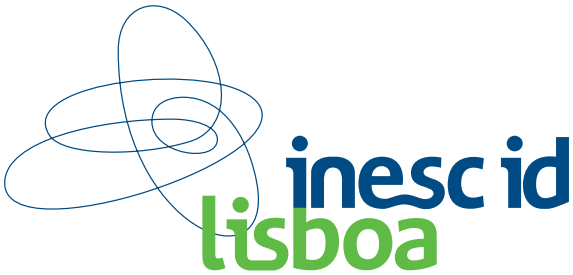

Technical Report RT/39/2011

Reducing conflicts on JVSTM transactions - STMBench7: A case study

Luís Pina
INESC-ID/IST
luis.pina@ist.utl.pt

João Cachopo
INESC-ID/IST
joao.cachopo@ist.utl.pt

Aug 2011

Abstract

Over the last years, multicores have become accessible to the common developer. Writing concurrent programs that are correct and that display good performance is hard. Software Transactional Memory (STM) is a step in the direction of solving the first problem, but it does not provide tools for the programmer to understand and optimize his code's performance, thus leaving the second problem as an open issue.

In this paper, we present a novel technique that informs the developer about which objects cause JVSTM transactions to conflict. Then, we describe how we used that technique together with several JVSTM conflict reduction techniques to improve the performance of a transactional application.

Keywords: Software Transactional Memory, Performance, Conflicts, Profiling.

Reducing conflicts on JVSTM transactions STMBench7: A case study

Luís Pina and João Cachopo

Instituto Superior Técnico /INESC-ID
{luis.pina,joao.cachopo}@ist.utl.pt

Abstract. Over the last years, multicores have become accessible to the common developer. Writing concurrent programs that are correct and that display good performance is hard. Software Transactional Memory (STM) is a step in the direction of solving the first problem, but it does not provide tools for the programmer to understand and optimize his code's performance, thus leaving the second problem as an open issue. In this paper, we present a novel technique that informs the developer about which objects cause JVSTM transactions to conflict. Then, we describe how we used that technique together with several JVSTM conflict reduction techniques to improve the performance of a transactional application.

Keywords: Software Transactional Memory, performance, conflicts, profiling

1 Introduction

Software Transactional Memory (STM) simplifies the task of writing concurrent programs. With them, developers group actions with *memory transactions* that execute atomically. Transactions may *commit*, making their whole set of changes atomically visible to all other transactions, or *abort*, discarding the changes and appearing as the transaction never took place.

The STM mechanism is responsible for serializing transactions, making an opaque history of committed transactions [4]. To do so, current STMs use read-write conflict detection on transactional memory locations: A transaction T can commit only if no other transaction has wrote to a transactional location that T has read since T started. The STM ensures that only transactions that have not experienced any conflict can commit. If the STM detects such conflict, it aborts and restarts T.

JVSTM [1] is an STM that uses multi-versioned transactional memory locations, called *VBoxes*. Each time a transaction successfully commits, it appends each new value that it wrote to the version list of each written VBox. JVSTM is optimized for read dominated workloads. For instance, as a direct result of versioning, read-only transactions never conflict with any other concurrent transaction and can thus always commit.

Despite the simpler programming model, how do STMs' performance scales with higher levels of concurrency? To help providing insight about this question, Guerraoui et. al. developed STMBench7 [5]: A benchmark for evaluating STM implementations. STMBench7 performs several types of different operations over a rich object graph to mimic a real world application that displays non-trivial concurrency.

To adapt STMBench7 to evaluate a new STM, developers just need to implement STMBench7's *backend* and *core* objects using the new STM. Backend objects are structures that every program uses to keep its data, such as sets, bags, and indexes. Core objects are STMBench7 specific objects that hold collections of references to each other. STMBench7 uses core objects to create its rich object graph. The bulk of the benchmark remains unchanged for every STM. Implementing these objects is much simpler than implementing the whole benchmark.

Figure 1 shows the results that STMBench7 produces for JVSTM with a direct implementation of the backend and core objects. By direct, we mean an implementation as a common developer unaware of STM internals would write. The results show that the JVSTM implementation scales up to 8 threads for the read-dominated workload, and up to 4 threads for the read-write and write-dominated workloads.

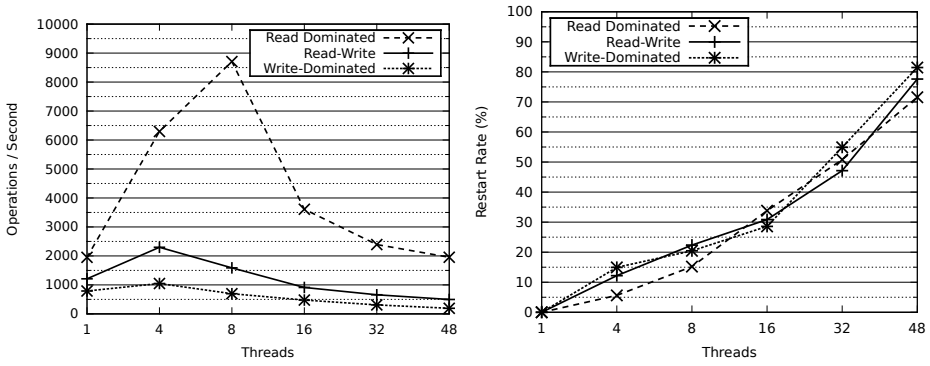

Fig. 1. Throughput and restart rate obtained when benchmarking JVSTM with STMBench7 using a direct implementation of STMBench7's backend and core objects. We describe the experimental methodology we used to obtain these results in Section 3.1.

