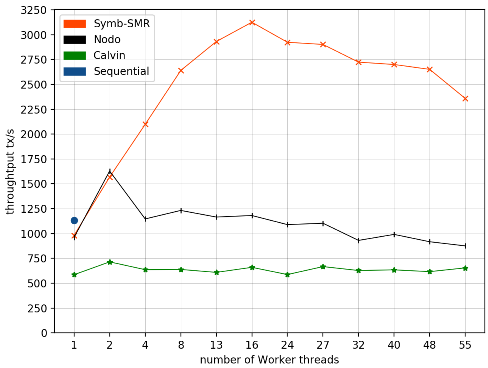
For these experiments, we generated TPC-C workloads of 100 batches with 100 transactions. Figure 4 depicts the results of the solutions throughput for processing these workloads. In these plots, we compare Symb-SMR with three other solutions, Sequential, Nodo and Calvin. The Sequential implementation works as a baseline to measure the impact of each solution. In Figure 4 (a) and Figure 4 (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 (a) approximately 50% of the transactions are Indirect transactions and in Figure 4 (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 (c) and Figure 4 (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 (c), increased compared to Figure 4 (a). This is due to TPC-C transactions conflicting less when the size



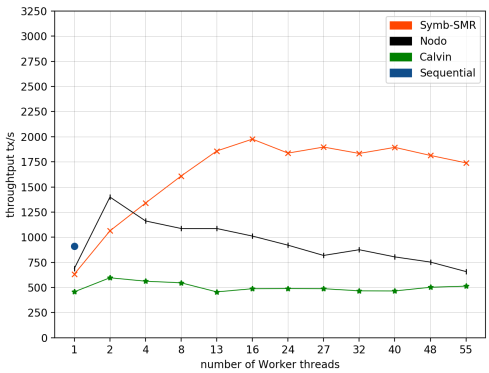
(a) 10 Warehouses with 50% Indirect transactions



(b) 10 Warehouses with 90% Indirect transactions



(c) 55 Warehouses with 50% Indirect transactions



(d) 55 Warehouses with 90% Indirect transactions

Figure 4: 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

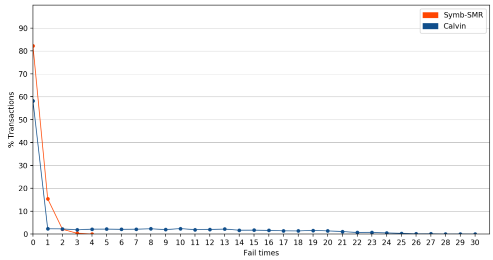
of the Warehouses table increases. Since Nodo only considers the tables to control the concurrency, an increase in the table size will not impact the performance as we can observe in all plots. Also, the number of Workers do not benefit Nodo, in fact, increasing the number of Worker threads slightly 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.

4.2.4 Failed Transactions

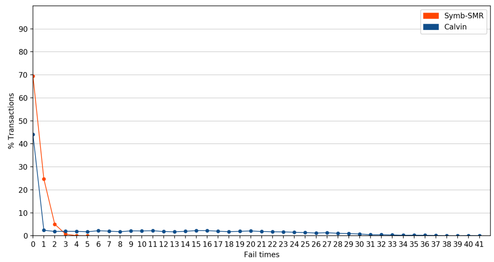
In these experiments we compare Symb-SMR with Calvin by measuring the number of times transactions fail until they are successfully executed. Nodo is not included in these experiments because is not impacted by IT resulting in transactions not failing due to changes in conflict classes. Figure 5 show 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 5 (a) and Figure 5 (b) are with 10 Warehouses, 50% and 90% of Indirect transactions respectively. Figure 5 (c) and Figure 5 (d) are with 55 Warehouses with the same changes in Indirect transactions respectively. In Figure 5 (a) and Figure 5 (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.

5. Conclusions

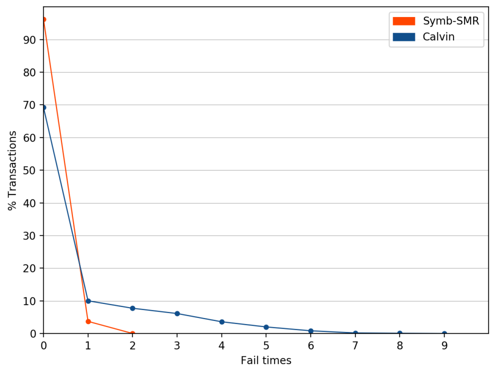
The main goal of this dissertation is to increase the level of parallelism of a SMR-based system and to



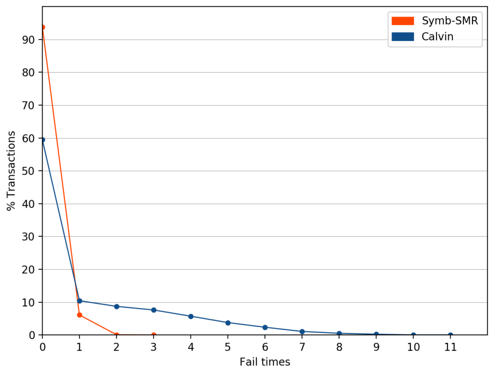
(a) 10 Warehouses with 50% of Indirect transactions



(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 5: 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

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.

Acknowledgements

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. I would also like to thank Pedro Raminhas and Nuno Machado, for the huge support they given me during the implementation. Secondly, I would like to thank all my family, in particular my parents for always supporting me no matter what and for helping me in every obstacle I faced. 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.

References

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

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Chapter 8: The primary backup approach, 1993.

[4] J. 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.

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

[6] P. J. Marandi and F. Pedone. Optimistic parallel state-machine replication. *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 57–66, 2014.

[7] 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.

[8] 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.

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

[10] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

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