Profiling and Tuning the Performance of an STM-Based Concurrent Program

Luís Pina 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 but writing concurrent programs that are correct and that display good performance still is a hard task. 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.

Categories and Subject Descriptors H.2.4 [*Systems*]: Transaction Processing; D.2.8 [*Metrics*]: Performance Metrics; D.1.3 [*Concurrent Programming*]

General Terms Experimentation, Performance, Measurement

1. Introduction

Until recently, programming multicores was considered a niche left for developers specialized in parallel programming. However, over the last years, multicores have become accessible to the common developer.

Transitioning to multicores is hard due to two problems that make writing concurrent programs a difficult task. First, it is hard to write correct concurrent programs. Software Transactional Memory (STM) is a step in the direction of solving this problem. It simplifies the task of writing concurrent programs that may take full advantage of multicore systems and, unlike other synchronizing techniques such as locks, STMs are immune to deadlock and transactions compose naturally. Second, it is hard to write programs that display good performance, which is the main reason for writing concurrent programs in the first place.

Unfortunately, multicores only add complexity to the performance problem. When writing a sequential program, the developer can use a profiler to know which method is called the most or how much time the program spends on each method. Then, he optimizes the program using this information.

However, when using an STM, the information that profilers report is not as useful. For instance, consider transactions that increment a shared counter in such way that the increment accounts for a marginal percentage of the overall transaction length. Consider, also, that this program displays a poor throughput because transactions conflict on the shared counter and get restarted over and over again. Clearly, using a profiler on this simple example does not help the developer to find the cause of the poor performance. The developer must know that the read and write of the transactional memory location that keeps the shared counter is causing a large number of conflicts.

Informing the developer about which memory locations cause transactions to restart is not enough to solve the performance problem. We must also provide tools for him to use to reduce those conflicts. On a sequential program, the developer solves the performance issue by using familiar tactics, such as writing a fast path for the common case or using a more efficient data-structure. On a transactional concurrent program, however, the developer must use different and unfamiliar[1] tactics to reduce the number of conflicts that afflict his program.

In this paper, we present a technique that uses the conflicts on the read-set of transactions to inform the developer about which transactional memory locations experience conflicts and cause transactions to abort and restart. We implemented such technique on JVSTM [1].

[1] Unfamiliar for the common developer, not new for the concurrent programming community.

The main contributions of this paper are:

1. The first empirical evaluation of two techniques for reducing conflicts on JVSTM previously described [1]: *per transaction boxes* and *restartable transactions*;
2. A technique to detect transactional memory locations whose conflicts cause JVSTM transactions to abort and restart;
3. A set of techniques to reduce the conflicts that transactional memory locations cause for STMs in general, and JVSTM in particular;
4. A case study that shows how to use contributions 1 and 2 to improve the throughput of STMBench7 on higher levels of concurrency.

The majority of the techniques that we present as part of contributions 2 and 3 are STM agnostic, although we implemented them for JVSTM. Even if different STMs ship with different tactics for reducing conflicts, we believe that the conflict detection technique can be ported to other STMs with little effort. Bringing the information about conflicts sheds light on the behavior of the application, thus increasing the confidence of the developer that uses the STM. We strongly believe that is an important step for the transition to multicores, paving the way for the common developer to adopt STMs to write his concurrent applications.

The remainder of this paper is structured as follows. In Section 2 we describe the effect that conflicts have on JVSTM's performance. In Section 3 we describe a technique for detecting which VBoxes cause conflicts and how many conflicts each VBox causes. In Section 4 we present the conflict reduction techniques that we used for JVSTM. In Section 5 we describe how we optimized STMBench7's JVSTM backend implementation. In Section 6 we compare our approach to reduce conflicts with other existing and similar approaches. Finally, we conclude in Section 7.

