Increasing the Scalability of a Software Transactional Memory System

Faustino Dabraio da Silva
Mestrado em Engenharia Informática e de Computadores
Instituto Superior Técnico
Universidade Técnica de Lisboa
faustino.silva@ist.utl.pt

Abstract

Software Transactional Memory (STM) introduces the transactional model of database systems to software programming, with the aim of being a simpler and more modular alternative to locks in concurrent programming. While most existing STM engine proposals are successful in delivering the promises of increased simplicity and modularity, there is room for improvement in terms of performance.

In this work I started by analyzing existing STMs to understand how they work and what different design alternatives exist. Then, after analyzing how the Java Versioned Software Transactional Memory (JVSTM) works, I identified potential improvements to its key data structures and algorithms, implemented them and evaluated whether or not (and why) they had the intended impact on JVSTM's performance.

1. Introduction

Although the availability of multiprocessor computers is nothing new in the server domain, it's only in recent years that multicore processors became commonplace across most computing devices. However, this hardware potential does not translate into real performance gains unless applications are specifically designed to take advantage of parallel computing. Traditionally, programmers have turned to locks as the favorite tool to introduce parallelism into their applications. However, fine-grained locks become extremely error-prone and complex to design in large applications [1]. On the other hand, coarse-grained locks, while less complex and error-prone, have limited parallelism and thus poor performance [2, 1].

While the lock-based approach was somehow tolerable when concurrent programming was somehow a niche feature, the increased popularity of multicore processors demands a more intuitive and higher-level method for programmers to write concurrent applications.

To this end, Software Transactional Memory (STM) brings the transactional model, already proved and tested for several decades in the database community, to concurrent programming [3]. In simple terms, when using an STM system, the programmer only needs to identify operations that run concurrently and that share the same memory space, and divide them into transactions. With this information on hand, the STM system monitors which shared memory locations are accessed by which transactions and determines the presence of conflicts.

However, the convenience of STMs compared to traditional approaches doesn't come without trade-offs. Keeping track of different transactions, and the memory locations they access, introduces space and time overheads over lock-based approaches. To be a viable substitute for locks, STMs have to reduce the overhead they introduce as much as possible while maintaining the ease of use.

1.1. Goals

The aim of this work is to analyze current STM systems and propose improvements to the scalability of one such system, the Java Versioned Software Transactional Memory (JVSTM). This system was chosen as the basis for this work due to the support of its authors and contributors.

Section 2 starts by describing some basic concepts and key design alternatives of STMs in general, Section 3 compares a few existent systems with different approaches, and Section 4 describes the JVSTM, the system whose scalability this work aims to increase, in detail to identify possible areas of improvement.
Section 5 presents several ideas on how JVSTM's structures and algorithms could be changed to address potential problems, and also describes the results of those proposals.
Finally, Section 6 briefly sums up the key conclusions of this work and identifies areas that could be further developed in the future.

2. Software Transaction Memory

2.1. Basic Concepts of STMs

2.1.1. Transactions

The notion of transaction when applied to software translates into a set of operations, specified by the programmer, that is to be executed atomically.
After a transaction starts, it performs a sequence of write and/or read operations on shared memory objects and may terminate in two ways. If it executes successfully without entering into conflicts with other transactions, we say the transaction *commits*, i.e. its operations become effective and its results become accessible to new transactions. However, if during the execution

or at commit-time conflicts are detected, the transaction *aborts* and is rolled back, which means it doesn't produce any changes to the system and must be restarted.

In databases, transactions have to respect the ACID properties [4]: Atomicity, Consistency, Isolation and Durability. In software transactions, usually only the first three properties are considered [5]. Durability implies that the results of a successfully committed transaction should be stored persistently, which isn't crucial to all application domains, and as such several STMs ignore this property.
From these basic properties, others such as *linearizability* and *Opacity* [6] can be obtained. STM developers decide which properties they want to ensure depending on what they want their STM to guarantee, and that leads to different levels of implementation complexity.

2.1.2. Transactional Memory

Also referred to as *shared memory*, this term refers to the part of the address space of an application that is accessed concurrently by transactions. These are the accesses that the STM engine has to monitor and synchronize in a way that minimizes conflicts as much as possible.

2.1.3. Read and Write Sets

These sets are usually associated to each transaction and help the STM engine keep track of the transactional data read and written by each transaction, which is useful to detect conflicts between transactions. *Read sets* keep track of each read memory location — either the value itself or a version number —, while *write sets* provide a way for transactions to defer the update of transactional objects to commit-time.

2.1.4. In-place/Out-of-place Updates

A transaction that wishes to write some value in shared memory may do so in two different ways depending on how the STM engine synchronizes accesses to transactional memory: *in-place updates* happen when a transaction writes tentative values to the shared memory location immediately during its execution, whereas with *out-of-place updates* the transaction stores tentative values in its local write set and only writes them to shared memory on commit.

2.2. Design Alternatives

STM has been researched as an alternative to locks for more than a decade [3], and as such several different proposals exist [1, 7, 2, 8, 9]. This section describes the most relevant features and design alternatives that differentiate STM systems from one another, presenting their corresponding strong points and drawbacks.