In this paper, we analyse the conflicts that prevent JVSTM from scaling, find the underlying conflicting objects and replace them by a concurrent-friendly and semantically equivalent version. We discuss in detail which techniques we used to create the concurrent-friendly objects, which range from general techniques used to increase concurrency to JVSTM specific techniques, such as *per transaction boxes* and *restartable transactions*. Although JVSTM specific, we believe that such techniques can be easily adapted to other STMs.

In summary, this paper's contributions are:

- An empirical evaluation of per transaction boxes and restartable transactions;
- A technique to detect conflicting objects that cause JVSTM transactions to abort;
- A set of techniques to implement concurrent-friendly objects for STMs in general, and JVSTM in particular.

The rest of this paper is structured as follows: We start by describing a conflict detection technique in Section 2. Then, in Section 3, we use that technique to find and optimize the objects that contain the VBoxes that experience the largest number of conflicts. We discuss the obtained results in Section 4. In Section 5, we compare the conflict reduction techniques that we present in Section 3 with the existing state-of-the-art conflict reduction techniques that other transactional systems use. Finally, we conclude in Section 6.

2 Detecting Conflicts

When optimizing an application for performance, developers use profiling techniques to help them to find the bottleneck. There are a large set of well known and robust techniques to profile sequential applications, such as the time spent inside each method and the number of times that each method is called. If a concurrent applications is lock-based, we can monitor how threads acquire, release, and wait for locks. These techniques, however, fail to give detailed information about bottlenecks in concurrent applications in general.

Using typical profiling techniques with transactional applications may even lead to misinformation about the bottleneck. For instance, consider a transactional binary tree with a large number of elements and a counter that keeps the number of elements currently on the tree. Consider also a set of transactions that add elements in such way that they change only the leaf nodes, besides the counter. It is clear that such transactions experience a high conflict rate due to updating the counter. However, the profiling information may mislead us, showing that the transactions spend most of the time navigating through the tree.

To properly identify which objects are responsible for creating conflicts, we developed a conflict detection technique. We enriched each transactional location (VBox) with information about which object holds the single reference to it, and in which field does it keep such reference. We also inject code on every object's constructor to properly initialize this information.

At commit time, JVSTM validates each VBox present in the transaction's read-set. We modified how JVSTM performs this task: Instead of aborting the transaction as soon as it detects the first conflict, it validates the whole read-set. When JVSTM detects each conflict, it registers that the owner of the conflicting VBox was involved in one more conflict. Finally, when the application exits, JVSTM prints a list of which VBoxes were involved in conflicts, in how many

conflicts were each VBox involved, and in which field does each owner object stores the conflicting VBox reference.

Figure 2 shows the results that we obtained using this technique for STMBench7. We can see that there are two fields of class `AtomicPartImpl`, `x` and `y`, responsible for most of the restarts.

	4	8	16	32	48
AtomicPartImpl::y	1.726.871	521.417	1.457.949	18.477.298	64.052.709
AtomicPartImpl::x	1.726.871	521.417	1.457.949	18.477.298	64.052.709
LargeSetImpl::elements	213.660	706.930	961.391	882.978	93.991
LargeSetImpl::count	212.310	706.587	960.635	870.216	47.154
ManualImpl::text	47.799	15.765	12.144	13.838	16.850

Fig. 2. Number of conflicts that the top 5 most conflicting VBoxes experience when using STMBench7 with the direct implementation of its backend and core objects. We obtained these results using the write-dominated workload. We describe further details about the experimental methodology in Section 3.1. The first column shows the field that keep the conflicting VBox. Rows are sorted by the descending values of the rightmost column.

3 Reducing Conflicts

In this section, we analyse each source of conflicts that we find on STMBench7 using the technique that we describe in Section 2. Then, we: (1) Describe the measures we took to reduce the conflicts that that source generates, (2) report the new throughput, restart rate, and conflicts after introducing the optimization, and (3) repeat the process until the restart rate drops to an acceptable point, thus leading to a better throughput scalability.

3.1 Experimental Methodology

We obtained the results that we present along this paper using a machine equipped with 4 AMD Opteron 6168 chips (12 cores per chip, 48 cores total) with 128GB of RAM. We used Java(TM) SE Runtime Environment (build 1.6.0_24-b07) with Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02, mixed mode).

Along this paper, each value that we report (either a point on a chart or a cell on a table) represents the average of 10 STMBench7 executions, which take place in sequence.

3.2 Atomic Part

Figure 2 shows the results that we obtained using the conflict detection technique on the direct implementation of STMBench7. We can see that there are two

fields of class `AtomicPartImpl`, `x` and `y`, responsible for most of the restarts. After looking at the code, we noticed that there is only one method that writes these fields: Method `AtomicPartImpl::swap` swaps the values on `x` and `y`.

There are two types of conflicts involved here: (1) Between two transactions that both swap `x` and `y`, thus writing them both, and (2) between a transaction that reads `x` or `y` and another that swaps them.

The invariant that involves fields `x` and `y` is as follows: The absolute value of their difference must always be equal to one. Therefore, conflicts of type 1 are benign: A transaction T that wants to swap `x` and `y` is not interested on the values that these fields hold. However, to swap them, T must read both fields, thus adding them to its read-set and generating conflicts at commit time.

To reduce conflicts of type 1, we used a *per transaction box*. Per transaction boxes, as their name indicates, hold a different value for each transaction. When using them, the developer must specify their initial value and must override a method that shall be executed at commit time, after validating the transaction and while holding the global commit lock. Per transaction boxes are useful to avoid a conflicting read during the transaction, registering instead the intention and performing that read at commit time, when it does not generate any conflict.

To reduce conflicts of type 1, we added a per transaction box `AtomicPartImpl::swapped` that holds a boolean value, initially `FALSE`. Method `swap` negates the value that the per transaction box holds. At commit time, if the per transaction box contains `TRUE`, it swaps the values of boxes `x` and `y`.