2. Conflicts and Performance

To write a concurrent program using an STM, the developer groups actions that access shared memory locations within transactions. Transactions may *commit*, making their whole set of changes atomically visible to all other transactions, or *abort*, discarding the changes and appearing as if the transaction never took place. Conflicts happen when the STM mechanism cannot serialize a transaction whilst ensuring an opaque history [3] of committed transactions.

JVSTM [1] is an STM that detects conflicts at commit time. When a transaction T_1 reaches the commit stage, JVSTM checks if all transactional locations (or VBoxes, in JVSTM terminology) in T_1's read-set are still valid — that is, if no other transaction T_2 that committed between T_1's start and commit has written to those locations.

When JVSTM finds a conflict on validating T's read-set, it aborts and restarts T. Therefore, when a transaction experiences a conflict, it loses all the progress that it made. A shared object that causes a large number of conflicts can thus render useless most of the work that a transactional program performs, severely hindering its performance.

Figure 1 shows how conflicts affect the throughput of a transactional program. We obtained the results that we present in 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). Each value that we report (either a point on a chart or a cell on a table) represents the average of 10 executions, which took place in sequence.

We used STMBench7 to generate the results that we present in this paper. STMBench7 is a benchmark to evaluate different STM implementations. There are a set of classes that each STM implementation must provide. Such classes are: (1) *backend objects*, typical data structures that every program uses, such as sets, bags, and indexes, or (2) *core objects* that STMBench7 interconnects to build a rich object graph. STMBench7 interconnects the instances of those classes and creates a rich object graph. Then, it uses several traversal operations to visit and modify parts of that graph atomically, resulting in a non-trivial concurrent behavior. It supports three different workloads, each one featuring a different mix[2] of read-only and read-write operations: (1) read-dominated, (2) read-write, and (3) write-dominated. To generate the chart on Figure 1, we used a direct implementation of those classes. By direct, we mean an implementation as a common developer unaware of STM internals would write. [3]

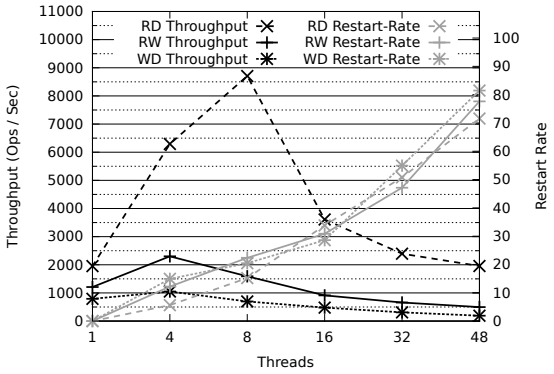

Figure 1. Throughput and restart rate obtained with the direct implementation of STMBench7. The black and grey lines are measured on the left and right y axis, respectively.

We can see in Figure 1 that STMBench7 tolerates conflicts without losing performance until the restart rate caused by those conflicts reaches 15%. This happens for the read-

[2] Operation types and relative percentages of each operation.

[3] The direct implementation was indeed written by a developer unaware of STM internals, at the time. Please see the acknowledgments section.

write and write-dominated workloads until 4 threads, and for the read-dominated workload until 8 threads. Past that restart rate value, the predominance of conflicts prevents STMBench7's throughput to improve with the increasing number of threads. This illustrates how deeply conflicts affect the throughput of programs that use STMs.

3. Detecting Conflicts

In an object-oriented language such as Java, programs are a set of classes. Some fields of those classes are accessed concurrently. Using JVSTM, the developer uses VBoxes to keep the shared fields' values and access their value transactionally. The value of the field itself, the VBox, never changes. Only the contents of that VBox changes.

To understand why a given VBox causes so much conflicts, developers are interested in knowing which field keeps that VBox. Thus, we added information about which field "owns" each VBox to usefully report the conflicts back to the developer. We implemented a Java agent that modifies the bytecode of every loaded class and performs the transformation that Figure 2 shows.

```
// Original
class C { final VBox box = new VBox(); }

// Transformed
class C { final InfoVBox box = new InfoVBox ("C::box"); }

class InfoVBox extends VBox {
  String ownerName;
  InfoVBox (String ownerName) { this.ownerName = ownerName; }
}
```

Figure 2. Transformation that the Java agent performs on every loaded class `C` with fields of type `VBox`.