2.2.1. Synchronization Technique

The ultimate goal of STM systems is to allow several transactions to run simultaneously and access shared data concurrently. Because two transactions cannot access the same shared memory location simultaneously when that constitutes a conflict, the STM system has to somehow synchronize those accesses. This synchronization can be either blocking or non-blocking [7].

The *blocking synchronization* technique associates locks to shared data and presupposes that those locks are acquired by the transaction prior to data being effectively changed and released after the transaction commits or aborts. Also, these locks can be acquired either at encounter or at commit-time, depending on whether they're acquired immediately during execution, or only later when the transaction is committed.

STMs that rely on *non-blocking synchronization* do not operate directly on the shared data, but on copies of it instead. Each of these copies is local to a single transaction and thus can be manipulated through the whole lifecycle of the transaction without synchronization concerns. On commit, the shared data is atomically overwritten with the new values. Non-blocking synchronization can have progress guarantees with different strengths [10]: *obstruction-free*, *lock-free* and *wait-free*.

2.2.2. Transaction Validation

Before the results of a transaction become definite, i.e. before it can be successfully committed, the STM engine must first validate it. Essentially, validating a transaction consists in checking whether committing it would go against any of the properties an STM system is expected to preserve, i.e. if it conflicts with any other transaction [1]. This is done by observing which shared data locations a given transaction has accessed for reading and/or writing and it can be done at two different times: *pessimistic* STMs assume that a transaction can enter a conflict at any time and thus check for conflicts each time the transaction tries to access the shared memory space, whereas *optimistic* STMs assume all transactions will be successful and thus allow them to run speculatively and defer all validity checks to commit-time.

3. Related Work

With the goal of understanding how current STMs work, what problems their developers were faced with and how they solved them, I analyzed 5 existing systems. Some systems were chosen because they are common references in related literature, and others because they offer a different combination of design choices.

For the sake of brevity, I will not describe each system in-depth. Instead, Table 1 presents a comparison between all systems using some of the most relevant design alternatives presented earlier, and Section 4 describes the JVSTM with more detail, as that is the system on which this work will be based.

Note that the memory update policy in blocking STMs is closely related to how locks are acquired: blocking STMs that update memory in-place require encounter-time lock acquisition, whereas out-of-place updates only need locks to be acquired at commit-time.

	DSTM	McRT	TL2	CS	LBP
Synchroniz. technique	Non-blocking	Blocking			
Data granularity	Object-based	Word or object-based			Object-based
Concurrency control	—	Versioned write locks			R/W locks
Lock acquisition	—	Encounter-time	Commit-time	Encounter-time	Commit-time
Memory update	In-place	In-place	Out-of-place	In-place	Out-of-place
Lock placement	—	Adjacent or separate			?
Validation	Pessimistic			Pess. or Optim.	Pessimistic

Table 1: Comparison between STMs

4. JVSTM

In this section I will describe the JVSTM more thoroughly, with a special focus on the features and data structures that are more relevant for the optimizations I will propose in Section 5.
JVSTM started off as a lock-based system [11, 8], but over time it was subjected to several revisions, and eventually a lock-free version [12] was developed, but the concept of versioned boxes is common to all versions. Unless specifically stated, the features and behaviors described henceforth refer to the lock-free version, as this is the version this work will try to improve upon. Moreover, there are other iterations of the lock-free JVSTM that use different data structures, but those won't be described here to avoid confusion.

4.1. The Versioned Memory Model

The most distinctive feature of JVSTM is its multi-version memory model. Under this model, each shared memory location is represented by a **versioned box**, implemented by the `VBox` class described in Section 4.3.1. Each box encapsulates multiple versions of a shared memory location, i.e. it contains that location's history. Each value is associated with a version number that corresponds to the final number of the transaction that committed that version. Like other STMs' memory locations, versioned boxes support two operations — read and write. While reads happen during a transaction's execution, writes to a box only happen at commit-time; tentative values are written in the transaction's write set.

Having multiple versions of the same memory location allows JVSTM to guarantee that all read-only transactions always complete successfully, because even if another transaction commits a newer value to that box, the read-only transaction can still access the version it needs. This guarantee also eliminates the need for validation in this kind of transactions. Because read-only transactions always observe a consistent state that is defined by the number they were assigned at start, it follows that such transactions are linearizable at the time they started [8].

On the other hand, write transactions can only be linearized at the time they commit, because that's when their changes become visible to the rest of the system. However, they ran with a consistent view of the system at the time they started, which means these transactions' read sets have to be validated at commit time, as explained in Section 4.2, to ensure their validity at that point. Because write-only transactions have an empty read set, their validation is always successful and they can always be linearized at commit-time, which means that, like read-only transactions, they never abort.

4.2. Transaction Lifecycle

In JVSTM, a typical top-level transaction's lifecycle can be divided into several main stages.

The **begin** phase is the shortest of all three and essentially consists in initializing fields and data structures that will be used in the subsequent phases. For example, it's in this phase that a transaction is assigned a temporary version number, the same as the most recently committed transaction's number, that will define the context under which the transaction will run, i.e. which version of the transaction memory it will "see".