There are two methods that consult the value of boxes `x` and `y` called `getX` and `getY`, respectively. We modified these methods to consult the value of the per transaction box and return the appropriate value. This means that an operation that reads box `x`, if box `swapped` is TRUE, ends up adding to the read-set box `y`. However, keep in mind that boxes `x` and `y` are modified only by swapping them, thus writing both. Therefore, this modification of methods `getX` and `getY` does not break the original implementation semantics.

Please note that transactions involved in both types of conflicts are read-write. In JVSTM, each time a transaction successfully commits, it appends each new value that it writes to the version list of each written VBox. As a direct result of versioning, read-only transactions never conflict with any other concurrent transaction and can thus always commit.

Figure 3 shows the impact that adding the per transaction box deals on the throughput and restart rate. Although this modification seems to yield an insignificant throughput increase, it was able to visibly decrease the restart rate measured on higher levels of concurrency.

Running the conflict detection technique again, whose results are shown in Figure 4, shows that this modification was able to effectively decrease the number of conflicts that fields `AtomicPartImpl::x` and `AtomicPartImpl::y` cause.

3.3 Large Set

After dealing with boxes `AtomicPartImpl::x` and `AtomicPartImpl::y`, Figure 4 shows us that class `LargeSetImpl` has now the boxes involved in most con-

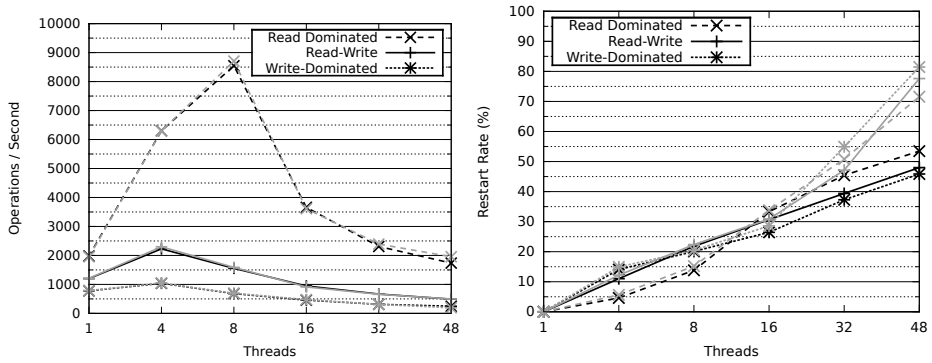

Fig. 3. Throughput and restart rate obtained when benchmarking JVSTM with STMBench7. The chart shows the results after optimizing the Atomic Part implementation, as well as the previous results (Figure 1) in grey. We describe the experimental methodology we used to obtain these results in Section 3.1.

	4	8	16	32	48
LargeSetImpl::elements	205.812	700.881	984.145	1.198.163	1.256.247
LargeSetImpl::count	205.057	700.722	984.102	1.198.098	1.256.140
VIndex::index	18.152	46.291	58.713	68.969	71.523
VQueue::front	26.849	31.269	39.754	50.541	48.997
ManualImpl::text	48.103	15.385	7.145	9.139	11.245
AtomicPartImplWithPerTxBoxes::y	9	23	83	185	249
AtomicPartImplWithPerTxBoxes::x	9	23	83	185	249

Fig. 4. Number of conflicts that the top 5 most conflicting VBoxes experience when using STMBench7 after optimizing the Atomic Part implementation. We obtained these results using the write-dominated workload. We describe further details about the experimental methodology in Section 3.1.

flicts. This class is a set of objects, implemented as a functional Red-Black tree (RBT). VBox `LargeSetImpl::elements` holds the functional RBT and VBox `LargeSetImpl::count` holds the number of elements on the tree.

The class `LargeSetImpl` supports 4 operations: (1) add an object to the set, (2) remove an object from the set, (3) check if the set contains an object, and (4) iterate over the objects that the set contains. Taking a look at how STMBench7 uses this instances, we notice that transactions that write the set (by using operations 1 or 2) do not use operations 3 and 4. Moreover, each transaction that modifies the set does so using only one type of operation, either 1 or 2.

With this knowledge, we added two per transaction boxes to the class `LargeSetImpl`: (1) box `LargeSetImpl::added` to hold the values that the current transaction added to the set, and (2) box `LargeSetImpl::removed` to hold the values that the current transaction removed from the set. Then, we modified the add and remove operations to add the object to the respective per transaction box. At commit time, each per transaction box performs the add or remove opera-

tion on VBoxes `elements` and `count`. We also modified the other operations on `LargeSetImpl` to remain correct on the presence of the new per transaction boxes.

	4	8	16	32	48
ManualImpl::text	81.905	183.317	230.422	186.706	345.556
VIndex::index	8.947	11.021	11.604	34.967	20.342
VQueue::front	17.103	18.019	14.612	31.713	21.623
BuildDate::buildDate	164	517	1.332	4.806	9.744
VQueue::rear	740	1.050	1.265	3.338	1.513
LargeSetImpl::elements	0	0	0	0	0
LargeSetImpl::count	0	0	0	0	0

Fig. 5. Number of conflicts that the top 5 most conflicting VBoxes experience when using STMBench7 after optimizing the Atomic Part and Large Set implementation. We obtained these results using the write-dominated workload. We describe further details about the experimental methodology in Section 3.1.

Running the conflict detection technique with the modified Large Set shows, in Figure 5, that this modification eliminates all conflicts involving boxes on class `LargeSetImpl`. To understand why, keep in mind that read-only transactions never generate any conflict and always commit in JVSTM. Thus, we can conclude that transactions that invoke method `LargeSetImpl::contains` or iterate over the set are read-only. This means that all the conflicts were generated by transactions that added or removed objects from the set. The technique of adding two per transaction boxes `added` and `removed` effectively eliminates all of such benign conflicts.

