On the design space of Parallel Nesting [*]

Nuno Diegues João Cachopo

INESC-ID Lisboa / Instituto Superior Técnico, Technical University of Lisbon

{nmld, joao.cachopo}@ist.utl.pt

Abstract

One of the key strengths of Transactional Memory (TM) is that transactions compose. Yet, very few TM implementations allow a transaction to be split into several parts that execute concurrently. To overcome this limitation in transaction's composability, TMs must support parallel nesting.

In this work we elaborate on how the design choices of the underlying TM are inherently coupled with the complexity bounds achievable in the support for parallel nesting. For that, we compare three different parallel nesting algorithms that explore opposite directions in the design space. We also show that the theoretical analysis of these algorithms may be deceiving and we support that conclusion with empirical results in known benchmarks.

1. Introduction

A major selling point of Transactional Memory (TM) [9] as a synchronization mechanism lies in its ability to compose code that uses transactions [8]. This is an advantage over explicit locking, which requires programmers to break abstractions if they want to be safeguarded from deadlocks and other pitfalls of locking.

To compose code that uses transactions, TMs rely on nesting [11]: A transaction T_i is nested within T_k if T_i is created within the control flow of T_k. Unlike with locks, creating nested transactions does not change the semantics of an outer transaction. Yet, most nesting implementations do not take into account parallelism within transactions. To handle these cases, TMs have to be extended with a more complex algorithm that supports parallel nesting. The challenge of this task lies in making that algorithm efficient: Its overhead cannot be prohibitive in a way that it supplants the parallelism being explored within the transactions.

In this paper we briefly compare three designs for implementing parallel nesting. In Section 2, we introduce our parallel nesting algorithm along with two alternatives that represent different design choices, which we analyse in Section 3. Then, in Section 4, we provide an empirical evaluation of these three designs. Finally, we discuss those results in Section 5 and we conclude in Section 6.

2. Design space of Parallel Nesting

Parallel nesting algorithms are severely restricted by the design choices inherited from the baseline algorithms. Therefore, it is important to understand those restrictions to be able to compare and to analyse the various designs.

In Section 2.1, we introduce JVSTM and how we extended it to support parallel nesting efficiently. Due to space restrictions, we describe this extension at a high-level only, but a thorough description of our design can be found in [4]. Next, we describe two alternative Software Transactional Memories (STMs) that provide parallel nesting and that are representative of the diversity of choices in the design space: NesTM in Section 2.2 and PNSTM in Section 2.3.

2.1 Parallel Nesting in the JVSTM

The Java Versioned STM (JVSTM) [5] is a word-based, multi-version STM that was specifically designed to optimize the execution of read-only transactions: Read-only transactions have very low overhead and never contend against any other transaction. In fact, once started, their completion is wait-free in JVSTM.

To achieve this result, JVSTM maintains the history of values that have been written to each transactional location. The values in a history are versioned by a global clock, and each value is kept as long as some alive transaction may need to read it; otherwise, it is discarded by the garbage collection algorithm. Therefore, a read-only transaction always commits successfully because it reads values in a version that corresponds to the most recent version that existed when the transaction began, thus being serialized in that instant. Read-write transactions, however, must be serialized when they commit. Therefore, they are validated at commit-time

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

WTTM '12 19 July, Madeira, Portugal
Copyright © 2012 ACM . . . $10.00

to ensure that values read during its execution have not been changed in the meantime by another concurrent transaction.

In our extension to support parallel nesting, writes are made in-place, if the transaction is able to acquire the ownership of the transactional location. This means that only a transaction (and its children) may use that in-place slot at a given time. This fast-path is expected to be the common case. Writes applied in the same nesting tree to a variable form a linked list in which the head entry is the most recent and its owner is a descendant of all the owners of the entries that follow it. This way, reads are able to peek directly the head of that list and return immediately in the common case.

Even in the case that the read operation needs to traverse that list, it executes in a bounded number of steps and independently of concurrent transactions, thus being wait-free. In contrast to this, the write operation may be delayed by concurrent transactions in the same nesting tree. Moreover, the commit of a parallel nested transaction acquires a lock on its parent, and, therefore, contends for the same lock as the commit of its siblings. So, commits of parallel nested transactions at different nesting levels or nesting trees proceed independently.