When a transaction tries to commit, JVSTM starts by validating its write-set, thus ensuring that all VBoxes present on the read-set have not changed since the committing transaction started. If this validation fails, JVSTM aborts and restarts that transaction at the first conflict.

Our conflict detection technique is an extension of this mechanism: When validating a transaction, JVSTM validates all VBoxes present on the read-set instead of stopping on the first one that causes a conflict, and collects information about which VBoxes cause conflicts and in how many conflicts is each VBox involved.

Using the conflict detection technique on the STMBench7 setting that generates the most conflicts (48 threads, write-dominated workload) yields the information that Figure 3 shows. We can clearly see that class `AtomicPartImpl` owns the two VBoxes that generate most conflicts.

4. Reducing Conflicts

So far, we have described a technique that informs the developer about which VBoxes cause most conflicts and thus hinder the application's performance. But how can he reduce such conflicts?

AtomicPartImpl::y	64.052.709
AtomicPartImpl::x	64.052.709
LargeSetImpl::elements	93.991
LargeSetImpl::count	47.154
VQueue::front	18.245
ManualImpl::text	16.850
VIndex::index	15.210

Figure 3. Fields that cause most conflicts on the direct STMBench7's implementation (WD workload, 48 threads).

STMs typically explore the notion of *benign conflicts* to introduce mechanisms to reduce conflicts. A benign conflict poses as a false positive: Although the STM considers it a conflict due to the conflict detection that it performs, semantically that conflict is tolerable. Different STMs have different mechanisms for dealing with such conflicts, we leave that discussion to Section 6.

JVSTM uses a commit lock to ensure that only one transaction is committing at any given instant. To commit a transaction T, JVSTM starts by validating its read-set (ensuring that no other committed transaction modified any VBox read by T). Then, if T is valid, JVSTM writes back the values modified by T to the respective VBoxes, thus making T globally visible.

JVSTM's original proposal [1] introduces mechanisms for dealing with benign conflicts: *Per transaction boxes* and *restartable transactions*. We extended those mechanisms with *transactional futures*. Some of these mechanisms delay computations so that they execute inside the commit lock. We shall now describe each mechanism with a brief example.

4.1 Per transaction boxes

Consider a program in which we have a shared counter that every transaction increments at least once and only some transactions, with low probability, read the counter.

A naive implementation that reads the value of the counter, increments it, and finally writes the result back to the counter, performs miserably. This happens because, among N concurrent transactions, only 1 is able to commit. When each of the other transactions enter the commit stage, JVSTM detects that the counter has changed since the transaction read it and restarts the transaction.

A per transaction box, as the name implies, keeps a different value for each transaction. Moreover, they provide the developer with a hook to execute code at the commit stage, while still inside the commit lock and after the transaction's read-set is successfully validated. Developers can use a per transaction box to store how many times the program did increment the shared counter during a transaction. Then, they can use the hook to actually add the contents of the per transaction box to the shared counter.

By using a per transaction box, the increment operation does not register the shared counter on the transactions' read-set. Therefore, it eliminates all conflicts on that shared counter. It allows that several concurrent transactions increment the counter without conflicting with each other.

4.2 Transactional futures

Following the shared counter example, let us now consider that the transaction that increments the counter executes inside a method that returns the concrete value left on that counter after the transaction finishes. The transaction itself is not interested on the particular value of the counter. However, it must read the counter to return the value. This scenario precludes using per transaction boxes to avoid adding that counter to the transaction's read-set.

We propose *transactional futures* to deal with this scenario. A transactional future, like a regular future, represents the promise of a result. JVSTM is free to delay the computation of a transactional future until its value is used for the first time. If this first time does not happen during the lifespan of the transaction which originally created the future, JVSTM can compute its value inside the commit lock.

Getting back to the counter example, the transaction returns a transactional future to the enclosing method, which in turn returns the concrete value from that future after the transaction ends. This way, the shared counter does not reach the read-set and does not cause any conflict.

4.3 Restartable transactions

Consider a shared map backed up by a sorted linked list. From the map's point of view, there are two kinds of operations: *Find* and *modify*. A *find* operation, which is the most common, gets the item mapped to a key on the map, without modifying the map. A *modify* operation adds or removes items, modifying the map. Both types of operations must navigate through the list until finding the right position. Transactions group several operations together.

When they are navigating through the list, operations cause the surrounding transaction to add every list node read to the read-set. A modify operation changes the value of a list node. Therefore, all concurrent transactions with that node on their read-set shall fail to validate when they attempt to commit. As a result, a *modify* operation may cause several other transactions that execute the *find* operation to abort even when the *modify* operation modifies a different item.

JVSTM defines *restartable transactions* to deal with this problem. A restartable transaction wraps a read-only method's execution, keeping a separate read-set and saving the value that the method returns. If JVSTM fails to validate a restartable transaction's read-set, it re-executes the associated read-only method inside the commit lock and against the most recent version of the world. If the method returns the same result on the re-execution, the restartable transaction is still valid. In the example that we have been following, the developer would annotate the *find* operation to execute inside a restartable transaction.

5. Reducing Conflicts on STMBench7

To reduce the conflicts on STMBench7's JVSTM original implementation, we repeated several rounds of the following process: Running the conflict detection technique that we present in Section 3, and modifying how the direct implementation uses the most conflicting VBox. In this section, we describe how we optimized the direct implementation to reduce the number of conflicts. Due to space limitations, in this paper we give only a brief overview and description of the various techniques used to reduce conflicts in the STMBench7 implementation. A more detailed description of these techniques with all intermediate results after each individual optimization can be found elsewhere [9].

5.1 Atomic Part

There are two VBoxes on class `AtomicPart`, `x` and `y`, that are responsible for a large number of conflicts. The only method that writes to them, `swap`, swaps their value. We added a per transaction box `swapped` that holds a boolean that indicates if the conflicting VBoxes should be swapped at commit time. Method `swap` negates the value that the per transaction box keeps. This modification moves the real swapping and reading of the boxes to commit time, avoids adding the boxes to the read-set during the transaction and, thus, avoids conflicts.

5.2 Large Set

Class `LargeSet` represents a set of objects, backed by a functional red-black tree. Transactions use this set in one of three possible ways: (1) add objects to the set, (2) remove objects from the set, or (3) iterate over the set or check if a given object is part of the set. We added two per transaction boxes `added` and `removed` that hold sets of elements. Adding or removing an element just adds that element to the per transaction box `added` or `removed`, respectively. At commit time, each value in per transaction box `added` is added to the set, and each value in `removed` is removed.

5.3 Manual

Class `Manual` represents a singleton instance that keeps a large string in field `text`. The only method that modifies field `text` is method `replaceChar`, which replaces all occurrences of a given character by another character and returns the number of characters that it replaced. We modified the implementation of method `replaceChar` to return a transactional future. Such transactional future, when computed at commit time, replaces the characters on the string stored in VBox `text` and returns how many substitutions it performed.

However, this optimization actually yielded worst results when we introduced it. This happens because we moved

a large computation to execute inside the commit lock. [4] When STMBench7 emits an operation that involves using this computationally intensive transactional future, all other transactions that attempt to commit must wait for the commit lock. To mitigate this problem, we implemented a concurrent version of method `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.

5.4 VIndex

Class `VIndex` is an index that maps integer based keys to objects, backed by a functional red-black tree. To reduce the conflicts on the VBox that keeps the index, we used two per transaction boxes `added` and `removed`, similarly to what we did with class `LargeSet` that we explain in Section 5.2.

However, almost every STMBench7 transaction starts by consulting a `VIndex` instance. Transactions that modify the `VIndex` are much less frequent. To reduce those conflicts, we used restartable transactions to wrap the execution of the method that consults the index.

5.5 ID Pool

STMBench7 generates a-priori all integer-based identifiers that it associates with objects on `VIndex` instances. Class `IDPool` keeps the unused IDs. The direct implementation uses a queue to keep the unused IDs. When a transaction requests an unused ID, it gets the first element of that queue. This leads to a large number of conflicts because every concurrent transaction that requests an unused ID gets the same one. The first one to try to commit is able to do so, all others that use that same ID abort and get restarted.

We implemented a new ID pool that has one queue per each thread. Initially, IDs are scattered among all pools on a round-robin basis. To get an unused identifier, each transaction uses the queue of its thread, therefore avoiding conflicts with other concurrent transactions that also request IDs.

5.6 Experimental Results

This section reports the results that we obtained after introducing all the optimizations that we described previously.

We can see in Figure 4 that the optimizations that we introduced succeed at avoiding conflicts and thus keeping the restart-rate low (always below 25% when before they were as high as 80%). As a consequence, we can see that the throughput improves considerably on all three workloads. Moreover, the benchmark scales better: Its throughput peaks at 16 threads for all workloads, whilst previously it peaked at 8 threads on the best case. After reaching the peak, introducing more threads yields a slightly lower throughput.

[4] We discovered that method `java.lang.String.replace` was computationally intensive on this long string using a regular profiler.

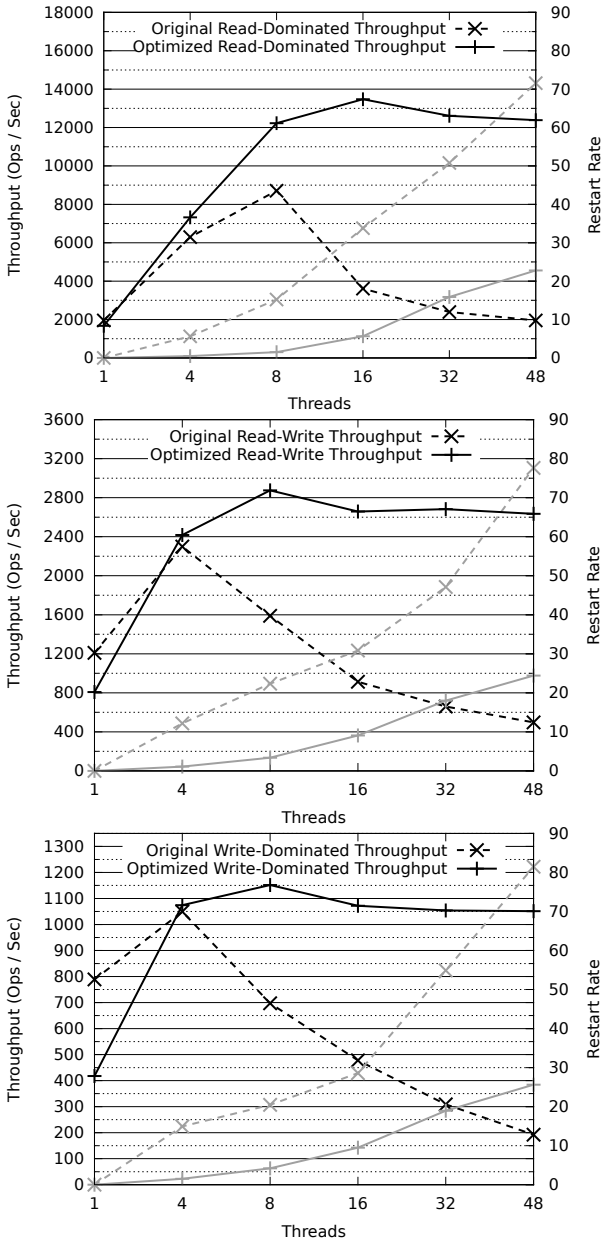

Figure 4. Throughput and restart rate obtained using STMBench7's optimized implementation. The black and grey lines are measured on the left and right y axis, respectively.

6. Related Work

To the best of our knowledge, little, if any, attention has been devoted to the problem of identifying which objects cause conflicts. However, several researchers in the past acknowledged the influence of conflicts on the performance of STMs and proposed mechanisms to reduce them.

For instance, Herlihy et. al. [7] introduced early release, and Ni et. al. [8] proposed open nesting. In both cases, the goal is to remove from the read-set elements that the programmer knows that are not relevant to the correctness of

the transaction. With early release, it is up to the developer to guarantee that the read-sets are correct without the removed elements. With open-nesting, the developer must write compensation actions to undo the effects of an open-nested transaction and must take care with the order in which such compensation actions take place to avoid deadlocks. Both of these approaches are thus very error-prone.

Herlihy and Koskinen propose Transactional Boosting [6], a methodology for transforming highly-concurrent linearizable objects into highly-concurrent transactional objects. Transactional boosting is not generally applicable and may lead to deadlocks. It is thus very error-prone and fit for specialized developers to code highly-concurrent transactional libraries that other developers use modularly.

In contrast to these approaches, we use per transaction boxes, transactional futures, and restartable transactions, which allow us to solve many of the same problems but in a less error-prone way.

The idea of re-executing a part of a transaction at commit time that restartable transactions explore can be traced back to field calls [2]. Our use of restartable transactions is similar to the advantages provided by Abstract Nested Transactions later proposed by Harris and Stipić [5].

7. Conclusion

In this paper, we present a simple technique that uses the conflicts on the read-set of transactions to inform the developer about which transactional memory locations cause conflicts, thus aborting and restarting transactions. We implemented such technique on JVSTM [1]. Furthermore, we put ourselves in the common developer shoes: We use this simple technique to identify which transactional locations prevent STMBench7's [4] throughput to scale to higher levels of concurrency, and use a set of techniques to reduce the conflicts that the implementation of STMBench7 on top of JVSTM experiences. Among those techniques , we included some JVSTM specific techniques that were previously described [1] (*per transaction boxes* and *restartable transactions*). We also include some other known techniques that are generally applicable to concurrent programs.

After introducing the optimizations that reduce conflicts on STMBench7's JVSTM implementation, we conducted an experimental evaluation. The results show that the optimizations succeed at avoiding conflicts, always keeping the restart-rate below 25% when before they were as high as 80%. As a direct result, we were able to boost STMBench7's throughput on all three workloads, enabling it to scale up to 16 threads, whilst originally it scaled only up to 8 threads on the better case. Moreover, when running STMBench7 with more than 16 threads, the throughput keeps near the peak performance value (even though below it), whereas previously it dropped considerably.

Writing correct concurrent programs is a hard task that STMs simplify. But developers are also concerned about performance, and current STMs do not provide tools to help developers finding performance bottlenecks or solve them. This paper presents a conflict detection technique that allows the developer to reason about performance in terms of conflicts between shared transactional memory locations. It also shows that per-transaction boxes, restartable transactions, and transactional futures are effective tools to decrease the conflicts that shared objects cause.

Acknowledgments

We would like to thank Hugo Rito for the direct implementation of STMBench7 on top of JVSTM that he wrote when he first started his MsC thesis work.

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] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[2] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[3] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184. ACM, 2008.

[4] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324. ACM, 2007.

[5] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[6] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[8] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 68–78, New York, NY, USA, 2007. ACM.

[9] L. Pina and J. Cachopo. Reducing conflicts on jvstm transactions - stmbench7: A case study. Technical Report 39, INESC-ID, August 2011.