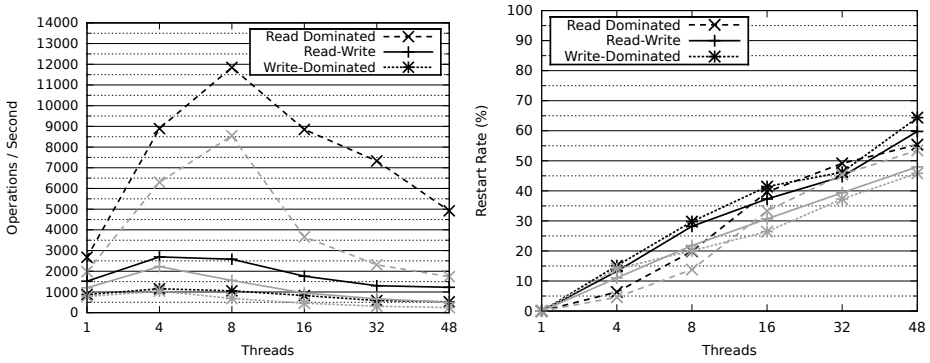

Fig. 6. Throughput and restart rate obtained when benchmarking JVSTM with STMBench7. The chart shows the results after optimizing the Atomic Part and Large Set implementation, as well as the previous results (Figure 3) in grey. We describe the experimental methodology we used to obtain these results in Section 3.1.

Adding the per transaction boxes to class `LargeSetImpl` has a positive impact on throughput, as Figure 6 shows. The modified version displays a significant increase on throughput, outperforming the previous version on all combinations of workloads and threads. The restart rate, on the other hand, experiences a small increase.

3.4 Manual

So far, we have made two classes concurrent-friendly to JVSTM: `AtomicPart`, at Section 3.2, and `LargeSet`, at Section 3.3. Now, Figure 5 shows us that class `ManualImpl` owns the VBox involved on most conflicts: VBox `ManualImpl::text`.

Class `ManualImpl` exports four read methods that operate on a single and large string: (1) `countOccurences`, (2) `checkFirstLastCharTheSame`, (3) `startsWith`, and (4) `getText`. It also exports one write method: `replaceChar`, which replaces all occurences of the given character on the string that VBox `text` keeps by another character, returning how many characters it replaced. It is also worth mentioning here that there is only one instance of class `ManualImpl` for the whole benchmark.

The read methods do not generate any conflicts because they are invoked only during read-only transactions. Which means that the conflicts that we measured for VBox `ManualImpl::text` come from transactions that are concurrently executing method `replaceChar`.

Per transaction boxes do not help us on reducing these conflicts because method `replaceChar` must return how many characters it replaced. It is possible to use them to postpone the replacing operation to commit time. However, we must compute how many characters shall be replaced to return the correct value. This ends up adding the conflicting VBox to the read-set, which is exactly what we are trying to avoid.

Following the trail of the value that method `ManualImpl::replaceChar` returns, we realized that it is used only after the transaction ends. With this knowledge, we implemented a new mechanism for implementing concurrent-friendly objects on JVSTM: *Transactional futures*. A transactional future, like a regular future, represent the promise of a result. Also, as regular futures, transactions are free to delay the computation of a transactional future until it is used for the first time. However, if this first time occurs after the transaction has finished, transactions can compute the value of the future inside the commit lock.

Using transactional futures to reduce the conflicts on VBox `ManualImpl::text` is quite simple: Operation `replaceChar` returns a transactional future. Figure 7 shows that this optimization eliminates all conflicts. This happens because the transactional future is used after the transaction finishes, which means that it effectively moves the computation of the operation to the commit phase, thus not adding the conflicting VBox to the read-set of the transaction and eliminating the conflicts.

Figure 8 shows the result of such optimization in terms of throughput and restart rate. Despite eliminating a major source of conflicts, the throughput drops considerably on all workloads. This happens because method `ManualImpl::replaceChar`, which we moved to the commit phase, is computationally intensive. In JVSTM,

	4	8	16	32	48
VIndex::index	17.500	46.624	82.784	76.904	77.335
VQueue::front	18.008	44.523	69.518	59.029	57.577
BuildDate::buildDate	486	2.131	6.238	12.508	18.429
VQueue::rear	933	2.119	4.502	5.587	4.523
VLinkedSet::entries	1.882	5.220	5.215	1.903	1.505
ManualImpl::text	0	0	0	0	0

Fig. 7. Number of conflicts that the top 5 most conflicting VBoxes experience when using STMBench7 after optimizing the Atomic Part, Large Set, and ManualImpl implementation. We obtained these results using the write-dominated workload. We describe further details about the experimental methodology in Section 3.1.

at most one transaction may be at the commit phase at each time. Thus, entering the commit phase is now a bottleneck for all transactions, slowing down the whole benchmark.

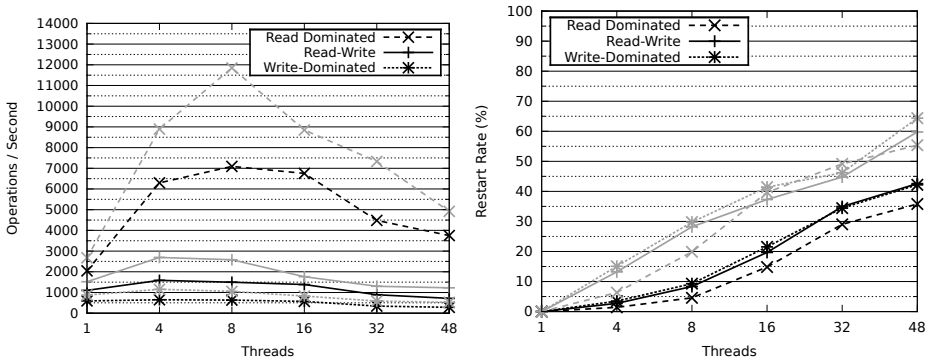

Fig. 8. Throughput and restart rate obtained when benchmarking JVSTM with STMBench7. The chart shows the results after optimizing the Atomic Part, Large Set, and Manual implementation, as well as the previous results (Figure 6) in grey. We describe the experimental methodology we used to obtain these results in Section 3.1.