The original JVSTM is a lock-free STM, because no top-level transaction can prevent the system as a whole from making progress. But, when we take into account the parallel nested transactions spawned in the scope of a nesting tree, we are implicitly attributing the following semantics to them: All of the siblings must commit successfully for the parent to be able to commit—that is, it does not make sense for the parent to succeed while some of its children are still executing. Therefore, if one sibling does not progress, the parent will not be able to progress either. In this sense, it is not harmful if a nested transaction inhibits its siblings from progressing, as the progress of the nesting tree as a whole requires that all the transactions in it succeed. For this reason, our extension of JVSTM to support parallel nesting is blocking. Note that this does not change in any way the fact that the top-level transactions in JVSTM are still lock-free: A parallel nested transaction may never prevent other transactions (nested or not) from progressing unless they have a common ancestor, meaning that they belong to the same nesting tree.

Finally, the commit procedure validates the read-set and then propagates its ownership records to its parent. A transaction starts with one ownership record and accumulates the records of its descendants, as they commit.

2.2 NesTM

The Nested STM (NeSTM [2]) is based on McRT-STM [12] as a blocking, eager conflict detection, word-granularity TM with undo logs for writes and a global version clock for serializability. Each address is mapped to a variable that acts both as a lock and as storage for a version number. The former contains the address of a structure with information about the transaction holding the variable, whereas the latter contains the global clock version corresponding to the last write applied to the address that is mapped by the variable.

Whereas the original McRT-STM design assumes that no other transaction can access a locked variable, that is no longer true with parallel nested transactions: In this case, other transactions can correctly access the locked object as long as they are descendants of the owner.

At transaction start, the global clock is used to timestamp the transaction. A read access to x causes an abort if x was written after the reader started.

When writing a value, a transaction attempts to acquire the lock corresponding to the variable and then it validates the variable: The transaction attempting to write, as well as its ancestors, must not have a timestamp smaller than the variable's timestamp, in case they read it previously. To reduce the work needed for this validation, only transactions that were not ancestors of the previous owner of the object must go through the check.

The nested commit procedure requires validating the reads, across the transaction and its ancestors, followed by the merge of the sets into the parent. This set of actions must be atomic in the algorithm. This is meant to prevent concurrent siblings from committing simultaneously and breaking opacity [6]. In practice, this is solved by introducing a lock at each transaction and making nested transactions acquire their parent's lock in mutual exclusion with their siblings.

Moreover, NeSTM is subject to livelocks: If T_1 writes to x and T_2 writes to y, they will both have acquired the ownership of those variables. Now, if the first transaction spawns $T_{1.1}$ while the second one spawns $T_{2.1}$ and both these nested cross-access y and x, respectively, they will abort because those variables are owned by non-ancestors in each case. However, they will have mutually blocked each other unless one of their ancestors aborts as well and releases the corresponding variable. The authors placed a mechanism to avoid this in which they heuristically count consecutive aborts and abort the parent as well.

2.3 PNSTM

The Parallel Nesting STM (PNSTM [3]) provides a simple work-stealing approach with a single global queue such that the programs' blocks may be enqueued for concurrent transactional execution. A stack is also associated with each transactional object containing accesses (not distinguishing between reads and writes) performed by active transactions. This allows transactions to eagerly determine in constant time if a given access to an object conflicts with a non-ancestor's access.

When a transaction commits, it leaves behind traces in all the objects it accessed, namely the stack frames stating its ownership. To avoid having to go through all the objects in the write-set by locking and merging the frame with the previous entry, PNSTM does that lazily. This may lead to false conflicts when some transaction accesses an object and finds an entry in the stack that corresponds to an already

	JVSTM	NesTM	PNSTM
read	$O(maxDepth)$	$O(k)$	$O(1)$
write	$O(1)$	$O(txDepth)$	$O(1)$
commit	$O(r + children)$	$O(r + w)$	$O(1)$

Figure 1. Complexity bounds for the worst-case of transactional operations in parallel nested transactions.

	JVSTM	NesTM	PNSTM
r-r	-	-	yes
r-w	yes	yes	yes
w-w	yes (if nested)	yes	yes

Figure 2. Possible conflicts that may lead to abort in each STM. Note that we refer here to read-write transactions in the case of the JVSTM.

committed but not yet reclaimed transaction. The authors show that it is possible to avoid these conflicts by resorting to a global structure maintaining data about all committed transactions and some lazy cleaning up.

