Speculative Execution on Distributed and Replicated Software Transactional Memory Systems

(extended abstract of the MSc dissertation)

João Fernandes
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—This paper describes and evaluates SPECULA, a distributed and replicated software transactional memory system based on a certification scheme. This system tackles the negative effects of network latency, through the optimistic execution of code. Transactions are executed on a single node, in an uncoordinated fashion, and the result of their local validation is used as a prediction of the result of the final validation. The results of speculatively executed transactions (i.e., the modifications to the transactional state) are made visible to future transactions in an optimistic fashion. This speculative process repeats itself, creating a chain of speculatively executed transactions. If the final validation of a speculatively committed transaction allows it to commit system-wide, its result is made definitive; otherwise, a cascading abort takes place and the system restarts its execution in the state that preceded the misspeculation. This speculative behavior is fully transparent for both the application and the programmer. The proposed design was evaluated using STMBench7, a well known benchmark that exercise very different scenarios.**

I. Introduction

The mainstream adoption of multiprocessor systems has brought parallel programming to the center of the main stage. For a very long time, application developers relied on the *free lunch* offered by the increase in single processor performance, and were able to avoid the *pains* of parallel programming. That time has come to an end.

Software Transactional Memory (STM) is an abstraction that eases the life of programmers dealing with concurrent data access. When using a STM, the programmer is freed from having to explicitly deal with concurrency control mechanisms. Instead, he only has to identify the sequences of operations that need to access shared data in an atomic fashion (i.e., transactions). Given that the use of low-level concurrency control mechanisms such as locks and semaphores is known to be extremely error prone [1], the use of STM has the potential to increase the code's reliability, and to shorten the software development cycles.

Although STM was first proposed for cache-coherent shared memory architectures, the need to increase the scalability and fault-tolerance of STM-based systems has motivated the development of distributed and replicated STM implementations.

This work addresses the problem of building efficient Distributed Software Transactional Memory (DSTM) implementations that are fully replicated by presenting SPECULA, a system that explores the notion of possibly useful (speculative) computation to tackle the negative impact of network latency on the throughput Distributed and Replicated Software Transactional Memory (DRSTM) systems.

The remainder of the paper is organized as follows. In Section II we introduce state-of-art DSTMs. In Section III we present SPECULA, the system that aims to solve the problem we identify. Section V contains the evaluation of the prototype that we developed for the JVM platform. Finally, in Section VI we conclude this paper with some final remarks.

II. Related Work

There exist some replication schemes specifically tailored for DRSTMs, namely BFC [2], AGGRO [3] and SCert [4]. BFC (Bloom Filter Certification) is a certification-based replication protocol that has as its main characteristic the encoding of read-sets into Bloom filters [5], a probabilistic data structure, as way of reducing the size of the messages exchanged among nodes, and consequently, the duration of inter-node communication.

AGGRO (AGGRessively Optimistic) is an active replication protocol that explores early indications from the network layer to execute transactions in a speculative fashion. It leverages on the fact that in LAN environments, the natural message delivery order typically matches the delivery order defined by Atomic Broadcast (AB) protocols. Speculatively executed transactions propagate their write-sets to future transactions, creating a chain of speculative dependencies. In case of mismatch, i.e., the the optimistic delivery order and the AB delivery order are not equivalent, the system performs a cascading rollback, aborting all invalid transactions.

SCert (Speculative Certification) works similarly to AGGRO, but it is a certification-based replication scheme. By propagating the write-sets of speculatively committed transactions to future ones, SCert effectively reduces the abort rate of transactions, as more of them fed with non-stalled data.

III. SPECULA

A. The Problem

The main motivations behind the creation of DRSTMs are scalability and reliability. Regarding scalability, a commodity cluster is much cheaper than a supercomputer, but it is also typically much more difficult to scale systems horizontally than vertically, due to the cost of inter-node communication. Regarding reliability, high-availability requisites are very common in real world applications. However, high-availability must be achieved at the minimum possible cost, thus replication protocols should be simultaneously effective and efficient.

Most replicated systems communicate through some sort of computer network. Due to distance and medium propagation speed, computer networks feature much higher latency and lower bandwidth than the bus of a computer. This means that if a computer stops executing to communicate with another node, it is wasting the possibility to execute millions of instructions, as it is just sending or receiving data. Therefore, programmers struggle to minimize the effect of the communication latency, either by reducing the number of communication steps or the amount of data to exchange, or by overlapping communication with useful computation. In this context, the cost of the communication can be measured by the number of instructions that could be executed while the process is waiting for the communication exchange to terminate.

Certification-based replication schemes have shown to offer good performance on multi-master database environments. In those settings, there are several sources of delay. First, transaction processing is subject to pre-execution stages like parsing and query optimization. Second, database transactions are required to access stable storage synchronously. These effects dilute the costs induced by communication. It is worth noticing that in certification-based replication most communication costs are associated with the execution of an atomic broadcast primitive, that can take two or more communication steps and requires the exchange of multiple messages [6].

In a DRSTM, many of the costs above are not present. This amplifies the relative cost of communication. Therefore, naive ports of DBMSs protocols to the DRSTMs may offer poor performance [8]. Figure 1 depicts the transaction execution times observed in two benchmarks, STMBench7 and TPC-W, being the former for STMs and the later for DBMSs. As we can see, almost 80% of all memory transactions executed in the STMBench7 benchmark took less than 1 *ms* to finish, while a similar percentage of database transactions executed in the TPC-W benchmark needed almost 10 *ms*.