To mitigate this problem, we implemented a concurrent version of method `ManualImpl::replaceChar` that breaks the string on several chunks, assigns a task to replace the characters on each chunk, and uses a thread pool to execute such tasks. When a thread attempts to grab the commit lock and fails, the JVM blocks it until the lock is released. This frees one processor to work on another thread, namely a thread that belongs to the thread pool. The throughput improvement over Figure 8 that Figure 9 shows means that the JVM is smart enough to assign that free processor to a thread that belongs to the thread pool.

This optimization does not improve the throughput, as Figure 9 shows. But it does not harm the throughput significantly either. And, more importantly, it

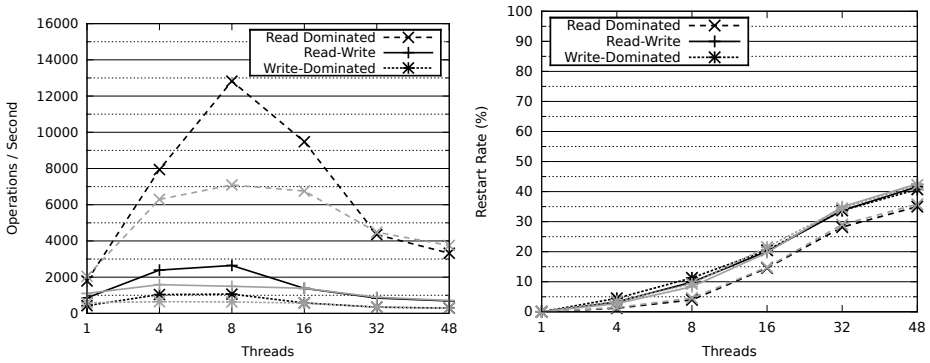

Fig. 9. Throughput and restart rate obtained when benchmarking JVSTM with STMBench7. The chart shows the results after optimizing the Atomic Part, Large Set, and Manual (with the concurrent `replaceChar` operation) implementation, as well as the previous results (Figure 6) in grey. We describe the experimental methodology we used to obtain these results in Section 3.1.

decreases the restart rate, thus making room for further optimizations to improve the scalability.

3.5 VIndex

Up to this point, we presented 3 optimizations to JVSTM's implementation of STMBench7's core and backend objects: (1) Atomic Part, (2) Large Set, and (3) Manual. Now, the VBox that participates in the largest number of conflicts is `VIndex::index`, as Figure 7 shows. Class `VIndex` is an index that maps keys (integer based) to objects (core objects). It is implemented using a functional Red-Black Tree (RBT), which is held by VBox `VIndex:index`.

Class `VIndex` supports the typical set of index operations: (1) add a key-object mapping, (2) remove a mapping given its key, and (3) get the object associated with a given key. Almost every STMBench7 transaction starts by generating a random key and getting the associated object on a given index. Then, it traverses a portion of the object graph starting on that object. The few transactions that do not use indexes start operating on the known root parts of the object graph.

The transactions that modify STMBench7's indexes behave much like the transactions that modify large sets in the sense that transactions that add elements to the index do not remove other elements from that same index, and transactions that remove elements do not add any element. To decrease the number of conflicts between transactions that modify indexes, we added two per transaction boxes `added` and `removed` to the index implementation, just as we did for class `LargeSet` (described in detail in Section 3.3).

The per transaction boxes alone do not solve the problem because any transaction T_1, which may not be read-only, that consults an index without modifying it conflicts with any other transaction T_2 that modifies that index. Most of the

times this conflict is benign because transaction T_2 does not modify the mapping that transaction T_1 consulted on the index.

We used *restartable transactions* to decrease the conflicts between transactions that read an index and transactions that modify that index. Restartable transactions, which were previously described [1] but never implemented, wrap a read-only method execution, keep a separate read-set, and save the value that the method returns. When validating a transaction, JVSTM validates each restartable transaction that was executed during the transaction separately and after validating the parent transaction's read-set. If any restartable transaction fails to validate its read-set, JVSTM re-executes it inside the commit lock and against the most recent version of the world. If the result of the re-execution is the same as the original result, the restartable transaction is still valid.

To decrease the conflicts involving transactions that read the index, we wrapped the get operation inside a restartable transaction. This modification eliminates the benign conflicts that afflicts such operation.

Figure 10 shows that the optimizations that we describe in this Section effectively decrease the number of conflicts in which instances of VBox `Vindex::index` are involved.

	4	8	16	32	48
VQueue::front	268.246	308.583	72.270	59.902	62.246
BuildDate::buildDate	571	2.078	6.598	14.716	23.213
VIndexWithRestartable::index	254	55	1.053	1.095	2.561
VQueue::rear	5.791	10.077	6.392	6.674	4.930
VLinkedSet::entries	2.890	4.569	878	765	983

Fig. 10. Number of conflicts that the top 5 most conflicting VBoxes experience when using STMBench7 after optimizing the Atomic Part, Large Set, Manual, and VIndex implementation. We obtained these results using the write-dominated workload. We describe further details about the experimental methodology in Section 3.1.

Despite reducing the number of conflicts on the most conflicting VBox, Figure 11 shows that the optimizations that we present in this Section have negligible effects on the throughput and restart rate. To understand such behaviour, please keep in mind that two transactions may conflict on more than one VBox. This results suggest that transactions that conflicted on VBox `Vindex::index` also conflicted on another VBox. Therefore, the benchmark as a whole experiences the same number of conflicts, thus yielding a similar throughput and restart rate.