The **execution** phase is when the transaction's actual work is done by reading and/or writing boxes. In the case of write transactions (both write-only and read/write), read and write operations are logged in the transaction's read and write sets, respectively. On the other hand, read-only transactions don't require a read set because they don't need to be validated, as explained in Section 4.1. The absence of a read set means read-only transactions have a lower overhead than write transactions, which was one of JVSTM's original goals [8]. Write-only transactions are also guaranteed to commit successfully, but their write operations must be logged so that other (read/write) transactions can validate themselves against the write-only ones.

Validation and commit: when a write transaction finishes the execution phase without conflicts, this third stage can be divided into three parts — write-back (first), validation and write-back (second), whereas those that encountered a conflict during execution skip this stage and abort.
The first write-back step ensures that the write sets of all transactions that are already validated and in queue to be committed are written-back, i.e. the values they've written are made permanent.
The first step of validation, called snapshot validation, consists in validating the transaction's whole read set by checking whether any box was updated in the meanwhile by another transaction.
If snapshot validation succeeds, the transaction tries to put itself in the commit queue by performing a CAS

operation. If the CAS succeeds, the transaction enters the queue and doesn't need further validation, otherwise we know that another competing transaction entered the queue, and thus the transaction whose CAS failed must revalidate itself by performing incremental validation, which consists in looking for intersections between the transaction's read set and the other transaction's write set.
Once the transaction has successfully validated itself and entered the commit queue, it again helps all pending transactions in the queue, up to and including itself, to write-back their write sets.

Abort: when a transaction aborts, no permanent changes were made to the shared boxes and the transaction wasn't put in the commit queue, which means that no actions need to be rolled back to keep the system consistent. Instead, an exception is thrown to let the user decide how to proceed (typically restart the transaction).

4.3. Important Data Structures

4.3.1. `VBox`

As previously explained, JVSTM's most distinctive feature is that each memory location is encapsulated in a box capable of holding several versions of a shared variable. Each box is implemented using the `VBox` class, which has three fields: `tempValue`, `currentOwner` and `body`.

The first field has the purpose of allowing a running transaction, identified by the `currentOwner` field, to write a tentative value directly into the box instead of using the traditional write set.

The last field, `body`, references a linked list of bodies, where each body holds one committed version of the memory location. Tentative values are either stored in the `tempValue` field or the running transactions' write sets. When a new version is committed by a transaction, a new body with the new value and version number is created and inserted into the start of the list using a CAS.

When a transaction wants to read directly from a box during execution, it can't simply read the most recent version. Instead, it must traverse the bodies list to look for a version compatible with its own transaction number in order to guarantee that it observes a consistent view of the system.

4.3.2. Read Set

Read sets are stored as a list of `VBox` arrays, implemented using cons pairs. By default, each array has a capacity of 1.000 boxes.

Whenever a transaction reads a value from a box, JVSTM places a reference to that box in the first array (the "active" array) of the list. An integer field (`next`) keeps track of the next position in the active array to be used. When this value is below zero, JVSTM pushes a new, clean array into the top of the list and resets the `next` field so that further reads are stored in the new array.

4.3.3. Write Set During Execution

In a write transaction, the `setBoxValue` method, responsible for tentatively writing a value into a box, may opt to record write operations in one of two fields: `boxesWritten` or `boxesWrittenInPlace`. The former is a Java `HashMap` that maps a box to the tentative value written by the transaction and the latter is a list of box references implemented using cons pairs.

To decide where to put a reference to the box being handled, the `setBoxValue` method uses the box's `currentOwner` and the transaction's `orec` fields, which are ownership records. The `OwnershipRecord` class essentially has one integer field, called `version`, that indicates whether the transaction associated to that ownership record is running, aborted or committed.

In summary, a transaction's write set is split into two parts: the boxes where it was able to write directly into the `tempValue` field (`boxesWrittenInPlace`) and the boxes where it had to record the tentative value locally (`boxesWritten`). This division has the goal of speeding up scenarios where the same transaction writes into the same box more than once and where it reads from a box after having written into it.

4.3.4. Write Set After Execution

In the lock-free implementation of JVSTM, all transactions that have finished running but are still waiting their turn in the commit queue take part in the process of writing back other transactions' write sets. Because several transactions will be writing back the same write set, it needs to be converted to a structure that combines both previous structures used during execution (the `boxesWritten` map and the `boxesWrittenInPlace` list) and allows the write-back work to be divided between transactions while minimizing duplicate work.

This conversion happens right after the transaction finishes running and just before it tries to enqueue itself in the Active Transactions Record. The new write set is stored in the respective transaction's record and may be accessed by other transactions for write-back and validation purposes.

To combine both previous structures, the `WriteSet` class contains an array with references to all boxes where the transaction wrote. This array allows each helping transaction to be assigned a block of boxes to write back.

This write set can be used for two purposes — validation and write-back. For the former, transactions simply need to go through the box array sequentially to perform incremental validation.

To take part in the write-back phase, a helping transaction chooses a starting block at random so that not all transactions start at the same block. Then it processes all blocks from that point on sequentially until it arrives back at the starting block, ignoring blocks that it observes were already written back by other transactions.