3. Analysis of the STMs

All the STMs described provide opacity [6]. Yet, in the rest, they explore different design choices. As a consequence, they provide different bounds and guarantees to applications using them. Figure 1 lists the worst case bounds in the operations of a transaction in the mentioned STMs, where: k is a constant value defined statically in NesTM; r and w are the size of the read-set and write-set, respectively; $children$ is the number of descendants of a transaction; $maxDepth$ is the maximum depth of the nesting tree; and $txDepth$ is the depth of the transaction executing the operation.

JVSTM is the only one where the read operation depends on the maximum depth of the nesting tree. This worst case may happen if the variable being read has been written by all the transactions in a given branch of the nesting tree (and necessarily different from the branch of the reader transaction). The read in NesTM may need to be repeated k times, where k is defined statically. The access has to read the value and lock separately, and thus uses a double read pattern to ensure a consistent reading. A repetition takes place when a concurrent abort occurs between the double read. Finally, PNSTM only needs to look at the top of the access stack to decide in constant time if the access is conflict-free.

Regarding the write operation, both JVSTM and PNSTM need only to insert the new value in a single location (reachable in constant time). On the other hand, NesTM may need to validate the variable being written in all the ancestors' read-sets.

The commit operation is also performed in a constant number of operations in PNSTM. On the other hand, NesTM needs to validate the read-set and to propagate the ownership of the writes to the parent. JVSTM also requires the validation, which cannot be avoided given its lazy update nature [1], but has a different bound for the propagation of the write-set: It depends on the number of children of the committer.

The bounds presented thus far are clearly in favor of PNSTM, but in practice, what affects more the performance of an STM is the complexity of the common case, rather than the complexity of the worst case, if they are rarely the same. For instance, the read operation of JVSTM has a worst case that seems very unlikely to happen in practice because it requires that all the transactions in a nesting branch write to the same variable. This single fact is of great importance as the read operation is typically predominant in the applications.

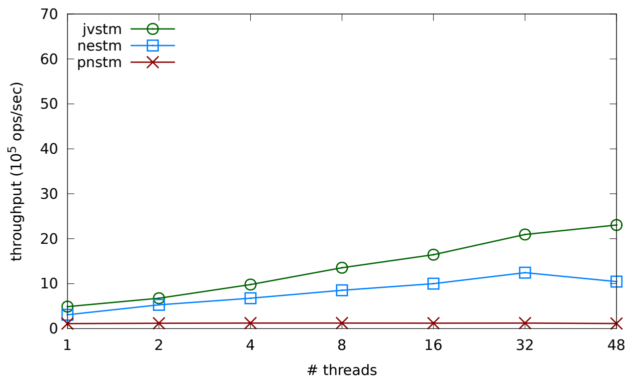
Moreover, providing constant time operations in parallel nesting does not come for free. The design choices of the STMs are reflected in the types of conflicts that may lead to abort, which we summarize in Figure 2.

As we may see, there is a relation between the complexity bounds and the conflicts detected: The cheaper the worst case complexity bounds are, the more conflicts the STM has to detect to guarantee correctness. There is no perfect solution as we are in face of a trade-off. Despite having constant-time worst-cases, PNSTM limits severely the concurrency of transactions whose footprints intersect with each other.