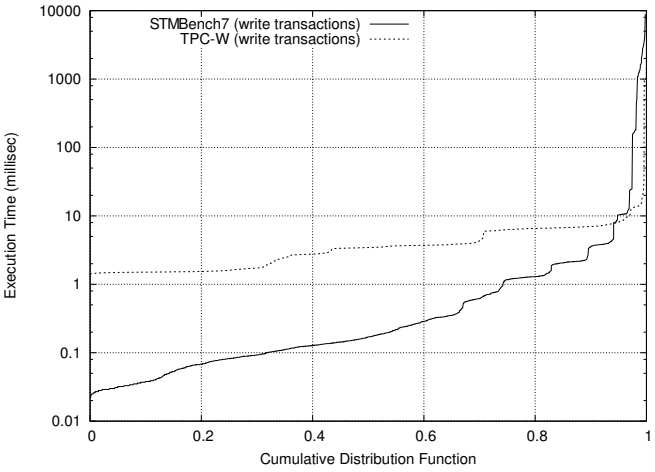
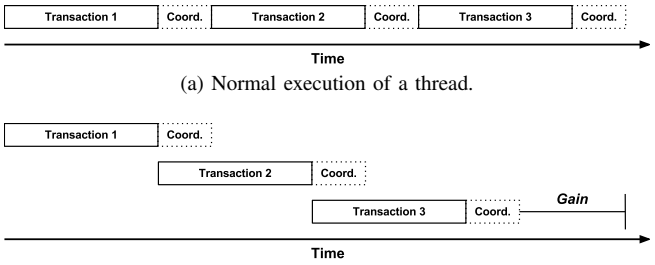


Figure 1: Execution time of database and memory transactions [7].

There are several approaches to mitigate the performance loss due to costs involved in inter-replica coordinations:

- One approach consists in reducing the amount information exchanged in the messages, to speed up the message exchange. This can be achieved using some encoding techniques, such as Bloom filters. This is the approach used by D^{2}STM, and experimental results show that it is an effective solution [2].
- Another approach would be to start a speculative commit when the transaction begins, in an attempt to run most of the coordination in parallel with the computation of the transaction. This approach has several limitations. First it requires the read and write set of the transaction to be accurately estimated, which is generally hard (unless the system is restricted to use static transaction). Another disadvantage is that coordination is still much slower than computation, so the process would be required to wait in any case. The experimental results in [8] show communication taking around 6 to 26 times more time than execution when considering an AB-delivery time of 2 *ms*. To put these results in a different perspective, conducting the same tests over the same network environment, but with 10 times faster machines, will result in a speedup varying between around 1.03 and 1.14, which truly reflects the amount of computational power that is wasted by standard solutions.
- A third alternative consists in using speculation in the coordination protocol to start the certification of the transaction earlier, and proceed with the (speculative) execution of other transactions before the final outcome of the coordination is known. On LAN networks, experimental results show that the OAB-delivery order matches the AB-deliver order around 85% of the time [9]. This property establishes an upper bound on the amount of mis-speculations that can occur. SCert [4] uses this approach.

In this work we propose a novel approach, that builds on the strategies listed above but aims at exploring in higher degree the idea of executing transactions speculatively.



(a) Normal execution of a thread.

(b) Execution of a thread in SPECULA.

Figure 2: Comparison between SPECULA and the normal execution of a thread.

B. The Solution

Certification-based replication is an optimistic scheme. A transaction tx is locally executed assuming that there will be no conflicts with other transactions, and that, therefore, tx will commit. Only when the transaction terminates, will coordination with other nodes be established. On systems using standard certification-based replication schemes, while tx is being certified, the thread that is committing it stays blocked. The system that we propose here, named SPECULA, avoids blocking by letting the thread run based on a speculative snapshot of the system state (this snapshot may be invalidated when coordination is completed).

SPECULA integrates with a certification-based replication protocol and extends its default level of optimism. Figure 2b depicts the optimistic process, and we can compare it to a classic execution depicted in Figure 2a. In SPECULA, if transaction tx is locally valid, it is *speculatively committed* before the results of its global validation are known. This allows the committing thread to unblock and execute code while slow inter-node communication is taking place in background. New transactions can be executed in the same thread while tx is being certified. This creates a flow dependency among transactions. Both those transactions and concurrent ones gain access to the snapshot created upon the speculative commit of tx, so they can also depend on previous speculations, by reading speculatively committed data.

The trade-off of being more optimistic is having to undo more modifications when mis-speculations occur, i.e., when the global validation of a speculatively committed transactions fails. SPECULA has the ability to undo changes made to both the transactional and the non-transactional data of the application. Moreover, it resumes execution where the wrong speculation was initially performed.

Ideally, all speculation should be transparent for both the programmer and the application. It is also crucial that it guarantees the correct execution of the application, which means that the application should present the exact same behavior as if it was running over a non-speculative environment. As it will become clear later in the text, SPECULA achieves these goals, by controlling both the memory and the execution flow of the application.

IV. System Architecture

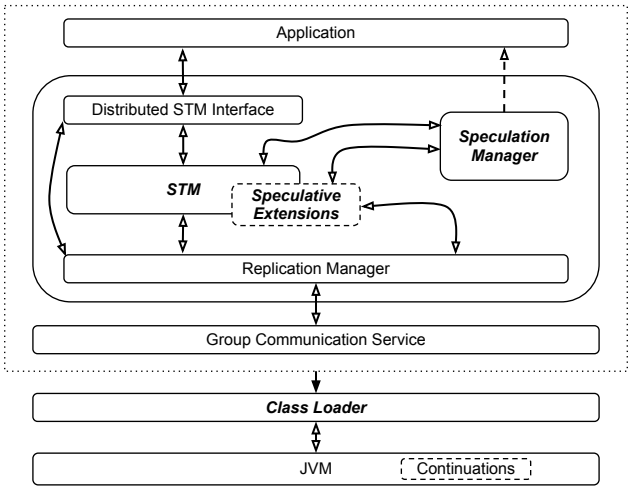


Figure 3: The system's architecture.