4.3.5. `ActiveTransactionsRecord`

In JVSTM's original lock-based version, the Active Transactions Record (ATR) was essentially a list of ac-

tive transactions kept for garbage collection purposes. It's implemented as simple linked list and each record therein contains information about a previously committed write transaction to help determine when the values it wrote can be safely garbage collected. Each record has 4 fields: the number of the transaction it refers to (`transactionNumber`), a list of the bodies written by that transaction (`bodiesToGC`), the number of transactions that are still running under the same version number (`running`) and a reference to the next record (`next`).

The lock-free version of JVSTM extends the ATR's functionality — transactions enter the ATR as soon as they are successfully validated and record now also hold their respective transaction's `WriteSet`. Instead of merely representing the history of previously committed transactions, the new ATR also represents the order in which valid transactions will be committed. Moreover, by including each transaction's write set in the ATR, JVSTM is now able to perform the incremental validation process that allows it to be lock-free.

5. Proposed Optimizations

5.1. Implement the Read Set Using an Identity Hash Set

To keep track of each transaction's read set, JVSTM uses a list of arrays to store references to each box that the transaction reads. The simplicity of this structure makes it very fast to insert a new box in the read set, in observance of JVSTM's requirement of low overhead read accesses.

Lookup operations are very simple too, but also expensive because they require a linear search, and this cost is of great significance in the incremental validation process, which must guarantee there are no intersections between the validating transaction's read set and another transaction's write set.

To speed up the lookup process, I propose to use an Identity Hash Set (IHS) as a substitute for the list of `VBox` arrays currently in use to store the transaction's read set. Essentially, the `IdentityHashSet` class has the same properties as a traditional Java `HashSet`, except all key comparison operations are implemented using reference-equality instead of object-equality, in an effort to make them as fast as possible and reduce overhead. The use of reference-equality is possible in JVSTM's context because we know that the only way for two `VBox` objects to be equal is if they are one and the same. In an effort to be even more efficient, the hash function doesn't rely on remainder operations as `Hashtable` does. Instead, the `IdentityHashSet`'s size is always set to a power of 2, so that the hash function can be implemented using a bitwise AND operation.

This new structure allows for much faster lookups because the read set is split in several buckets and each lookup will only be done within the bucket determined by the hash function. Moreover, its different insertion process also avoids duplicate entries, which would be too costly to guarantee using the current array implementation.

However, the `IdentityHashSet`'s insertion algorithm has a big disadvantage: with JVSTM's original implementation, inserting a box in the read set consists only in putting its reference at the end of the current array, and eventually creating a new array if the current one is full, whereas inserting a new item in an `IdentityHashSet` requires several more steps.

Initial testing showed that this solution performed better than the original under high-concurrency, but was slower in low-concurrency scenarios. The next section describes a solution to this problem and its results.

5.1.1. Variation with a Hybrid Structure for the Read Set

As seen in the previous experiment that always uses an `IdentityHashSet` as a structure for transactions' read sets, the balance between cost and benefit of the `IdentityHashSet` structure determines a positive or negative outcome depending on the circumstances. In this section I present the results of a variation of the JVSTM that attempts to combine the benefits of both the original array solution and the `IdentityHashSet`.

As we've seen previously, `IdentityHashSet` introduces a higher cost in the insertion operation than the simple array structure used before, but the benefits are only reaped if the transaction performs incremental validations against other transactions, which doesn't always happen. To try to avoid those costs in some situations, this adaptive solution uses the original array structure as the default for the read set during transaction execution, and postpones the creation of the `IdentityHashSet` until after snapshot validation. Thus, transactions that successfully enter the commit queue immediately after snapshot validation won't incur the extra costs of the `IdentityHashSet`. In turn, transactions that do end up creating and populating an `IdentityHashSet` will need more time and space than in the non-adaptive version because they will have populated the traditional read set before converting it into the new structure, but hopefully this extra cost will be offset by the benefits of using `IdentityHashSet`.

Threads	Running time (ms)		
	Original	IHS	Hybrid RS
1	91.358	99.949	88.580
2	47.247	55.079	44.856
4	25.848	31.437	24.319
8	15.326	17.020	14.235
16	14.118	10.320	9.858
32	13.569	7.917	8.143
48	13.142	7.951	8.127
64	13.097	8.339	8.207
96	13.373	8.652	8.331

Table 2: Performance comparison of the original version, `IdentityHashSet` and hybrid read set

Table 2 shows the running time for the original version of the JVSTM, the one that uses `IdentityHashSet` exclusively and the one with a hybrid read set for the tests

with 250.000 transactions, 1.000 operations per transaction and a 10% write ratio. Comparing the performance of the IHS version with the hybrid read set, we can see that tests up to 8 simultaneous threads are significantly faster ($11-19\%$), since in this situation most of the times it is not useful to use an `IdentityHashSet` for the read set (as the probability of doing incremental validation is low). Higher concurrency tests only show marginal variations ($\pm 5\%$) because in this case there is incremental validation most of the times. The number of aborted transactions has a behavior similar to the time results — low-concurrency tests perform similarly to how they did in the original version, whereas high-concurrency tests have values similar to the `IdentityHashSet` version. Comparing the hybrid read set with the original implementation, it matches JVSTM's performance under low concurrency (up to 8 threads) with slight performance gains, and performs clearly better than the original implementation with 16 or more threads.