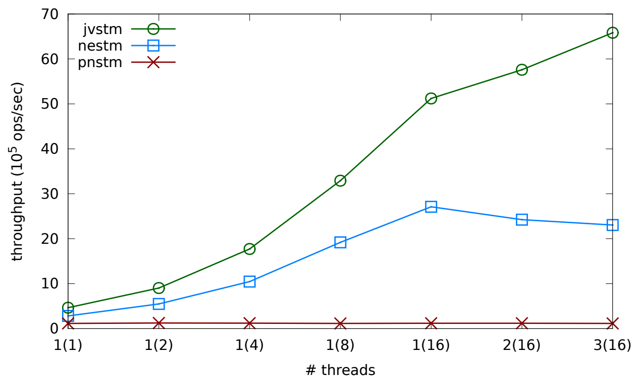
4. Practical comparison

In this section we present the results of an experiment that we conducted to assess the practical effect of the design choices of the STMs. All tests were carried on a machine with four AMD Opteron 6168 processors (48 cores total) and 128GB of RAM (using at most 4GB), running Red Hat Enterprise 6.1 and Oracle's JVM 1.6.0_24. We implemented both NesTM and PNSTM according to the algorithms in their publications (this was the only alternative given that they are not publicly available nor were we able to reach the corresponding authors). To make the comparison fair, we defined an API for these implementations that allows the applications to specify which locations are transactional. This way, we ensure that we are adding the same level of instrumentation to all STMs. Still, our evaluation of the different parallel nesting implementations is necessarily affected by the underlying TM designs. Because of that, we also executed the workloads using only top-level transactions, so that we may have an idea of how the baseline TMs compare with each other.

Figure 3 presents the average number of operations executed per second for a contended workload with dominance of reservations in the `Vacation` benchmark [10]. Looking at Figure 3(a), we may see that JVSTM is considerably faster than the alternatives when using only top-level transactions. In particular, it is 58% faster than NesTM and 441% faster than PNSTM with a single top-level transaction. Given that

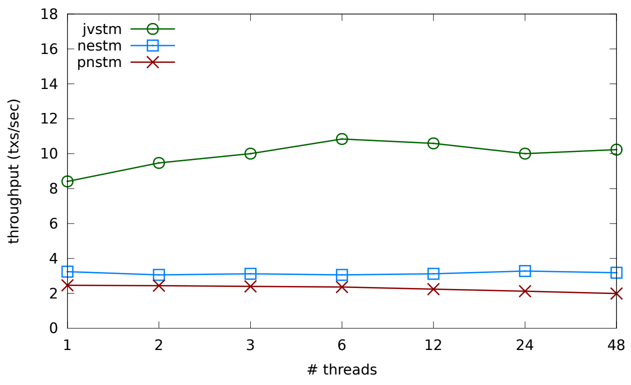


(a) Using only top-level transactions.

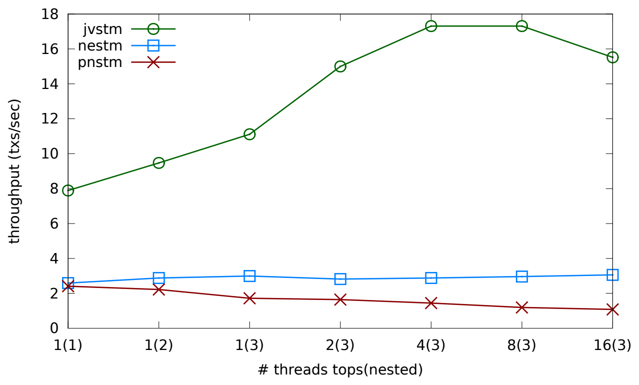


(b) Using parallel nesting.

Figure 3. Throughput in a contended workload in the `Vacation` benchmark. When using parallel nesting, the number of top-level transactions used is followed in parentheses by the number of nested transactions that each one may spawn.



(a) Using only top-level transactions.



(b) Using parallel nesting.

Figure 4. Throughput in the write-dominated workload of the `STMBench7` benchmark with long-traversals enabled. When using parallel nesting, the number of top-level transactions used is followed in parentheses by the number of nested transactions that each one may spawn.

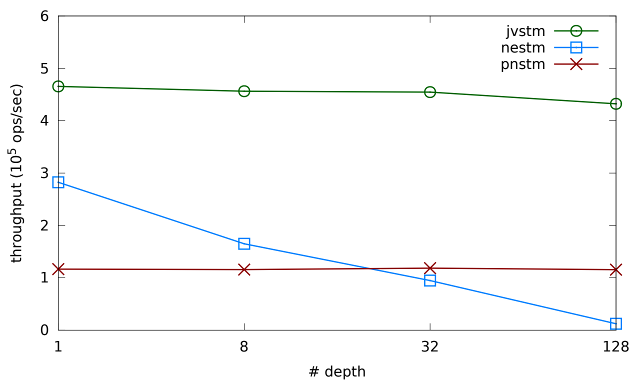
the baseline JVSTM is already faster than NesTM, it is expected that using parallel nesting is also faster in JVSTM. We can see that in Figure 3(b). Still, the actual improvement (for 48 threads) is greater for JVSTM (2.8 times) than for NesTM (2.2 times) and PNSTM (no improvements).