The architecture of each SPECULA replica is depicted in Figure 3, in which the components that are either new or that we have modified are highlighted in bold. These new and modified components add the following functionality to the system:

- The transactional engine was modified to support the notion of *speculative state*. SPECULA propagates the write-sets of speculatively executed transactions to their following transactions, while ensuring the correctness properties described in Section IV-A.
- The SPECULA approach requires that, in case of mis-speculation, it is possible to rollback the state of the application. This means undoing writes made to both transactional and non-transactional shared state, as well as rolling back the execution flow of threads that executed the aborted transactions, all in a way that preserves the system's correctness. To do this, we use a modified JVM that has support for capturing and resuming continuations [10], and developed *Class Loader* that makes modifications to classes prior to loading them. The STM was also enhanced with the ability to rollback changes made to the transactional state.
- The replication protocol was modified to be aware of the existence of speculative transactions, as their presence requires the execution of preventive measures, in order to preserve the correctness of the system.

It is possible to rely on a set of different technologies to implement all the functionalities described above. For our prototype, we decided to use the following software components: the JVM is a modified version[1] of the OpenJDK[2] that offers support for capturing continuations; the STM is the JVSTM, as it features Multiversion Concurrency

[1] `http://wikis.sun.com/display/mlvm/StackContinuations`
[2] `http://openjdk.java.net`

Control (MVCC), which makes it easier to add support for speculative state; the GCS is the Appia toolkit [11].

The *Class Loader* uses the ASM bytecode manipulation framework [12] to perform modifications to the application. These modifications intend to facilitate the life of the programmer, by injecting code that will automatically add support to each thread for capturing continuations and create all the data structures that the Speculative Manager (SM) requires. In turn, the SM will control the state of the application. This is represented in Figure 3, by the connector between the SM and the Application.

A. System Properties

SPECULA was designed to hold most of the properties of its underlying system. The Replication Manager ensures one-copy serializability (1SR). According to what Bernstein et al. wrote in [13], 1SR determines that the interleaved execution of clients' transactions must be equivalent to a serial execution of those transactions on a single replica. 1SR is more than a consistency model, and is the most common correctness criterion among transactional replication protocols.

SPECULA also maintains properties of the underlying STM, such as opacity and strong progressiveness. For the latter we should only consider transactions that do not depend on any speculation.

However, SPECULA does not guarantee that read-only transactions are abort-free, thus it is not multiversion permissive [14]. This limitation derives from the fact that read-only transactions can depend on speculations.

B. Operation

Although our prototype was implemented over a specific set of software components, the algorithms we developed are technology agnostic. Nevertheless, we assume that the underlying transactional engine features MVCC.

We shall start with the disambiguation of some concepts. A transaction is *speculatively committed* if, upon its commit request, it successfully passes on its local validation. The write-set of a speculatively committed transaction is applied locally, making it visible for future local transactions. The thread that executed the transaction proceeds normally, as the commit procedure returns. The system is, therefore, optimistic about the outcome of the transaction's global validation. The write-set of a speculatively committed transaction becomes available as *speculative state*.

We call *speculative transaction* to a transaction that depends on any speculation, i.e., one that has read speculative state or was executed over the coordination phase of a previously speculatively committed transaction.

A speculation ends up being either right or wrong. When the result of the global validation of a transaction is known, the node that executed the transactions has to act accordingly. If the speculation reveals itself correct, the transaction is *finally committed*, which leads to the system-wide apply of its write-set as *final state*. Otherwise, the system has to hide the transaction's write-set, which was made (locally) public during its speculative commit, and abort all transactions that depend directly or indirectly on the wrong speculation.

During its life, a transaction is always in one of the following states:

- *Executing:* the transaction is performing operations.
- *Committing:* the client has requested the commit of the transaction, and the the global validation procedure is taking place. In SPECULA, a transaction in this state is speculatively committed.
- *Committed:* the transaction was (final) committed.
- *Aborted:* the transaction was aborted.

1) Speculative Execution Support: In order to support the safe execution of both transactional and non-transactional code over speculative state, it is required to keep track of some data. Therefore, per thread, we maintain the following state:

- A FIFO queue containing all transactions that are in state *committing*.
- The oldest speculatively committed transaction that has been aborted since the last synchronization (described ahead).

Upon its start, a thread t gets associated with a `ThreadContext`, and vice-versa, in a one-to-one relationship between both. The `ThreadContext` data structure that holds information regarding the thread to which it is associated, namely, the state described above.

When a thread terminates, a synchronization is forced. Synchronizing consists in waiting until all transactions executed by the thread leave the state *committing*. If no transactions is aborted, the thread proceeds normally. Otherwise, the thread is rolled-back, i.e., all the modifications made to shared state performed on the thread since the speculative commit of the aborted transaction are undone, and the thread resumes its execution on the commit request of the oldest aborted transaction, from where the application is informed of the inability to commit that transaction.

Thread termination is not the only event that requires a synchronization point to be forced. All non-transactional operations, like for instance, the output of data to the screen, have to be preceded by a synchronization point. Each thread should also periodically check if any speculatively committed transaction was aborted. If that is the case, the thread should synchronize as soon as possible, as it is doomed to be rolled-back.

Continuations are essential for our speculative execution support. A continuation reifies the program's control state. A common analogy used to explain the concept is that continuations are the dynamic version of the GOTO statement, although much more powerful. From the perspective of our work, they can be seen as a snapshot of a thread, which saves the content of all its local variables and the program counter. Continuations allow us to, in case of mis-speculation, resume the execution at the precise moment where it should be resumed.

2) Modifications to the STM: Existing STMs have not been designed to take speculative state into account. We

propose a scheme that enables STMs to provide transactions with speculative values, while ensuring the whole system's correctness, namely the opacity property [15].

The modifications to the STM extend to the transaction initialization, read, validation, abort and commit procedures. Upon the initialization of a transaction, it it given a snapshot timestamp. During its life, a transaction always reads from the same consistent snapshots, based on the timestamp that it holds. A snapshot may or may not contain speculatively committed data.

Upon the commit request of a transaction, it is locally validated. If the transaction passes on its local validation, it is speculatively committed, which makes its write-set publicly available (locally). In standard MVCC, a transaction cannot commit if it conflicts (read–write conflict) with a concurrent but already committed transaction. This is the only reason why a transaction cannot commit.