The conclusion is that low-concurrency tests perform as well as in the original version, while high-concurrency tests perform similarly to the version with the `IdentityHashSet`. This means that, in effect, the hybrid read set succeeded in combining the strengths of both versions, as was expected.

5.2. Adaptive Validation

When a transaction T completes the execution phase, it must be validated before entering the Active Transactions Record (ATR). The validation process has two phases: snapshot and incremental. The first one is mandatory and consists in checking every box in T's read set for updates. If this succeeds, JVSTM attempts to put T in the ATR, which may trigger incremental validations against other transactions $U_{1..N}$. Validating T incrementally against a transaction U_x consists in looking for intersections between U_x's write set and T's read set. This allows the validation process to be lock-free and still guarantee a valid commit order for concurrent transactions. The rationale behind this two-phase process is that performing incremental validation should be cheaper than repeating snapshot validation.

Let's consider the scenario where a transaction tries to enter the ATR only 4 slots after it started. Let's also assume all transactions in this scenario read 900 boxes and attempt to write into 100. When the transaction tries to enter the ATR, there will be 3 records between its `activeTxRecord` and the ATR's end. In this scenario, JVSTM would perform a snapshot validation, which would guarantee the transaction is valid up to a certain point and then it would attempt to put it in the ATR, eventually triggering incremental validations. Under these circumstances, snapshot validating the transaction would require validating up to 900 boxes (its read set). However, we know upfront that it is valid at least up to and including the point where it started. This means that its snapshot validation could be substituted with 3 incremental validations against the transactions that finished after it started, which equates to checking only 300 boxes (remember each transaction wrote into 100 boxes).

With this in mind, I propose that JVSTM adopts an adaptive strategy to choose whether to use the current two-phase validation scheme (with both snapshot and incremental validation) or only incremental validation. I implemented this strategy by calculating the difference between the transaction numbers of records `lastSeenCommitted` and `activeTxRecord`. If this difference is below a certain threshold, only incremental validation is done starting at `activeTxRecord`, otherwise the two-phase process is performed.

Because incremental validation involves performing lookups in transactions' read sets, the initial implementation of this idea using the original read set implementation (an array) didn't perform well. Thus, I combined adaptive validation with the hybrid read set proposed earlier, as that structure is better suited for lookups.

This new variant with the hybrid read set has much improved results in the 10% write ratio tests with 8 or more simultaneous threads: with 8 threads there is an 9% improvement in running time, whereas the tests between 16 and 96 threads are $28-38\%$ faster. These results repeat the behavior seen in Section 5.1.1 with the adaptive read set: low concurrency tests are slower and high concurrency tests show considerable gains, whereas the number of aborted transactions also shows a comparable behavior. Thus, it's important to compare these results with those of the implementation with adaptive IHS and traditional validation, in order to assess the effectiveness of adaptive validation.

Threads	Adaptive IHS	Adapt. Valid.	Improv.
1	102.207	87.132	15%
2	50.189	61.139	−22%
4	30.297	29.032	4%
8	17.051	16.919	1%
16	11.237	11.528	−3%
32	10.049	10.085	0%
48	10.129	10.449	−3%
64	10.279	10.201	1%
96	10.401	10.568	−2%

Table 3: Total running time of adaptive IHS and adaptive IHS with adaptive validation (in milliseconds, 10% write ratio)

By observing Table 3, which shows the impact of adaptive validation in the total running time of the 10% write ratio scenario, it's clear that this technique has a positive impact when there is no concurrency and that it barely changes the results of tests with 4 simultaneous threads and above.

This result can be explained by how the commit queue, the Active Transactions Record, works, where transactions enter the commit queue when they're guaranteed to be valid. Naturally, as the number of transactions running concurrently increases, the higher will be the number of records entering the commit queue between a given transaction starting and entering the queue. Thus, the distance between the `activeTxRecord` and `lastSeenCommitted` records is less likely to fall be-

low the adaptive validation threshold as concurrency increases, while it will always fall below the threshold when there is no concurrency, which explains why the first test shows improvements and the high-concurrency ones barely show an impact.

From these results we can conclude that adaptive validation can only be considered if the read set structure is altered, and even then its only positive contribution is for cases where there is no concurrency — the implementation described herein didn't improve JVSTM's scalability in any way.

5.3. Collaborative Validation

As described previously, JVSTM employs a two-phase validation algorithm to guarantee a valid transaction commit order, where the first phase (snapshot validation) is mandatory and the second (incremental validation) is only necessary when two transactions try to enter the Active Transactions Record (ATR) simultaneously. These two phases are necessary because each transaction validates itself and then competes with others for a place in the ATR, which defines the commit order. Losing transactions then have to revalidate themselves against the one that entered the ATR and try again. The higher the number of transactions running simultaneously, the greater are the chances that two or more will be competing for a place in the ATR at the same time, and thus more time will be spent on incremental validations. Currently, a transaction's lifecycle in JVSTM can be described as follows:

1. Execution
2. Help all transactions already in the ATR commit (collaborative commit)
3. Snapshot validation
4. Repeat incremental validation until successfully entering the ATR
5. Help write back/commit all transactions in the ATR up to, and including, itself.