We also show a comparison between the three STMs in a write-dominated workload of the `STMBench7` benchmark [7], with long-traversals enabled. The results are shown in Figure 4, where we may see that JVSTM is again faster already with only one thread: It is 2.6 and 3.4 times faster than NesTM and PNSTM, respectively, when using only top-level transactions. Moreover, even though we do not show them, the results are very similar across the other workloads of the `STMBench7`.

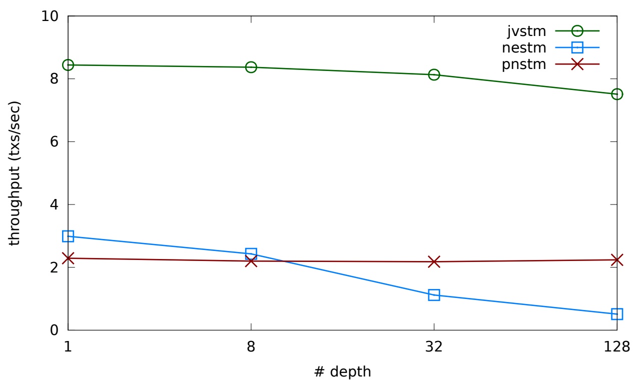
Yet, unlike the results obtained in the `Vacation` benchmark, in this case the parallel nesting algorithm of NesTM is unable (together with PNSTM) to obtain improvements over its execution with only top-level transactions.

Finally, in Figure 5, we present results that show how each STM performs when the nesting depth increases. To obtain these results, we modified both benchmarks so that they execute all of their transactions entirely within a single nested transaction at a certain depth. This yields a nesting tree with a single branch that is increasingly deeper as the nesting increases up to a level of 128.

These results are consistent with the theoretical complexity bounds of each STM. Namely, PNSTM performs independently of the nesting depth, whereas the other two degrade their performance. However, JVSTM not only performs significantly better, but it also degrades at a much slower rate than NesTM. This behavior is similar in both benchmarks shown and is representative of the data that we collected in several other workloads.



(a) Vacation.



(b) STMBench7.

Figure 5. Throughput obtained with a single nested transaction at an increasing nesting depth.

So, just as PNSTM gets better results than NesTM for a sufficiently high depth, we expect the same to happen also, at some depth, with regard to JVSTM. Yet, given the slow decay of the JVSTM, that will require a much higher nesting depth (note that the horizontal axis is growing exponentially). In fact, we argue that such depth would seldom, if at all, be seen in real applications, specially taking into account that these results hold only for a single thread.

5. Discussion

A naive interpretation of the worst-case complexity bounds for the STMs presented in Section 3 could lead us to conclude that PNSTM should obtain the best performance (followed by NesTM). Yet, the results shown in Section 4 contradict that conclusion. This happens for two reasons: (1) the time that it takes to execute a transaction does not always fall under the worst-case; and (2) the amount of conflicts detected influences greatly the resulting performance. In this section we look at both reasons to understand why JVSTM obtained the best performance despite the analysis in Section 3.

We first look at the usage of fast paths in the algorithms that avoid the worst-cases. For that, we used an execution with 48 threads in the write-dominated workload of the `STMBench7`. For JVSTM, we verified that the in-place metadata was used in 43% of the reads [1]. Moreover, the reads that were not a read-after-write were able to avoid checking the in-place metadata in over 99% of the times. This means that nearly all the time is spent in the fast paths of JVSTM.

Table 6 presents this data together with the corresponding data for the other two STMs. We considered that the fast path in both of them is the read-after-write, because, given their eager in-place nature, such operation is considerably cheaper than a normal read. JVSTM has two possible fast paths: one when it reads in-place, and another when it reads a globally consolidated version without having to check the in-place slot. As we may see, JVSTM is able to execute nearly all of the times in either of these two modes, whereas the other two STMs go through their fast path only 39% of the times. Moreover, when reading in-place, JVSTM was able to obtain a write entry immediately without incurring in the worst-case of the read operation in over 99% of the time.

The immediate consequence of these fast paths is visible in the average time to execute a transaction. To assess this, we took into account only executions of transactions that committed: JVSTM 1046μs; NesTM 5200μs; PNSTM 7357μs.

To address the second reason, we show, in Figure 7, the conflicts registered in an execution with 48 threads in the write-dominated workload of the `STMBench7`. These results are consistent with executions with a decreasing number of threads and show that JVSTM had the least number of conflicts detected. Namely, NesTM detected approximately twice more conflicts, whereas PNSTM detected a hundred times more conflicts.

Adding to this, we also registered the percentage of transactions that failed to succeed in their first attempt in an execution with 48 cores:

- `STMBench7`: JVSTM 0.1%; NesTM 0.4%; PNSTM 6%;
- `Vacation`: JVSTM 27%; NesTM 36%; PNSTM 98%;

The general trend of these statistics follows the expectation according to the conflicts that are detected in each STM (as presented in Figure 7): The less conflict types detected, the smaller the percentage of unsuccessful transactions. Note that the conflicts in the execution in `STMBench7` are much smaller because of a serialization imposed by a scheduler. Nevertheless the relative degree of conflicts between each STM is consistent with the other results and withstands the point raised.

The data presented in this section backs up the two reasons pointed out for the higher performance of JVSTM.

[1] These reads correspond to read-after-writes.

JVSTM			NesTM		PNSTM	
Fast Path	Slow Path	Read Inplace	Fast Path	Slow Path	Fast Path	Slow Path
0.56	0.01	0.43	0.39	0.61	0.39	0.61

Figure 6. Fraction of occurrence of each type of read per STM in an execution of a write-dominated workload in `STMBench7`.

JVSTM			NesTM			PNSTM
R-W (eager)	R-W (commit)	W-W (parNest)	R-W (commit)	Spurious	R-W (eager)	R-W (eager)
465	177	203	39	1465	123	84496

Figure 7. Occurrence of conflicts per STM in an execution of a write-dominated workload in `STMBench7`.

Even though this comparison was performed against our implementation of the other STMs (rather than implementations provided by their authors), we believe that the conclusions still hold, as we may single out some aspects that explain why JVSTM obtained the best results: Its multi-version property allows more concurrency (by detecting less conflicts), which results in less aborts and re-executions; moreover, its many optimizations allow fast paths that result in faster executions in the common case.

Conversely, NeSTM suffers mainly from its costly validation procedure, naive merging at commit-time, and spurious aborts (which serve to avoid livelocks heuristically). On the other hand, in the case of PNSTM most of the overhead comes both from the reads that have the same costly path as the writes, and from its read-read conflict detection.

This also strengthens the idea that we cannot reason about the performance of each parallel nesting approach independently of the underlying TM used, because the quite different design choices of the underlying TM may have a significant effect on the performance of the parallel nesting algorithm. In particular, PNSTM was designed specifically with the purpose of providing a parallel nesting algorithm that performed completely independent of the nesting depth. Unfortunately, that decision makes it extremely inefficient in practical applications such as those mimicked by these benchmarks. A similar conclusion applies to NesTM because it uses single versions and, thus, read-only transactions may abort.

6. Conclusions

We described and analysed three different STMs that support parallel nesting. That analysis suggested that the parallel nesting of JVSTM would perform worse due to the higher worst-case complexity bounds of its operations. Yet, as we discussed, the performance of each parallel nesting approach is heavily influenced by the design decisions of the underlying TM that enable those worst-case complexity bounds.

To assess the consequences of those decisions, we provided an evaluation in two well-known benchmarks. Although the parallel nesting algorithm of JVSTM is theoretically worse, it contributed efficiently to the improvement of the performance over its baseline implementation. On the other hand, the alternatives either obtained weaker results or could not improve at all. In particular, PNSTM was tailored to suit better parallel nesting but failed to be adequate in practice for a wide variety of workloads.

References

[1] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 163–174, New York, NY, USA, 2008. ACM.

[2] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 253–262, New York, NY, USA, 2010. ACM.

[3] João Barreto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 91–100, New York, NY, USA, 2010. ACM.

[4] Nuno Diegues and João Cachopo. Exploring parallelism in transactional workloads. Technical Report RT/16/2012, INESC-ID Lisboa, June 2012.

[5] 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.

[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] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[8] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM Press, 2005.

[9] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[10] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35 – 46, sept. 2008.

[11] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, December 2006.

[12] 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.