In SPECULA, there are two additional reasons that may force a transaction to abort:

- If a transaction has read from the write-set of a speculatively committed transaction that was aborted – we call this a *speculative data dependency*;
- If a transaction was executed on a thread that has to be rolled-back and resumed at a point prior to the transaction's commit request – we call this a *speculative flow dependency*.

Locally valid transactions are announced to the network and subject to a global validation phase. A transaction that passes on its global validation is final committed, which leads to the system-wide availability of its write-set as final state. On the node where the transaction was executed and thus, speculatively committed, the final commit hides the equivalent speculative commit from future transactions. Reading from a speculatively committed version or from its equivalent final committed version is the same with regard to the validation procedure.

The abort of an executing transaction is delegated to the underlying STM, and thus follows standard procedures. Aborting a speculatively committed transactions requires hiding its speculatively committed write-set from future transactions and invalidating it, so that other transactions that have read from it also fail to commit (this is what we described as speculative data dependencies). Moreover, the corresponding `ThreadContext` has to be notified, since all transactions that were executed after, and on the same thread as the aborting transaction, also have to be aborted (what we described as speculative flow dependencies).

3) Speculative Execution Control: High conflict scenarios are not favorable for a speculative scheme like the one we propose. Mis-speculations introduce unwanted load on the system. The overhead created by SPECULA only pays off if a large faction of all speculatively committed transactions are final committed. Notice that in Section III-A, we stated that experimental results have shown that communication time dominates execution time in DRSTM environments. This means that a thread can execute a high number of transactions in the time required to by a single broadcast.

We decided to implement a scheme that limits the amount of speculations per thread. The algorithm derives from the *additive increase/multiplicative decrease* algorithm used in TCP [16]. Its key heuristic is that successful speculations should allow more speculations to occur, while, conversely, mis-speculations should decrease the maximum number of speculations allowed. At any given time, the number of transactions executed by a thread that are in state *committing* is never smaller than the value of the $MIN\_COMMITING\_TXS$ variable and never greater than the value of the $MAX\_COMMITTING\_TXS$ variable.

This scheme enables the system to better adapt to its surrounding environment.

4) Restoring Non-transactional State: Few application are fully transactional. Most make use of transactions to access shared state, but perform heavy computations and system calls outside of them. Changes made to non-transactional state have be undone in case of mis-speculation, just like the changes made to transactional state. In order to restore the state of the non-transactional shared memory, at the point where execution is resumed, we propose the construction of undo logs. Since we are using continuations to snapshot the execution flow of a thread, and continuations save the state of the thread's stack, we only need to deal with writes made to non-transactional shared state.

To build an undo log, all writes to non-transactional shared state have be intercepted. When an instruction executed on thread t tries to modify non-transactional shared state, we save the value that is in target address before it is overwritten with the new value. An undo log keeps just one value per memory address: the oldest. Each saved value is kept in the undo log that is associated with the last speculatively committed transaction that was executed on t. If later, a synchronization procedure has to resume the execution of t using a continuation, it applies all undo logs required to restore the state that was available when the continuation that is going to be resumed was captured.

5) Correctness Arguments: We now provide some informal arguments to show that our algorithm is correct.

All replicas start with the same state and final commit the same transactions, by the same order. Therefore, the final state is always kept coherent among all nodes. This guarantees 1SR, as it is easy to see that, if all replicas go from snapshot s_i to snapshot s_{i+1}, then the interleaved execution of transactions among all nodes produces a serialization order that matches the one produced by a serial execution, i.e., both executions would end up producing the same snapshot s_n.

The key to guarantee opacity is ensuring that every transaction always reads from the same consistent snapshot. SPECULA's read procedure ensures this by forcing transactions to read from the newest snapshot that was available when they began. Moreover, since no snapshot is ever modified after its creation, and the apply of the write-

sets of remote transactions is preceded by a clean-up of the speculative state, temporary inconsistent states never exist. This suffices to guarantee opacity.

Notice that local validation ensures that all speculative commits create potentially serializable snapshots. If a transaction always reads from the same consistent snapshot, its commit produces a new guaranteed consistent snapshot. It is not even possible for a transaction to build an inconsistent write-set.

Finally, as we assume that the program's correctness does not depend on any kind of synchronized access to non-transactional shared state, we are free to undo all modifications made to it.

C. Strengths and Weaknesses

SPECULA makes a trade-off between the latency required to commit a transaction and the amount of memory consumed by the middleware. With SPECULA, transactions can (speculatively) commit based on local information, and transaction processing can continue with no further delays. This is achieved at the cost of having to continuously create snapshots of the system. Therefore, SPECULA consumes more memory than other non-speculative transactional memory systems. However, the increase in memory usage occurs for a limited amount of time, and one that is expected to be short, as the final commit of a transaction hides its speculative commit from new transactions, and most memory transactions are themselves very short, as stated in Section III-A. SPECULA has also the drawback of requiring the write-set of a transaction to be applied twice: once when the transaction commits locally and again, when the transaction is applied system-wide. We expect this cost to be paid off by the overlap of communication with computation.

On the other hand, new transaction are always provided with a consistent snapshot of the system, and are shielded from the concurrent execution of other transactions. Thus SPECULA preserves the programming model that makes software transactional memory appealing. Furthermore, in face of low-conflict workloads, cascading aborts due to speculation rarely occur, enabling SPECULA to effectively hide network delays.

V. Evaluation

A. Experimental Environment and Settings

All the experiments presented here were performed in a cluster of eight nodes, each one equipped with two Intel Xeon E5506 at 2.13GHz and 8 GB of RAM, running GNU/Linux 2.6.32 – 64 bits. The nodes are interconnected via a private Gigabit Ethernet switch.

The JVM is a development snapshot of version 1.7.0 of the OpenJDK. The AB service provided by the Appia GCS toolkit [11] uses a a sequencer-based algorithm to order messages on top of a multicast layer that relies on point-to-point TCP links [17], [18]. It also implements batching of the sequencer messages. We have set the batching value either to the same value of $MAX\_COMMITING\_TXS$, or to one (i.e., no batching) on the baseline system. The batching timeout is set to 250 *ms*.

Unless stated otherwise, all runs were executed with a single thread per node producing transactions. The coordination among replicas is achieved using a non-voting certification-based replication protocol. We compare our prototype with that of a system using the same configuration but with the speculative extension turned off.

We assume a that the environment is stable, in which no nodes crash/stop or deviate from their normal behavior.

B. Evaluation Criteria

The two main evaluation criteria for our system are speedup, and the average time required to execute and commit a transaction. We also consider, although as secondary criteria, the abort rate, the number of speculatively executed transactions that were committed, the overhead in the local execution of transactions and the amount of time that worker threads were halted by the execution control algorithm. These criteria allow us to measure the efficiency and the effectiveness of our system. Ideally, a system should achieve high throughput and low latency. However, in most practical system, there is a trade-off between these two aspects. In this evaluation we aim at assessing if SPECULA achieves a reasonable compromise between these two criteria.

The number of speculatively executed transactions that were committed and the overhead in the local execution of transactions allows us to assess the the effectiveness and the benefits of being speculative.

C. STMBench7 Benchmark

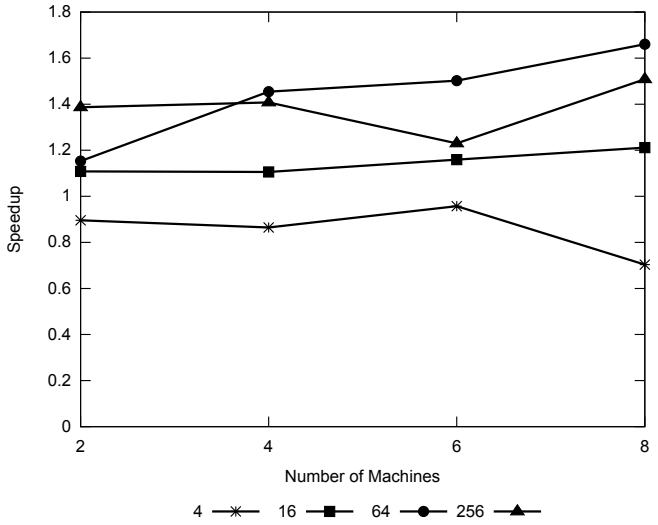
Description: The STMBench7 benchmark was introduced by Guerraoui et al. in [19] as a complex benchmark that generates realistic workloads, by mimicking the operations present in large object-oriented applications, such as CAD/CAM applications. Operations vary significantly in complexity, containing both short and trivial transactions that read a few items, and highly read/write intensive ones, which perform not only deep structural modifications to an object graph with millions of vertices, but also do long transversals on the same graph, resulting in transactions with huge datasets.

The configuration parameters of this benchmark allow the selection of one of three different workloads: one read-dominated, one with a balanced mix of read-only and update transactions, and one write-dominated. It is also possible to enable/disable the execution of both long transversals and deep structural modifications.

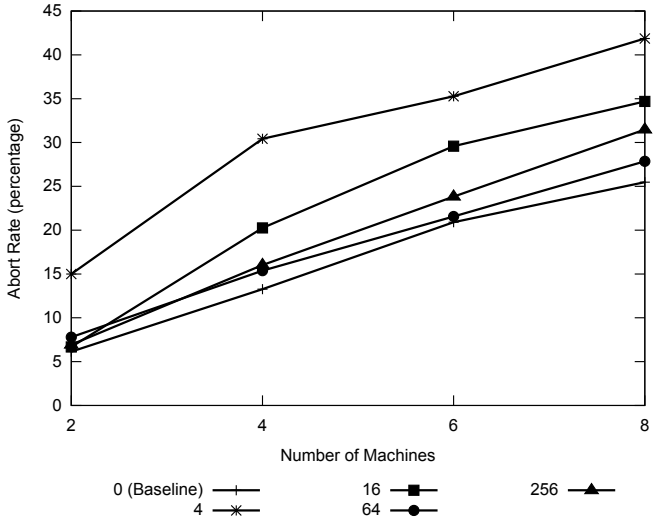
Configuration: We analyze the results of both write and read-dominated workloads. Long transversals and deep structural modifications were disabled, otherwise the number of conflicts becomes unbearable.

The operation ratios (in percentage) of the write-dominated workload are depicted in Table I, and of the read-dominated workload in Table II.

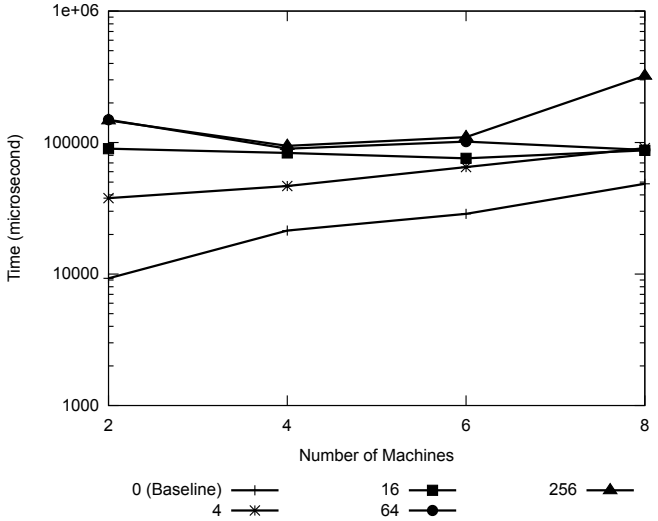
Results: All the results here presented are relative to a 60 seconds run of the benchmark.



(a) Speedup – varying the value of $MAX\_COMMITTING\_TXS$.



(b) Abort rate – varying the value of $MAX\_COMMITTING\_TXS$.



(c) Transaction total execution time – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4: STMBench7 – Write-dominated workload.

TRAVERSAL:	0.00
TRAVERSAL_RO:	0.00
SHORT_TRAVERSAL:	0.00
SHORT_TRAVERSAL_RO:	47.06
OPERATION:	47.65
OPERATION_RO:	5.29
STRUCTURAL_MODIFICATION:	0.00

Table I: STMBench7 – Write-dominated workload – Operation ratios.

TRAVERSAL:	0.00
TRAVERSAL_RO:	0.00
SHORT_TRAVERSAL:	0.00
SHORT_TRAVERSAL_RO:	47.06
OPERATION:	5.29
OPERATION_RO:	47.65
STRUCTURAL_MODIFICATION:	0.00

Table II: STMBench7 – Read-dominated workload – Operation ratios.

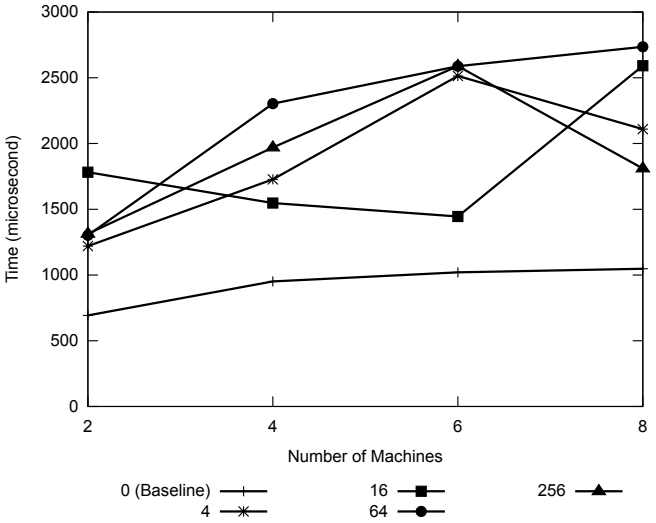
Write-dominated Worload: Figure 4a depicts the speedup achieved in this configuration. For values of $MAX\_COMMITTING\_TXS > 4$ we always observed positive speedups, with a maximum gain of 1.66 times when $MAX\_COMMITTING\_TXS = 64$.

The abort rate depicted in Figure 4b indicates that this configuration suffers from a big unbalance in the abort rate distribution among nodes. The abort decrease as we increased the value of $MAX\_COMMITTING\_TXS$ because one node of the group starts committing most of its transactions while the remaining nodes keep seeing their transactions being aborted. As the node that keeps committing becomes more speculative, it starts committing even more transactions, which leads to the decrease in the abort rate.

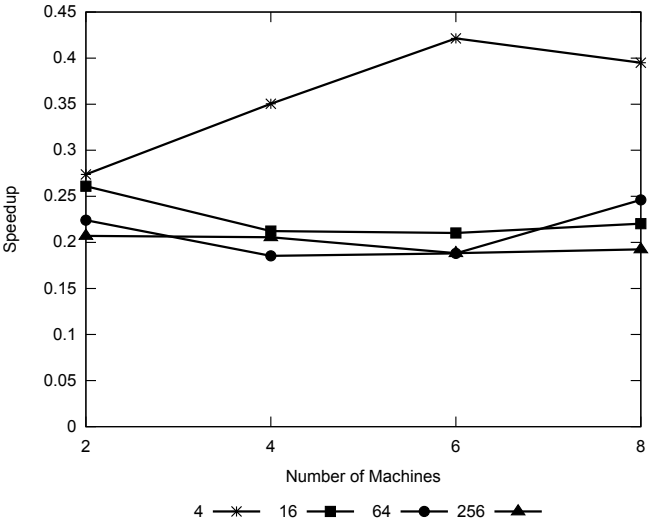
Figure 4d depicts the local execution time of transactions. The maximum overhead registered was 257%. In MVCC, the longer a transaction is, the more expensive is for it to read, as it has to iterate over the versions of newer snapshots to find the version of its snapshot. Also, the longer the transaction, the higher the probability that there were in fact other transactions committing during its execution. Besides this, in SPECULA, writes to non-transactional shared state are logged in undo logs, and it represented 15% off the total time of execution (however, using the undo logs to restore non-transactional shared state accounted for less than 0.01%), because STMBench7 has a rich application logic. Moreover, although the configuration we evaluated was the so called write-dominated, it has a ratio of read-only transactions over 50%, and read-only transactions incur into a meaning performance penalty in SPECULA, as they their read-sets have to be tracked and validated.

Figure 4e depicts the network latency values that were observed. It was abnormally high for a LAN environment, even for the baseline system.

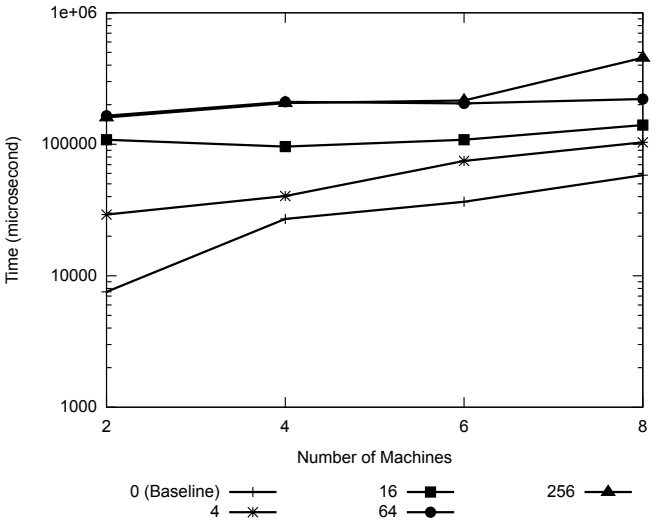
The percentage of speculatively executed transactions that were final committed is very significant. For instance, with



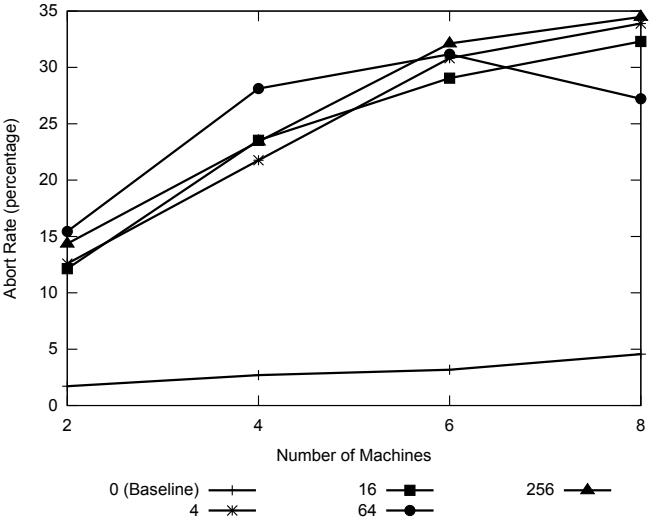
(d) Transaction local execution time – varying the value of $MAX\_COMMITTING\_TXS$.



(a) Speedup – varying the value of $MAX\_COMMITTING\_TXS$.

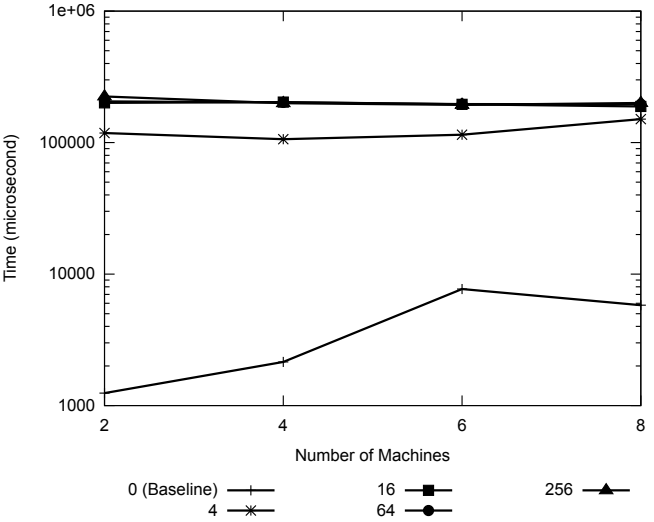


(e) Network latency – varying the value of $MAX\_COMMITTING\_TXS$.



(b) Abort rate – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 4: STMBench7 – Write-dominated workload.



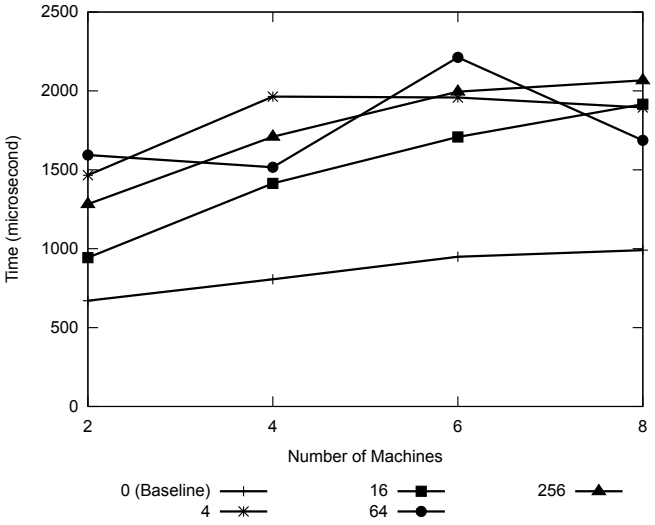
(c) Transaction total execution time – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 5: STMBench7 – Read-dominated workload.

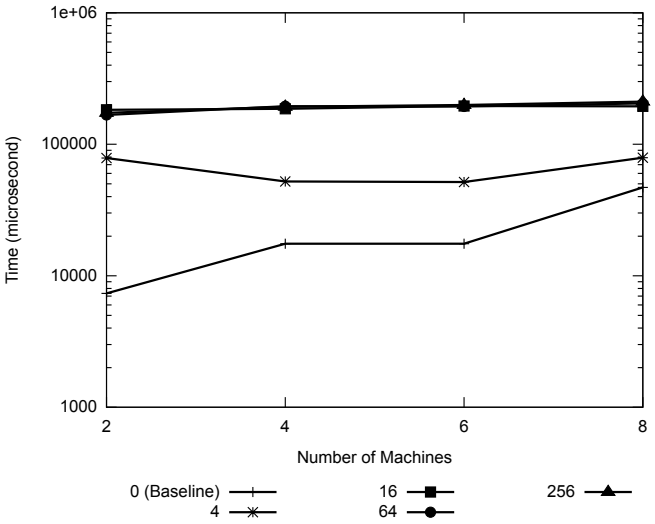
eight machines and $MAX\_COMMITTING\_TXS = 256$, more than 99% of all committed transactions had a speculative dependency. Unfortunately, for the same scenario, the execution control algorithm kept the worker threads blocked for an average of 46 s.

With $MAX\_COMMITTING\_TXS = 64$ the system achieved a significant speedup and a consistent response latency, therefore, based on these results it is a adequate value for high contention scenarios with not so short transactions.

Read-dominated Workload: Figure 5a depicts the speedup achieved in this configuration. As expected, it is negative, since as already stated, read-only transactions incur into a significant overhead that comes from the need of tracking and validating their read-sets. Furthermore, the application logic in this configuration is even richer than



(d) Transaction local execution time – varying the value of $MAX\_COMMITTING\_TXS$.



(e) Network latency – varying the value of $MAX\_COMMITTING\_TXS$.

Figure 5: STMBench7 – Read-dominated workload.

in the write-dominated workload, and so building undo logs accounted for 20% of the whole execution. Using them to restore non-transactional shared state accounted for less than 0.01%.

Figure 5b depicts the abort rate observed. Again, as expected, it was significantly higher than the one verified in the baseline system, which is totally normal as it ensures that read-only transactions are abort-free and SPECULA does not. For instance, with eight machines and $MAX\_COMMITTING\_TXS = 256$, from all the 1222 aborted transactions, 909 were read-only.

Figure 5d depicts the local execution duration of transactions that was observed. The explanation for these results is similar to the one present in the write-dominated configuration.

Figure 5c depicts the total execution time of transactions. The graphic is however misleading, because we have to associate read-only transactions with update transactions, in order to validate them according to a global serialization order. Therefore, the average total execution time of read-only transactions is similar to the total execution time of update transactions, although they are executed in a totally uncoordinated fashion.

Figure 5e depicts the network latency observed. Since this workload is composed by few update transactions, the network latency represents a minor issue in this case.

Unfortunately, all nodes remained blocked by the execution control algorithm for most of the run: 48 *s* with eight machines and $MAX\_COMMITTING\_TXS = 256$. This was not due to the network latency but to the high abort rate, which caused the execution control algorithm to reduce the level of optimism, allowing less transactions to be speculatively committed. However, for in the same scenario there was also a high amount of speculatively executed transactions were final committed: almost 99% of all committed transactions had a speculative dependency.

VI. Conclusions

DRSTMs allows programmers to develop highly concurrent and dependable systems with minimal effort. Unfortunately, the implementation of this powerful abstraction with good performance is still an open challenge.

To this end, this paper presents SPECULA, a novel system that can speculatively execute transactional and non-transactional code in a safe manner, by being able to rollback all changes made to memory if a mis-speculation is detected. SPECULA relies on a certification-based replication scheme to enable the speculative commit of memory transactions. This allows transaction processing to proceed while the global serial order of (speculatively) committed transactions is defined in background. For this purpose, SPECULA manages access to speculatively committed data and performs modifications to the application at the bytecode level that enables it to undo writes made to non-transactional shared state, and also to rollback the execution flow of threads, all in a fully transparent fashion for both the application and the programmer.

Experimental results show that SPECULA achieves significant speedups in low contention scenarios and can be also useful in high contention scenarios. By extending the level of optimism of standard certification-based replication solutions, SPECULA promotes a better use of the computational resources available in the system.

SPECULA does not address the problem of optimizing network usage, a topic that is orthogonal to the focus of this work. Therefore, as other certification-based approaches, the costs of inter-replica synchronization can saturate the underlying group communications system, with the resulting penalties in the resulting coordination latency. This fact constrained the system's throughput in low contention scenarios and made its responsiveness significantly lower as the level of optimism increased.

As future work the system would benefit from better execution control mechanisms, feedback and auto-tuning schemes, in order maximize its throughput without affecting its responsiveness. Static analysis of the application's code can also help to reduce the overhead of building undo logs, by identifying write operations that never need to be undone. A reevaluation of the system in a low network latency environment should also be performed to further validate this work. To this end, the performance of the system should be assessed with various AB protocols (e.g., token based) and GCSs (e.g., JGroups).

ACKNOWLEDGMENTS

This work was partially supported by the "ARISTOS" (PTDC/EIA-EIA/102496/2008) project, by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds, and by the European Union, through the "Cloud-TM" (257784) project. Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Nuno Carvalho and Paolo Romano.

REFERENCES

[1] J. Cachopo, "Development of Rich Domain Models with Atomic Actions," Ph.D. dissertation, Technical University of Lisbon, 2007.

[2] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2stm: Dependable distributed software transactional memory," in *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, ser. PRDC'09. Shanghai, China: IEEE, Nov. 2009, pp. 307–313.

[3] R. Palmieri, F. Quaglia, and P. Romano, "Aggro: Boosting stm replication via aggressively optimistic transaction processing," in *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications*, ser. NCA'10. Cambridge, MA, USA: IEEE, Jul. 2010, pp. 20–27.

[4] N. Carvalho, P. Romano, and L. Rodrigues, "Scert: Speculative certification in replicated software transactional memories," in *Proceedings of the 4th Annual International Systems and Storage Conference*, ser. SYSTOR'11. Haifa, Israel: ACM, May 2011, p. 10.

[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, jul 1970.

[6] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems - concepts and designs (3. ed.)*, ser. International computer science series. Addison-Wesley-Longman, 2002.

[7] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proceedings of the 2nd ACM Workshop on Large-Scale Distributed Systems and Middleware*, ser. LADIS'08. Watson Research Labs, Yorktown Heights, NY, USA: ACM, 2008, (invited paper).

[8] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho, "Evaluating database-oriented replication schemes in software transactional memory systems," in *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS'10. Atlanta, GA, USA: IEEE, Apr. 2010, pp. 1–8.

[9] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 1018–1032, Jul. 2003.

[10] C. T. Haynes, D. P. Friedman, and M. Wand, "Continuations and coroutines," in *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*. Austin, TX, USA: ACM, Aug. 1984, pp. 293–298.

[11] H. Miranda, A. S. Pinto, and L. Rodrigues, "Appia: A flexible protocol kernel supporting multiple coordinated channels," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, ser. ICDCS'01. Phoenix, AZ, USA: IEEE, May 2001, pp. 707–710.

[12] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable systems," in *In Proceedings of Adaptable and Extensible Component Systems*, Grenoble, France, Nov. 2002.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[14] D. Perelman, R. Fan, and I. Keidar, "On maintaining multiple versions in stm," in *Proceedings of the 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC'10. Zurich, Switzerland: ACM, Jul. 2010, pp. 16–25.

[15] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP'08. Salt Lake City, UT, USA: ACM, Feb. 2008, pp. 175–184.

[16] M. Allman, V. Paxson, and W. Stevens, "RFC 2581: TCP congestion control," 1999.

[17] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[18] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[19] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," in *Proceedings of the 2007 EuroSys Conference*, ser. EuroSys'07. Lisbon, Portugal: ACM, Mar. 2007, pp. 315–324.