With collaborative validation, I suggest that the snapshot validation phase is moved forward in the transaction's lifecycle in such a way that incremental validations are rendered unnecessary without compromising the validity of the commit order, hopefully reducing the time it takes to validate a transaction under high concurrency. Also, instead of each transaction validating itself, all transactions that are not in the execution phase help the oldest uncommitted transaction validate using the snapshot algorithm.

To accomplish this, transactions are allowed to enter the ATR before being validated. The competition for a place in the ATR still exists (as previously, a CAS is used), but upon failure the transaction merely has to repeat the CAS and doesn't perform any kind of validation. Because of this change, the ATR now contains records of transactions that will later fail validation and abort, but it still represents the order in which valid transactions will be committed. Now, instead of each transaction validating itself, all helper transactions take part in the validation process, and incremental validation is no longer necessary.

Compared to the previous usage of the ATR, this one contains records marked as "pending validation" that are already inside the ATR and have also already been assigned a definite transaction number, but it will only be used if these transactions end up being valid.

Because this algorithm relies solely on snapshot validation, it can potentially reduce the overall cost of validating a transaction because the incremental process is no longer required.

The experiments made showed that the idea of collaboratively validating transactions does not scale well. The reason for this is that where threads were previously spending their time validating themselves (which is guaranteed to be necessary work), with collaborative validation and under high concurrency there are too many threads trying to validate the same transaction, which results in unnecessary work being done, and thus higher execution times.
On the other hand, under low concurrency this solution at best matches the original implementation.

5.4. Simultaneous Write-back

As the name suggests, the idea of simultaneous write-back is to allow more than one transaction to be written back simultaneously, with the goal of reducing the potential for unnecessary work. However, this must be done keeping in mind that serialized write-back was necessary to guarantee that values were committed to the boxes in the correct order of versions.
To comply with this requirement, the algorithm must check for write/write conflicts between transactions, which occur when two valid transactions' write sets have at least one box in common. This means that the older transaction must be written back before the newer one can be committed (serial write-back). Otherwise, the order in which the transactions are committed is irrelevant, since they won't be committing to the same boxes.

Let's consider a scenario where T_7 was already committed, T_8 was already validated and is waiting in queue to be written back, and transactions $T_{X..Y}$ are being validated and have yet to enter the queue. T_8's write set contains boxes A and B, T_X's contains box A and T_Y wants to commit a value to box C. Let's assume there aren't any read/write conflicts between any of the transactions, meaning that both T_X and T_Y will eventually enter the queue.

To detect write/write conflicts, we'll leverage the `IdentityHashSet` described in an earlier optimization to also keep track of boxes written. When T_X wants to enter the queue, it will have to perform incremental validation against T_8 and at that point it will detect that box A is in T_8's `IdentityHashSet`, and thus will add that transaction's record to its list of write conflicts. T_X will then enter the queue as T_9, T_Y will validate itself against $T_{8..9}$ and enter the queue as T_{10}. This reduces the overhead of this technique.

Under the original algorithm, all three transactions would be committed sequentially, one at a time, with the newer ones helping the older one. However, note that T_{10} doesn't have any write conflicts, and thus can

commit itself without waiting for older transactions (i.e. $T_{8..9}$). With the simultaneous write-back algorithm, T_8 and T_{11} will start writing themselves back as soon as they enter the queue. A downside of this algorithm is that the write set of a committed transaction is only visible once all previous transactions in the commit queue have committed, otherwise new transactions could observe an inconsistent view of the system.
T_9, which can only be written after T_8 has finished, can then do one of the following: wait until its conflicts have been resolved ("wait" strategy), help any other transaction that has no outstanding write conflicts, or help the transactions with which it has a conflict (in this case, T_8) — "resolve" strategy.

The idea for simultaneous write-back comes from a scenario where transactions are persisted not only to memory, but also to disk. In this scenario, the cost of writing to disk is high, and thus the possibility of having several transactions writing simultaneously can lead to performance gains. Transactional systems usually guarantee persistence to disk, and in this case this is simulated writing only in one file.

However, due to JVSTM's help mechanism, where the same transaction can be written back by different threads, writing to disk would result in one the following problems: when writing back a transaction's block, an helper thread would have to wait for the file to be closed by another thread that's also writing back the same transaction, or a transaction's blocks would be scattered across different files.

To avoid those problems, a transaction's write set must be written to disk in full by the same thread, which means JVSTM's help feature can't be used. Without further modifications, threads that would otherwise help a transaction commit must instead wait in idle until their own transaction can be committed. Obviously, this leads to long execution times.

In this context, simultaneous write-back emerges as an alternative to the write-back help algorithm. Although threads don't help each other commit, multiple threads can commit their own transactions simultaneously, and thus performance is much better.

However, initial tests showed that there is a significant increase both in time and aborted transactions when the number of concurrent threads is doubled (in both versions). A possible reason for this in the simultaneous write-back version is that the disk becomes a bottleneck due to the fact that too many threads are writing to disk simultaneously, but it isn't the only reason because the same behavior happens in the version without help (where only one thread writes to disk at any given time).