3.6 Id Pool

Until now, we identified 4 STMBench7's objects that experience a high conflict rate using the conflict detection technique that we describe in Section 2. Then,

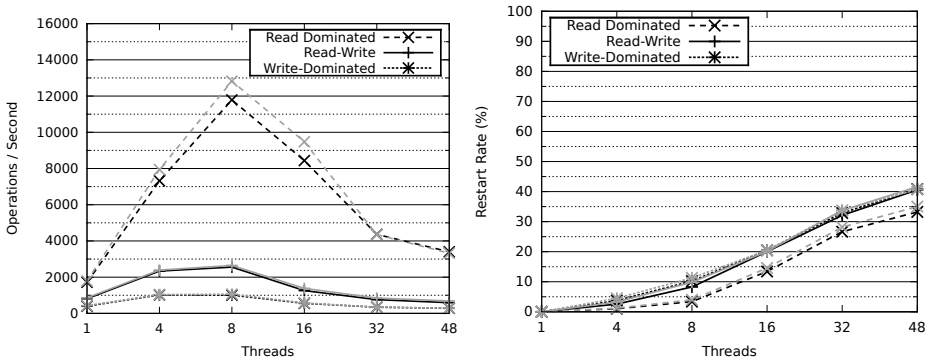

Fig. 11. Throughput and restart rate obtained when benchmarking JVSTM with STMBench7. The chart shows the results after optimizing the Atomic Part, Large Set, Manual, and VIndex implementation, as well as the previous results (Figure 9) in grey. We describe the experimental methodology we used to obtain these results in Section 3.1.

we made those objects concurrent friendly using some JVSTM techniques. Now, the VBox that experiences the largest number of conflicts is `VQueue::front`, as Figure 10 shows.

Class `VQueue` is an utility class that JVSTM provides. It is a transactional linked-list. The JVSTM direct implementation of STMBench7's objects uses VQueues to implement ID pools. STMBench7 generates all IDs a-priori, thus inserting them into the respective ID pool at the beginning of the benchmark. Then, during the benchmark, when it creates a new object that must be put into an index, it uses an ID pool to get a new ID for that object. When it deletes an object, it puts the now unused ID back into the respective ID pool.

The VQueue's put and get operation are implemented in a way that displays a low probability of conflicting. Therefore, transactions may ask a new ID whilst other concurrent transactions put back unused IDs without significantly increasing the overall conflict rate. However, considering a set of concurrent transactions that ask for a new ID from the same pool, using a VQueue to implement that ID pool makes all the concurrent transactions to get the same ID from that pool. The first transaction to reach the commit phase is able to commit, thus using the ID that was assigned to all these concurrent transactions. All the remaining concurrent transactions, when reaching the commit phase, detect the conflict and restart.

There is a key property of the ID pool semantics that we can use to make it concurrent-friendly: As long as all the IDs that the pool returns are unique and unused, the pool is free to return them by any possible order. In light of this property, we implemented each ID pool using several VQueues, one per thread, instead of a single VQueue. When initializing each ID pool, the IDs are divided randomly by all VQueues. Then, when the benchmark is running, each thread uses its own VQueue to get new IDs and put back unused IDs. When a thread T_i exhausts its VQueue, it starts stealing IDs from the VQueue of thread

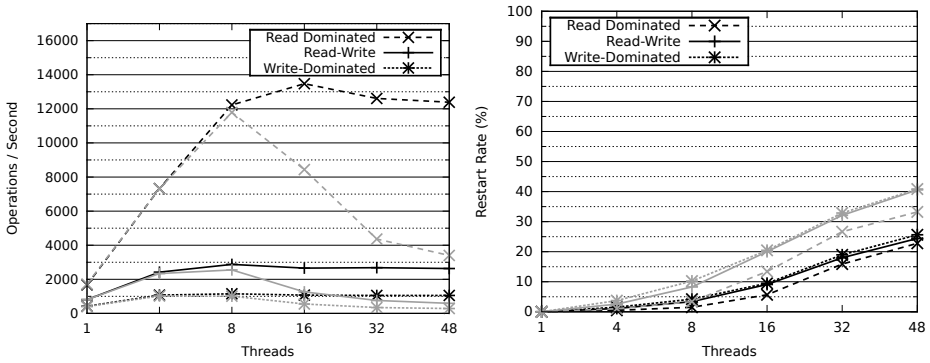

Fig. 12. Throughput and restart rate obtained when benchmarking JVSTM with STMBench7. The chart shows the results after optimizing the Atomic Part, Large Set, Manual, VIndex, and IdPool implementation, as well as the previous results (Figure 11) in grey. We describe the experimental methodology we used to obtain these results in Section 3.1.

$T_{(i+1)mod(N)}$, considering N threads. If that VQueue is also empty, thread T_i tries to steal from the VQueue of thread $T_{(i+2)mod(N)}, ..., T_{(i+N-1)mod(N)}$ until it finds an ID.

Figure 12 shows the throughput and restart rate that we measured after introducing the ID pool optimization. This optimization increases the throughput on all workloads. Moreover, it shows that the throughput on the read-write and write-dominated workloads does not drop after reaching the maximum value. As for the restart rate, the ID pool optimization decreases it on all workloads.

	4	8	16	32	48
VQueue::front	25.515	54.161	92.945	212.425	329.107
VLinkedSet::entries	3.723	13.442	35.492	89.122	146.182
BuildDate::buildDate	929	3.651	12.333	31.233	38.685
VQueue::rear	489	906	1.182	2.231	2.956
AtomicPartImplWithPerTxBoxes::to	333	400	550	580	1.044

Fig. 13. Number of conflicts that the top 5 most conflicting VBoxes experience when using STMBench7 after optimizing the Atomic Part, Large Set, Manual, VIndex, and IdPool implementation. We obtained these results using the write-dominated workload. We describe further details about the experimental methodology in Section 3.1.

Figure 13 shows the results of running the conflict detection technique, that we present in Section 2, after introducing the ID pool implementation. Compared to the results that we had before introducing this optimization, shown in Figure 10, we can see that this optimization actually increased the number of conflicts on ID pools. To understand this behaviour, keep in mind a transaction

that exhausts its ID pool now can add several VQueues to its read-set, therefore increasing the total number of VQueues involved in a single conflict.

4 Discussion

Throughout this paper we used a simple conflict detection technique to guide our optimizations, which raises concern about the accuracy of such technique. This is a relevant question, specially because the last optimization (described in Section 3.6) yields the best results, both in throughput increase and restart rate decrease. One might wonder if this optimization alone is responsible for the bulk of the improved throughput and restart rate, and whether all previous optimizations are really useful.

With that question in mind, we decided to reorder the optimizations, for instance swapping the last with the first, and selectively removing each optimization from the whole set while leaving all the others. We found out that we get the large performance boost only when we apply the last optimization. If we remove any optimization, the large performance boost stops. If we change the order by which we apply the optimizations, that large boost happens only when we apply the last optimization. This means that the conflict detection technique is accurate enough to detect the minimal set of VBoxes whose conflicts harm STMBench7's performance.

The JVSTM direct implementation of STMBench7's backend and core objects uses functional data structures inside VBoxes to make them transactional. This seems to be in conflict with the assumption that the direct implementation is what an average developer, without detailed knowledge of the internals of JVSTM, would do. However, please note that JVSTM does not ship with a large number of auxiliary data structures. Therefore, it is much simpler for the developer to use an existing functional data structure inside a VBox, treating it as a transactional black-box, than to modify some data structure's implementation, identifying the shared state and adding VBoxes to protect that shared state.

Besides STMs, STMBench7 also supports locks as a synchronization mechanism with two locking strategies: Coarse and medium grained locking. Figure 14 shows the throughput of the initial direct implementation of STMBench7's structures, the optimized implementation, and the two locking strategies. We can see that the initial implementation has higher throughput than the locks for read-dominated and read-write workloads, and only for 4 and 8 threads. The optimized implementation, on the other hand, is always higher than the two locking strategies for those workloads. For the write-dominated workload, the optimized version's throughput is near to the medium-grained locking until 32 threads. Figure 14 also shows that the optimized implementation scales up to 16 threads on read-dominated and up to 8 threads on read-write scenarios.

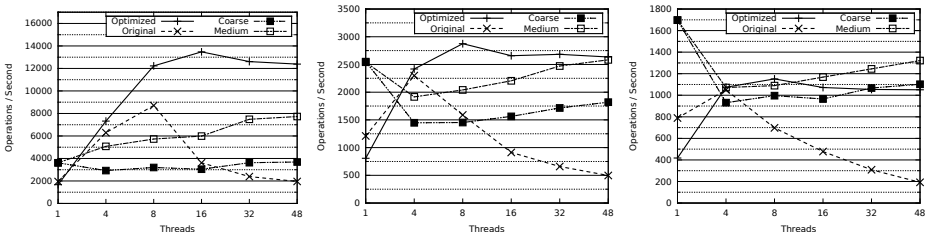

Fig. 14. This figure compares the performance of: (1) the direct JVSTM implementation of STMBench7's backend and core objects, (2) the optimized version of 1 that features all the optimizations that we present in this paper, (3) the coarse-grained locking strategy, and (4) the medium-grained locking strategy. Both (3) and (4) implementations already ship with STMBench7. This figure shows all three workloads, from left to right: Read-dominated, read-write, and write-dominated.

5 Related Work

Software Transactional Memory literature often considers two levels at which conflicts may occur and be detected: The *physical* level, and the *logical* level. A physical conflicts happens when a transaction reads a transactional location that another concurrent transaction wrote. On the other hand, a logical conflict happens at semantical level, when, for instance, two different transactions attempt to remove the same item from the same set. A physical conflict does not necessarily mean a logical conflict.

In this section, we describe techniques that other transactional systems use to reduce the conflict rate or to increase the throughput.

Herlihy et. al. introduced the concept of *early release* to reduce benign conflicts in DSTM [8]. With early release, the developer can remove transactional locations from the read-set as he sees fit. This allows that, for instance, two transactions add two items to separate locations on a long sorted linked list without conflicting: Transactions navigate through the list until they find the predecessor node of the value that they wish to insert. Those transactions early release each node that they read, leaving only the predecessor node on the read-set. Reducing the read-set effectively reduces the probability of a physical conflict, but it is up to the developer to guarantee that such reduction does not introduce semantical errors by eliminating logical conflicts.

Ni et. al. presented Open Nesting [9] as another technique to avoid benign conflicts. Typically, the effects of a committed nested transaction are visible to its parent only—closed nesting. The effects of an open nested transaction, on the other hand, are visible to the world as soon as the transaction commits. Open nested transactions can register handlers to execute some compensation actions, in case the parent transaction that encloses it aborts. Such compensation actions undo the effects of the open nested transaction. The developer must specify the compensation actions correctly, or else open nesting introduces semantical errors. Moreover, concurrently executing different abort handlers can lead to deadlock.

To avoid such deadlock, the programmer must ensure that open nesting contexts are partially ordered.

Harris and Stipić proposed Abstract Nested Transactions [6] (or ANTs, for short) to tackle the problem that benign conflicts pose. The developer identifies operations that are likely to be experience benign conflicts in ANTs. Then, the STM keeps a separate log for actions performed inside ANTs. When the STM detects a conflict on a transaction, it is able to determine if that conflict was generated from operations made inside an ANT. If so, the STM re-executes that ANT instead of the whole transaction. Finally, the STM can detect benign conflicts if the result of the re-execution is the same as the original execution.

Abstract Nested Transactions are very similar to Restartable Transactions, presented in Section 3.5. There are, however, some differences on how ANTs and Restartable Transactions are implemented. Restartable Transactions, at this point, allow read-only operations only. ANTs support read-write operations provided that they do not write to the same transactional memory locations that the enclosing transaction does.

The idea of re-executing a part of the transaction at commit time to, somehow, increase concurrency can be traced back to the concept of *field calls*, used in traditional relational databases [3]. During the transaction, a field call is an write operation protected by a boolean guard that evaluates quickly against the contents of the database without changing it. Later, on commit time, the boolean guard is re-evaluated to check if the transaction is still valid. Thus, instead of forcing write transactions to hold exclusive locks for its whole duration, field calls allow such transactions to use lighter shared locks to execute the guard and acquire the heavier exclusive locks only at commit time and only for a short duration.

Outside transactional systems' literature, there is a large body of work on highly-concurrent linearizable objects, some that ensure wait-free of lock-free progress conditions. However, despite being linearizable, making such highly-concurrent objects transactional is not trivial. Herlihy and Koskinen propose Transactional Boosting [7], a methodology for transforming a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects. They treat each *base* object (the original linearizable object) as a black box and require each base object's operation to have an inverse. Transactional boosting is not generally applicable and may lead to deadlocks. It seems fit for specialized developers to code highly-concurrent transactional libraries that other developers can use modularly when building transactional applications.

6 Conclusion

In this work, we modified a direct JVSTM implementation of the STM dependent part of the benchmark STMBench7, with the main goal of reducing the conflicts to, ultimately, increase the scalability. To reduce the conflicts, we executed several rounds of: (1) Identifying the most conflicting VBox, (2) designing a new and concurrent-friendly implementation of the class that owns that VBox,

and (3) applying that optimization, measuring the impact and restarting this cycle until the restart rate drops below an acceptable point. In this case, we repeated this cycle for 5 different classes.

To help us in task 1, we implemented a conflict detection technique that gathers information about, for each class, which VBox participates in how many conflicts. Then, we sorted the list of VBoxes by total number of conflicts on the most concurrent scenario (48 threads and write-dominated workload). Finally, we used the class that owns the top most conflicting VBox as the input class to task 2.

To design concurrent-friendly versions of the conflicting classes, we used JVSTM specific techniques such as: (1) per transaction boxes, (2) restartable transactions, and (3) transactional futures. Techniques 1 and 2 have already been described earlier [1], but this is the first paper that assesses their performance. In fact, technique 2 was not implemented before. Besides the JVSTM specific techniques, we also used other more general techniques to design concurrent-friendly objects, such as splitting a single large object that experiences high contention into several smaller objects, splitting that contention and achieving higher degrees of concurrency.

Using the concurrent-friendly versions of the top conflicting classes, we were able to boost STMBench7's throughput on all workloads. Moreover, we were able to make STMBench7 scale up to 16 threads on read-dominated workload, where previously it scaled until only 8 threads. When comparing our optimized version with the lock-based versions of STMBench7, out optimized version outperforms locks on read-dominated and read-write scenarios except when running single-threaded.

This paper presents a conflict detection technique that is accurate enough to pinpoint the minimal set of VBoxes whose conflicts harm STMBench7's performance. It also shows that per transaction boxes, restartable transactions, and transactional futures are effective techniques for implementing concurrent-friendly objects. This paper also describes several optimizations performed to data structures. We strongly believe that STMs should ship with their own concurrent-friendly data-structures, and this paper shows what kind of optimizations can be applied to such data-structures. Moreover, if such data-structures existed a priori, the whole task of reducing the conflicts would be simpler.

6.1 Open Issues and Future Work

This work describes several techniques to implement concurrent-friendly transactional data-structures using JVSTM. We plan on using the lessons that we learned to develop a package of concurrent-friendly transactional data-structures to ship with JVSTM. We hope that such package will help developers to write scalable applications using JVSTM.

This work reports the first implementation of restartable transactions. The current implementation is, however, quite naive and needs further optimization. For instance, creating a restartable transaction is, currently, a heavy operation because it requires the copy of the transaction's write-map. It worked so well in

this work because STMBench7 used restartable transactions at the very beginning of each transaction, when the write-map is still empty.

There is an improved and lock-free version of JVSTM's commit algorithm [2] that removes the global commit lock from the commit algorithm. The current implementation of per transaction boxes and restartable transactions requires transactions to commit in mutual exclusion, therefore requiring the global commit lock. We plan on developing lock-free versions of both these techniques, to be able to use them with the lock-free commit algorithm. We hope such lock-free versions will be able to scale to a higher level of concurrency without degrading the overall performance.

Acknowledgements

We would like to thank Hugo Rito for writing the direct JVSTM implementation of STMBench7's backend and core objects.

This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and the RuLAM project (PTDC/EIA-EIA/108240/2008).

References

1. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63(2), 172–185 (2006)
2. Fernandes, S.M., Cachopo, J.a.: Lock-free and scalable multi-version software transactional memory. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. pp. 179–188. PPoPP '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1941553.1941579
3. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (1992)
4. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 175–184. PPoPP '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1345206.1345233
5. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: a benchmark for software transactional memory. In: EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. pp. 315–324. ACM (2007)
6. Harris, T., Stipic, S.: Abstract nested transactions. In: TRANSACT '07: 2nd Workshop on Transactional Computing (aug 2007)
7. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 207–216. PPoPP '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1345206.1345237
8. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing. pp. 92–101. PODC '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/872035.872048

9. Ni, Y., Menon, V.S., Adl-Tabatabai, A.R., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open nesting in software transactional memory. In: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 68–78. PPoPP '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1229428.1229442