I concluded that another problem was the order of the commit procedure, which initially was: write back to disk, write back to memory, mark the transaction as committed. Because writing to disk takes time, transactions that started during this procedure and read boxes written by the committing transaction would eventually abort when it was marked as committed. In order for these transactions to observe a more recent view of the system, I changed the write-back protocol to: write back to memory, mark the transaction as committed, write back to disk, thus making the transaction's effects visible to others earlier.

Table 4 presents the results of this change, which yielded better results. Note that when using this new order in a production environment, any failures that occur when writing to disk could lead to a rollback of the values written to memory, which would be problematic given that new transactions could already be working with those values.

	Threads	First to Disk	First to Memory
Time	48	$38,2\ s$	$28,3\ s$
Aborts		1.327.112	628.516
Time	96	$84\ s$	$41,2\ s$
Aborts		11.491.939	1.897.284

Table 4: Comparison of the two different commit orders using simultaneous write-back

5.5. Tweak the Write-back Help Algorithm

With the goal of reducing the amount of duplicate work when transactions help each other write-back, I experimented with different tweaks to the help algorithm. These tweaks showed only modest improvements in some scenarios, thus I won't present them here for brevity.

5.6. Upgrade Transactions

In the JVSTM, when a transaction attempts to read a box whose version number is greater than the transaction's own version number, it is aborted because this means that there is a conflict with another transaction that committed a new value to that box after the transaction started. The consequence of this is that the work the transaction performed before aborting is thrown away and it will have to restart from the beginning with a new version number.
A possible solution to avoid this waste of work when this occurs is to try to *upgrade* the transaction to a new version number first, instead of immediately aborting it. The rationale behind this idea is that if the transaction was running with a version number higher than the box's, there wouldn't be a conflict. If the upgrade succeeds, the transaction can continue normally and its prior work won't be wasted.

Naturally, we must guarantee that the transaction remains valid after it is upgraded. A transaction's ability to be upgraded depends on whether the work it has performed until the upgrade is still valid under the transaction's new version number. In other words, we must guarantee that the transaction's current state (read and write sets) would be the same if the transaction had run with the new version number.
Since the transaction's write set is only effectively written after the execution phase, changing the transaction's version number during its execution doesn't have a direct effect in the transaction's validity, and thus the write set can always be kept intact during an upgrade.
On the other hand, the read set is dependent on the

transaction's version number. When we upgrade a transaction from version X to version Z (where $X < Z$), it's as if it started later than it actually did, and thus its current read set is only valid if none of the boxes it contains was changed between t_X and t_Y. In practice, this means that to upgrade a transaction we must run the snapshot validation algorithm on its current read set.

It follows that when a transaction T_X attempts to read from a box B whose most recent version Y is greater than X, instead of aborting right away, JVSTM does the following:

1. A local reference to the current `mostRecentCommittedRecord`, whose version is Z ($X < Y \leq Z$), is recorded.
2. Partial snapshot validation: if any of the boxes in T's read set (which at this point doesn't contain the conflicting box B yet) has a version number higher than X, the transaction can't be upgraded and must be aborted.
3. If all the boxes T already read remain valid, it can be upgraded to version Z.
4. The transaction can now continue reading B.

Note that the first step is needed because the `mostRecentCommittedRecord` can change if another transaction commits while we're verifying T. If that transaction happens to change a box we've already verified, without the local reference we'd be mistakenly upgrading T to a version where its read set isn't really valid. By having a local reference that doesn't change during verification, if this scenario arises it will be detected at the latest when the transaction is validated.

Naturally, this upgrade process will only be beneficial if the time it saves (namely the time the transaction already spent executing until a conflict arises) is greater than the time it takes to verify the read set's validity. From this follows that the longer the transaction has already run, the more time we can potentially save.
To test this theory, the following tests include different degrees of artificially inflated execution times, which are achieved by putting the thread to sleep every time the transaction writes a value. In practice, this simulates the execution of non-transactional operations, which a realistic application is likely to include, but the benchmark in use didn't support (all operations in a transaction involved reading or writing shared boxes).

	Time+	No Upgrade	Upgrade	Improv.
Time	None	$21,9\ s$	$21,2\ s$	3%
Aborts		789.231	546.220	31%
Time	1 ms	1.828 s	1.378 s	25%
Aborts		672.272	349.370	48%
Time	2 ms	3.395 s	2.568 s	24%
Aborts		670.554	350.026	48%

Table 5: Comparison of the original JVSTM with and without upgrade (48 threads)

Tables 5 and 6 compare JVSTM's performance in its original implementation (i.e. with no other optimization)

	Time+	No Upgrade	Upgrade	Improv.
Time	None	$24,1\ s$	$21,5\ s$	11%
Aborts		821.620	591.611	28%
Time	1 ms	1.395 s	998 s	28%
Aborts		1.223.685	631.715	48%
Time	2 ms	2.564 s	1.824 s	29%
Aborts		1.222.390	629.454	49%

Table 6: Comparison of the original JVSTM with and without upgrade (96 threads)

with and without the upgrade feature. The "Time+" inflation column refers to the time the thread is put to sleep every time its transaction makes a write operation to simulate non-transactional operations.
When transactions only operate on boxes (i.e. when we don't simulate other operations by putting the thread to sleep), upgrading transactions instead of aborting them results in a reduction of around 30% of the number of aborted transactions in both scenarios (48 and 96 threads). This only leads to negligible/modest gains in terms of execution time (3 and 11%) because the cost of upgrading transactions is only barely offset by the time this feature saves, especially with 48 threads. However, this allows us to conclude that even in a scenario where transactions only perform operations on shared boxes, the upgrade algorithm doesn't have a negative impact.
When we factor in additional, non-transactional operations, by simulating them with 1 and 2 ms sleep times, aborts are nearly halved and, more importantly, execution is at least 24% faster because this scenario has a better balance between upgrade cost and benefit.

6. Conclusions and Future Work

6.1. Conclusions

In Section 3 I started by studying a set of existing STMs to understand how they work, what problems their developers are faced with and the different approaches they have at their disposal to solve them. Not having any prior experience with STMs, the goal of this analysis was to introduce me to the key concepts of STMs.

Then, in Section 4 I studied JVSTM's implementation in depth, which allowed me to understand how it works and thus identify the following potential areas for improvement, which were then addressed in Section 5:

1. Linear read set data structure.
2. Fixed validation process.
3. The validation process is not collaborative.
4. Duplicate work in the write-back help algorithm.
5. Conflicts always cause transactions to be aborted.

The first problem was addressed by substituting the array used for transactions' read sets with a non-linear data structure (the Identity Hash Set). While this solution proved effective for high concurrency scenarios, it didn't perform as well as the original data structure in low concurrency scenarios. I then proposed a hybrid implementation that uses each structure at different times

(the original array during execution and snapshot validation and the Identity Hash Set in incremental validation), that matches JVSTM's prior results under low concurrency and performs better with hight concurrency.
Another interesting conclusion of these experiments was that changing only one aspect of the STM (in this case the read set structure) can have interesting side effects. While the goal of changing the read set structure was only to save time in incremental validation, it also lead to a completely different abort distribution, with more transactions aborting earlier in the lifecycle.

The second problem was approached with a dynamic solution that only performs incremental validation under certain circumstances and reverts to the original two-phase process in other cases. Because this solution relies more on incremental validation, it performed poorly when implemented on the original JVSTM because it is incompatible with its linear read set implementation, which lead me to combine it with the first optimization that uses an Identity Hash Set. Unfortunately, the best results for this idea were observed in low concurrency scenarios, and thus it doesn't contribute to improve JVSTM's scalability.

The third idea of applying the same collaborative principle of the write-back phase to the validation phase also didn't prove effective as it ends up creating new opportunities of the same work being done by two threads.

Also in the domain of duplicate work, I discussed different methods of reducing duplicate work in the write-back help algorithm. The most relevant contribution in this domain is the simultaneous write-back algorithm, which also addresses the problem of persisting transaction data to disk.

Finally, the last problem was addressed by upgrading transactions to a new version number when certain criteria are met, effectively "saving" some transactions and reducing wasted time, which yielded positive results.

6.2. Future Work

Even though the benchmark used throughout this work is enough to compare the performance of different structures and algorithms and draw conclusions on their effect in JVSTM's behavior, it does not represent a realistic usage scenario due to several limitations. Even though I have tried to address some of those issues in some tests, the ideas discussed in this document could have different results if they were to be tested using different benchmarks. Likewise, different testing scenarios could reveal potential optimizations that this work didn't address.

In some experiments, I simulated persistence by writing to disk. Further work could improve this by using a database and experiment with different strategies to write data to the database.

Additionally, some of the algorithms presented in this work can be developed further to better perform in different environments.
As implemented, the helper limitation algorithms assume hard coded thresholds that may not be suitable for all workloads. Future developments of such algorithms could include methods to dynamically set thresholds based on the current environment's characteristics.

In several test scenarios, it often is the case that increasing the number of threads doesn't result in any time savings or actually leads to worse performance because under high concurrency the probability of aborts occurring increases. With this in mind, further experiments where not all running threads are executing transactions could be done. For example, instead of having 96 threads running 96 transactions, JVSTM could have the same 96 threads, but only a portion of them would have an associated transaction, and the remaining threads would only be helpers, contributing exclusively to help the existing transactions.
With this feature, there would be less transactions running simultaneously at any given time, thus leading to less conflicts, and the remaining processing power would help them finish faster. The success of this idea depends on the existence of a good balance between not executing too many transactions at once and not having too many threads helping.

References

[1] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[2] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[3] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing, Special Issue*, 10:99–116, 1997.

[4] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] J. Eliot B. Moss. Open nested transactions: Semantics and support. 2006.

[6] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[7] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[8] João Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, 2007.

[9] Damien Imbs and Michel Raynal. A Lock-based Protocol for Software Transactional Memory. Research report, 2008.

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529. IEEE Computer Society, 2003.

[11] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63:172–185, December 2006.

[12] Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM.