Technical Report RT/1/2012

Review of Nesting in Transactional Memory

Nuno Diegues
INESC-ID/IST
nmld@ist.utl.pt

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

Jan 2012

Abstract

In face of the multicore paradigm that is now standard in most machines, we are left with the challenge of how to take advantage of all the potential parallelism: Programming applications that explore concurrency is far from trivial. It is in this context that the Transactional Memory (TM) abstraction promises to simplify this task.

So far, each part of an application identified as atomic is typically a sequential set of instructions. However, these may account for a large part of the program which is thus running sequentially and may represent bottlenecks in the system. These atomic blocks could be executed faster if they contained latent parallelism that could be used efficiently. In the context of Transactional Memories, however, there has been little work to allow a transaction to be split in several parts, all executed concurrently. Such behavior may be attained through parallel nested transactions.

Besides providing a more flexible TM, using parallel nested transactions may actually yield better performance in applications resorting to TM for synchronizing access to shared data. In this work we describe how we may still reduce the time to execute a highly-contending write dominated workload despite the inherent conflicts that arise in it. We expect to provide an adaptation of a lock-free and multi-version STM, the JVSTM, to use a parallel nesting model that allows the aforementioned flexibility.

Keywords: Transactional Memory, Nested Parallel Transactions, JVSTM, Multi-version, Lock-freedom

Technical Report RT/1/2012[1]

Review of Nesting in Transactional Memory

Nuno Diegues
INESC-ID/IST
nmld@ist.utl.pt

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

1. Introduction

Up until 2004, multiprocessor computers were only seen on research laboratories or as enterprise servers. Since then, we have reached a hard physical limit dictating the decline of Moore's Law applied to processors' frequency [46]. For many years, applications benefited from automatic improvements in performance as processors were upgraded with increasing clock speeds. However, as Sutter entitled his article in [47], "The Free Lunch is Over", in the sense that programmers can no longer expect their applications to become faster as they used to.

The other side of the coin is that processors are still becoming more powerful due to the increasing number of cores. Multicore architectures are now a commodity: This has spurred the interest on easing the development of concurrent programs for shared-memory multiprocessors. As a matter of fact, concurrent programming has been used for many decades, but it is only now that it is becoming an increasing trend affecting the daily life of programmers' beyond a niche of researchers.

As we will see, it is not easy to synchronize concurrent accesses in large applications by resorting to traditional mechanisms such as locks. Transactional memory (TM) is an appealing abstraction for making concurrent programming accessible to a wide community of nonexpert programmers while avoiding the pitfalls of mutual exclusion locks. In this work we shall present why this alternative is worth researching as well as several of its possible design choices and implementations. In particular, we depict the transaction composability models that state how a TM system handles transactions that may themselves create new transactions. Although several models have been proposed, which we shall address in this work, many TM proposals have ignored this matter of composability between transactions. The emphasis of this work lies in the model that allows more than one transaction to be composed in the same transaction and still run concurrently, thus forming a set of sibling transactions with a common parent. As we shall see, this parallel nesting model not only allows more flexibility to the programmer, but it may also increase the performance of a program synchronized with a TM system.

The structure of this document is as follows. In Section 2., I introduce the basic concepts used by a TM and the need for nested parallel transactions. Section 3. gives a brief overview of guarantees that may be provided by a TM algorithm. Then, in Section 4. we present the existing systems related to nesting regarding various design decisions, nesting models, and corresponding implementations that exist in the literature. Finally, in Section 5. we present some conclusions.

2. Synchronization of Concurrent Operations

The synchronization of concurrent programs has been traditionally achieved by resorting to blocking synchronization techniques such as locks, semaphores, monitors, and conditional variables. Although

[1]This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds, by the RuLAM project (PTDC/EIA-EIA/108240/2008) and was partially supported by the Cloud-TM project, which is co-financed by the European Commission through the contract number 257784.

these have been somewhat the consensual approach, they are not free from many pitfalls.

A usual scenario, upon which we shall build the examples, portraits concurrent objects, meaning that their methods may be called in such a way that the invocation intervals overlap each other. Typically such objects contain some state that is thus protected by some mutual exclusion lock. However, if we consider an operation that requires manipulating several of these objects without allowing intermediate states to be observed, it follows that we cannot simply rely on the individual lock acquisition that takes place inside the object. That is, the lock that protects each object is not enough for preventing inconsistent states for the outer operation. As a result, traditional techniques usually resort to additional locking that guarantees none of the objects may change while that bulk action takes place. This solution, however, is prone to deadlocks. Depending on the strategy used for the multiple lock acquisition, it may happen that concurrent threads acquire one or more of the locks each and remain indefinitely trying to acquire the rest.

Whereas there are techniques to avoid deadlocks, such as establishing some total order among the elements to lock, they are hard to apply in practice. In the past, when highly scalable applications were rare and valuable, these hazards were avoided by dedicating teams of expert programmers to develop these algorithms. Today, when highly scalable applications are becoming commonplace, the conventional approach is just too expensive. As claimed by Herlihy and Shavit: "The heart of the problem is that no one really knows how to organize and maintain large systems that rely on locking" [25, p. 418]. In practice, the association between locks and data is established mostly by convention and is not explicit in the program. Ultimately, this ordering convention exists only in the mind of the programmer, and may be documented only in comments as shown in [30]. Additionally, "Over time, interpreting and observing many such conventions spelled out in this way may complicate code maintenance" [25, p. 418].

The issues do not concern only deadlocks: Overall, locking, as a synchronization discipline, has many pitfalls for the inexperienced programmer [24]. Two other common problems with lock-based synchronization are priority inversion and convoying. Priority inversion occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads. Convoying may also occur when a thread holding a lock is descheduled, perhaps by exhausting its scheduling quantum, by a page fault, or by some other kind of interrupt. While the thread holding the lock is inactive, other threads that require that lock will queue up, unable to progress.

However, the fundamental flaw is that locks and conditional variables do not support modular programming: The process of building large programs by gluing together smaller programs. Creating software on top of modules, which synchronize access to shared data with locks, may entail finding out internal locks that are acquired as well as their order. However, not only this is impractical, but it also breaks the abstraction that was supposedly provided by the modules.

2.1. Transactional Memory

One of the alternatives for synchronization between concurrent units of work is the transactional memory abstraction [28]. The programmer is responsible for identifying, in his program, atomic blocks that the TM runs within transactions. A transaction is a dynamic sequence of operations executed by a single thread that must appear to execute instantaneously with respect to other concurrent transactions. The purpose is that this set of operations is seen as an indivisible action, meaning that they appear to execute sequentially in a one at-a-time order. Despite this traditional definition, we show that one may decouple a transaction from a single thread with resort to parallel nesting models in Section 4.5..

As a consequence, operations enclosed in transactions are given the illusion of no concurrency. If a transaction fails, it is as if it never ran (no partial executions). A failed transaction may be retried, depending on the nature of the failure, to achieve exactly-once execution wherever possible. Summarizing, transactions offer:

- Atomicity: Either the whole transaction is executed (when it successfully commits) or none of it is done (when it aborts), often referred to as the all or nothing property.
- Consistency: Every transaction starts from a consistent view of the state and leaves the system in another consistent state, provided that the transactions would do so if executed sequentially.
- Isolation: Individual memory updates within an ongoing transaction are not visible outside the transaction. When the transaction commits, all memory updates are made visible to the rest of the system.

A transaction typically works in three phases:

- Start: This event may have different purposes depending on the TM implementation. A common operation is to set up data structures that are used later on for bookkeeping.
- Accessing data: During the transaction itself, accesses performed to shared data may have to resort to the TM system. Some systems may be completely transparent, as in the case of Hardware Transactional Memories, whereas others may require the use of explicit calls to the TM in use. In any case, the TM ensures that writes are registered (in the write-set) and reads are consistent (and possibly also registered in the read-set).
- Commit: Attempts to consolidate the tentative changes, recorded during the accessing phase, making them globally visible. Depending on the system, the writes may already be in-place or still be in buffers before this phase. This operation may fail in which case it discards all its tentative changes.

Additionally, two transactions are said to conflict if there is no equivalent sequential execution ordering of the two transactions that explains the result of each individual operation that is part of the transactions. At a lower level, conflicts may be detected in different ways depending on the TM characteristics. For instance, if two operations belonging to different transactions access the same base object and at least one of them is a write, this may be seen as a conflict. Another way is to ensure that a transaction's read set is disjoint from concurrent transactions' write sets.

In this paradigm, the programmer is responsible for identifying code whose result in the system must be seen as taking effect all at once. This boils down to marking blocks or methods with some artefact that identifies that piece of code as an atomic action. Depending on the implementation, the programmer may be left with a more burdensome task of starting and committing transactions that encapsulate the atomic actions.

In Listing 1, we show an example in which a university course is represented with a maximum capacity and current enrolled students. The mutable shared state is the list of students that may be modified concurrently by multiple users enrolling in the same course. The lack of synchronization in the concurrent manipulation of the shared structure could lead to the loss of enrollments or exceeding the maximum capacity. Therefore, we have identified the `enrollStudent` method with the `@Atomic` annotation that demonstrates a possible way of indicating to the TM system which methods must be

Listing 1. Concurrent object representing a course whose enrollments are protected with transactions.

```
class Course {
  final int capacity;
  List<Student> enrolledStudents;
  @Atomic boolean enrollStudent(Student std) {
    if (enrolledStudents.size() < capacity) {
      enrolledStudents.add(std);
      return true;
    } else {
      return false;
    }
  }
}
```

ran transactionally. The enrollment of a student is now seen as an indivisible operation so that other threads cannot see any intermediate state of the operation.

Building on the same example, Listing 2 shows the enrollment in multiple courses, which allows the student to build his own schedule for the semester with an all-or-nothing semantic: If one of the courses is full, he will probably need to pick an alternative and rethink the schedule as a whole. Consequently, we identified the `enrollMultipleCourses` as an atomic action, which is successful only if all the individual enrollments succeed. More importantly, deciding upon `enrollMultipleCourses`'s atomicity did not interfere with the previous decision regarding `enrollStudent`. It is in this sense that transactions compose: The programmer need not know the internals of the method `enrollStudent` being called. In practice, when `enrollStudent` is called from within `enrollMultipleCourses`, a nested transaction may be created, among other alternatives explored in Sections 2.2. and 4..

The initial proposal for transactional memory introduced it as an abstraction that programmers could use for lock-free synchronization in their applications [28]. In that work, the authors presented designs for extensions to multiprocessors' cache coherence protocols. Later, Shavit and Touitou evolved the same concept solely to software in [48]. Yet, it was very restrictive as the programmer had to identify static transactions, that is, transactions that access a pre-determined sequence of locations. These issues were first overcome in DSTM [1], a Software TM providing a slightly more relaxed guarantee, obstruction-freedom, which will be described in Section 4.3.1..

Listing 2. Representation of an enrollment in multiple courses as an atomic action.

```
@Atomic void enrollMultipleCourses(
      Student std, List courses) {
    for (Course course : courses) {
      if (!course.enrollStudent(std)) {
        STM.abort();
      }
    }
}
```

A lot of promising work has been delivered on software implementations of Transactional Memory (STMs). Although hardware implementations are more efficient, its practicality is far more complicated. As we shall see, most of the work regarding nesting models has been performed on STMs. Consequently, that is where we turn my attention to in this work.

In fact, there has been some criticism regarding STM performance even if considering some implementations representing the state of the art [54]. As a matter of fact, studies have been carried to assess both the pure performance of STMs [29] as well as its application in real projects development [30, 53]. The authors claim to have reached positive conclusions regarding the advantages of transactional memory over mutual exclusion synchronization to deal with concurrency.

2.2. Composability and Concurrency

In this Section we present two different aspects that may influence transactions: Composability between transactions and exploiting concurrency inside transactions. Although it may seem that both aspects are independent from each other, we show how parallel nesting requires both of them to be taken into account.

The best practices in software development encourage programmers to modularize their code and abide by well defined interfaces [55]. As we have seen, this explicitly contradicts lock-based concurrency, in which the programmer has to be aware of modules' internal locking conventions. However, that is not the case for transactions that take into consideration composability.

Suppose that some application code enclosed in a transaction calls a library function. It is perfectly acceptable that the library itself uses transactions to protect its shared internal state from concurrent calls. In such a scenario, a transaction created in the library code would be nested in the outer application transaction, i.e., it would compose. Nested transactions are an extension of a basic transaction structure to multi-level structures: Sets of sub-transactions that may recursively contain other sub-transactions, thus forming a transactional tree. They have been long used in the database world, from which many concepts have been adapted to transactional memory systems [27].

For instance, take in consideration the kind of monolithic transactions typical in enterprise applications with large volume of work. If such a transaction aborts after having performed most of its content, there is a considerable penalty in performance due to the rollback and its repetition. In [9], the `orElse` construct was proposed to compose transactions as alternatives: Given two transactions, if the first alternative aborts, the second alternative is attempted as a way of avoiding a complete abort. Consider the example in Listing 3: The `orElse` construct allows the programmer to specify that if the multiple enrollment fails then the student schedule should be enqueued so that any drop outs may trigger another action to enroll students that were pending.

Breaking transactions such as `enrollMultipleCourses` down to smaller nested transactions preserves transactional memory semantics while allowing each new portion of work to be attempted, and possibly fail, without necessarily aborting all the work accomplished so far under the same top-level transaction. The fact that the changes performed by each alternative are isolated allows the TM system to discard the changes when moving from one alternative to the other.

On the other hand, atomic actions enclosed in transactions are still eligible targets of parallelization. Recalling the example provided before, suppose now that a new module keeps statistics about the enrollments: Given the anticipated high number of queries, the statistics are rebuilt upon each change in the state, being therefore cached to answer the queries right away, rather than being built on every query from some consistent view of the state. To fulfill this requirement we introduced a call to the

Listing 3. Changes to the multiple enrollment to allow an alternative action using the `orElse` construct if some course is full.

```
class CourseEnrollment {
  Map<Student, List<Course>> pendingEnrollments;

  @Atomic void enrollMultipleCourses(
        Student std, List<Course> courses) {
    orElse(attemptCourses(std, courses),
          enqueuePending(std, courses));
  }

  @Atomic void attemptCourses(
        Student std, List<Course> courses) {
    for (Course course : courses) {
      if (!course.enrollStudent(std)) { STM.abort(); }
    }
  }

  @Atomic void enqueuePending(
        Student std, List<Course> courses) {
    pendingEnrollments.put(std, courses);
  }
}
```

statistics module when the `enrollMultipleCourses` finishes successfully as shown in Listing 4.

Moreover, the statistics module will be internally sorting some shared data structure so that the users may know which courses have more students. What matters, however, is that the sort uses parallel units of work to speed up the operation (hinted by the `@Parallel` annotation to simplify the example). So far there has been an implicit coupling between a transaction and the thread in which it is running. In this setting, however, what is supposed to happen to the relation between the transaction that is running and the threads that execute the parallel sort in that transactional context? The alternatives range from having the threads running in the same transaction; running in parallel sibling nested transactions; or even escaping the transactional context and having no relation at all. It is not clear which solution is better, given that it is very likely that one can come up with use cases for each of them.

For instance, in the previous sorting parallelization example, the concurrent threads would not have any intersection among their read and write sets, meaning that their accesses would be disjoint from each other regarding the elements to sort. As a result, that setting would benefit more from having the threads running simultaneously in the same transaction (the one in which scope they had been created).

Conversely, if the parallelization was not guaranteed to be conflict-free, each thread would have to run in a nested transaction to allow detection and resolution of conflicts. This means that the set of threads would be associated to a set of parallel sibling nested transactions with a common parent transaction (the one in which scope the threads and transactions had been created).

Listing 4. Enhancement to the multiple enrollment that exposes transaction composition with task parallelism.

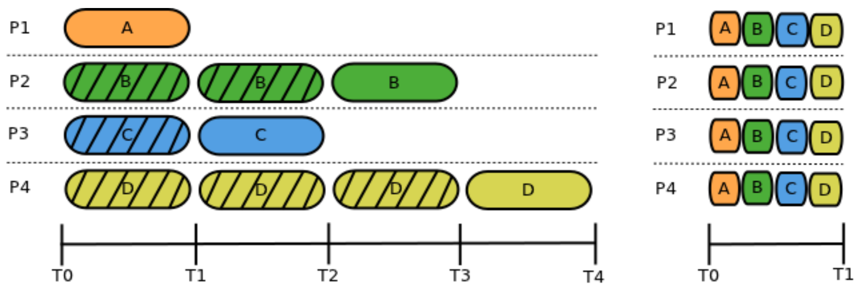
```
class CourseEnrollment {
  @Atomic void enrollMultipleCourses(
        Student std, List<Course> courses) {
    for (Course course : courses) {
      if (!course.enrollStudent(std)) {
        STM.abort();
      }
    }
    CourseStatistics.rebuildTopEnrollments();
  }
}

class CourseStatistics {
  @Atomic void rebuildTopEnrollments() {
    ...
    sort(...);
    ...
  }

  @Atomic @Parallel void <T> sort(
        List<T> elements, Comparator<T> cmp) {
     ...
  }
}
```

In workloads with some write dominance, if there is contention on some shared state, then inevitably the optimistic concurrency control approach does not result in profit and the critical path of some sequential execution becomes the bottleneck. Fig. 1 shows the execution of four transactions labelled from `A` to `D` in four processors labelled from 1 to 4. If these transactions conflict with each other as described (and shown in Fig. 1(a)), then the execution time ends up being the same as if they were executed in a single processor. On the other hand, if each transaction was parallelizable (in four independent, non-conflicting tasks), they could still be ran each at a time, but a lot faster as we may see on Fig. 1(b). Once again, parallel nested transactions are crucial to achieve this result if the parallelization does not guarantee that each part of the transaction being split is completely independent from its counter parts.

We have briefly described two distinct features in transactions, composability and concurrency, whose intersection is required by nested parallel transactions that we will explore in my work. As a matter of fact, over the last decade, researchers proposed many different designs and implementations for TM systems, steadily improving their performance and flexibility for a large variety of languages. Yet, despite all of this work, support for parallel nested transactions has been almost neglected. Recent work has suggested that providing parallel nesting in TM implementations allows exploring more concurrency and consequently obtaining better performance [52]. We survey the literature that takes this in consideration in Section 4.5..



(a) With conflicts (b) Conflict-free

Figure 1. Execution of four transactions (`A` to `D`) in four processors. Dashed transactions are aborted due to conflicts whereas non-dashed commit successfully.

2.3. Goals of this Work

The main goal of this work is to advance the state-of-the-art in STM research, by designing and implementing a new STM model that supports parallel nesting with unlimited depth, and without incurring into excessive overheads.

The starting point for this work is the current design of a lock-free multi-version STM and its implementation, the JVSTM [12], which has support for linear nesting only. Extending the JVSTM to support parallel nesting will require the following tasks:

- Explore the various nesting models that have been proposed in the literature.
- Extend the multi-version STM model of the JVSTM to support parallel nesting in an efficient way (ideally, preserving the lock-free property of the JVSTM).
- Extend the JVSTM API to integrate the creation of threads and transactions seamlessly without imposing many restrictions or difficulties on the programmer.
- Implement the new model of parallel nesting on the JVSTM.
- Evaluate its performance on a set of representative benchmarks.

As a consequence, the JVSTM will become more flexible to deal with different usages and workloads.

2.4. Why is this difficult?

As we shall see in the existing systems present in this document, there have been few TM implementations that address parallel nesting. So far, we have described why parallel nesting should be taken into consideration: The promise of unveiling more concurrency in scenarios where that may lead to an increase in performance is tempting. But if that is the case, then what exactly has been delaying the usage of nested parallel transactions in practice?

Providing parallel nested transactions entails not only the challenges of nesting but also the need to make sure parallel siblings synchronize their actions when necessary. On the first case, there is a concern regarding additional work that has to be performed in the transactional operations (such as accesses and commit) proportionally to the depth of nesting. The issue about the synchronization of siblings is, of course, an additional source of overheads that may entail significant costs on the use of parallel nested transactions.

The actual challenges shall be clear as we present the TM implementations that provide parallel nesting in Section 4.5.. Above all, the most important point to retain, beyond the particularities of what makes it hard, is that parallel nested transactions should provide a performance benefit. When the programmer takes specific care to identify parts of the program to parallelize, he is expecting to gain speedup in its execution. Therefore the difficulty is in providing a design and implementation of parallel nested transactions in which executing concurrent parts of an atomic block of the application does not end up being more costly than executing them one at a time sequentially.

3. Transactional Memory Theory and Guarantees

The intense research on TMs has resulted in both practical implementations and theoretical assertions regarding TM whose formal properties have been refined along the time. In this section we begin by presenting work that defines the correctness criteria and progress guarantees upon which the implementations are built. In particular, some of these concepts are referred later to characterize both existing work and the my proposed solution.

3.1. Correctness Criteria

From a user's perspective, a TM should provide semantics similar to the ones of critical sections: transactions should appear as if they were executed sequentially. Yet, a TM implementation would be inefficient if it never allowed different transactions to run concurrently. Reasoning about the correctness of a TM implementation implies stating if a given concurrent execution respects that correctness criterion.

Linearizability [49] was initially proposed as a safety property devised for concurrent objects. In the context of TM, linearizability means that, every transaction should appear as if it took place at some single, unique point in time during its lifespan. Although it has been used for reasoning about TM correctness, linearizability does not entirely suffice as an appropriate correctness criterion for TM. Note that a TM transaction is not a black box operation on some complex shared object but an internal part of an application: The result of every operation performed inside a transaction is important and accessible to a user. Therefore it is also important to define what exactly happens in each operation of a transaction. Yet, linearizability only accounts for the execution of the transaction as a whole.

On the other hand, serializability [26], which originated in the database transactions, states that the result of a history of transactions is serializable if all committed transactions in it receive the same responses as if they were executed in some serial order, i.e., without concurrency between transactions. Usually, it is said that such a serialization explains the concurrent sequence of operations of that execution. However, serializability does not state any behavior regarding accesses performed by live transactions (more specifically, about transactions that may eventually abort). As we will see, such accesses may render harmful if the correctness criterion does not safely prevent them from returning erroneous results.

In [31], Guerraoui and Kapalka argue that these previously described correctness criteria used for other purposes (databases, concurrent objects, etc...) do not fit the needs for TM. In particular, none of them captures exactly the very requirement that every transaction, including not yet completed ones, accesses a consistent state, i.e., a state produced by a sequence of previously committed transactions. Whereas a live transaction that accesses an inconsistent state can be rendered harmless in database systems simply by being aborted, such a transaction might create significant dangers when executed within a general TM framework.

Suppose that in some program there are two shared variables x and y related by the invariant $x < y$.

Listing 5. Example of code that may not be execute properly due to the lack of an appropriate correctness criterion.

```
int lowBound = x, upBound = y;
for(; lowBound < upBound; lowBound++)
    array[lowBound] = lowBound;
```

Consider the example shown in Fig. 5. Assuming that initially $x = 5$ and $y = 10$. The steps of two concurrent transactions are represented by: $R_{T_1}(x, 5)$; $W_{T_2}(x, 0)$; $W_{T_2}(y, 4)$; $C_{T_2}(ok)$; $R_{T_1}(y, 4)$, where $W_t(x, y)$ means that transaction t writes the value y to the transactional variable x, $R_t(x, y)$ means that transaction t reads the transactional variable x and finds the value y and $C_t(ok|fail)$ means that transaction t attempted to committed and either succeeded or failed. Then, transaction T_1 read some inconsistent state where the invariant $x < y$ is not respected. Taking in consideration the example, the lower bound of the cycle ends up being greater than the upper bound limit. Depending on the execution environment, the consequences may vary, but nevertheless are not acceptable.

To eliminate this problem, opacity is proposed as a correctness criterion that requires (1) transactions that commit to look as if they executed sequentially (equivalent to serializability); (2) aborted transactions must also be given the illusion of no concurrency, i.e., they must also observe consistent states all the time; and (3) operations executed by an aborted transaction must not be visible to any other transaction. The solution that we shall propose will abide the opacity criterion to avoid the hazards described.

3.2. Progress Guarantees

The correctness criteria presented above may be trivially achieved in a TM implementation that aborts every transaction. Despite its uselessness, it motivates the formalization of progress conditions that capture the scenarios in which a transaction must commit or may be aborted. Consider, for example, a simple progress condition that requires a transaction to commit if it does not overlap with any other transaction. Such condition can be implemented using a single lock that is acquired at the beginning of a transaction and released at its end. As a result, transactions will be running one at a time, thus ignoring the potential benefits of multiprocessing yielding zero concurrency [34]. Therefore the objective is to have positive concurrency in which TM implementations allow at least some transactions to make progress concurrently.

3.2.1. Operation level liveness

To achieve progress at the level of transactions, it is important to formalize which guarantees should be provided at the level of operations that constitute a transaction. In [25], Herlihy and Shavit present these guarantees which we briefly summarize next.

A blocking synchronization technique, in which an unexpected delay by one thread can prevent others from making progress, provides starvation-freedom if all threads eventually progress when trying to grab some lock. More relaxed, deadlock-freedom only requires that some thread manages to grab the lock and consequently it may happen that a specific thread never manages to do so.

On the other hand, a non-blocking synchronization technique is said to be wait-free if it ensures that every thread finishes its task in a finite number of steps even if it faces arbitrary delays of concurrent threads. Such events may take place due to blocking for Input/Output or adverse scheduling by the operating system. Lock-freedom only ensures that the system as a whole makes progress meaning that a specific thread may never make progress in the face of concurrent threads progressing.

Yet, there is a midterm guarantee: Obstruction-freedom guarantees that one thread makes progresses if it executes in isolation for sufficient time: a transaction T_k executed by thread p_i can only be forcefully aborted if some thread other than p_i executed a step (low level operation) concurrently to T_k. Although it was initially presented as a synchronization mechanism [56], obstruction-freedom has also been used to classify the progress guarantees of a TM system [1, 33]. Formally, if fits in the zero-concurrency category as a transaction is guaranteed to commit only if it faces no contention (theoretically allowing no concurrency).

Like stronger non-blocking progress conditions such as lock-freedom and wait-freedom, obstruction-freedom ensures that a halted thread cannot prevent other threads from making progress. Unlike lock-freedom, obstruction-freedom does not rule out livelock: interfering concurrent threads may repeatedly prevent one another from making progress. Compared to lock-freedom, obstruction-freedom admits substantially simpler implementations that are more efficient in the absence of synchronization conflicts among concurrent threads.

To cope with the possibility of livelock, Herlihy et al [1] proposed that modularized mechanisms could be used to enforce a given policy that seeks to avoid livelocks: contention managers. These may be queried to decide if a transaction is allowed to abort another one or if it should abort instead. We briefly describe some possible policies in Section 4.1.6..

3.2.2. Progressiveness

Some of the most efficient TM implementations internally resort to locking despite providing a lock-free illusion to the programmer that uses the TM system. To capture the guarantees provided by these TMs, progressiveness was proposed [36], in which a transaction encountering no conflicts must always commit. This property is common to all the following variants:

- Single-lock progressiveness: a transaction can abort only if there is a concurrent transaction. One TM implementation providing this guarantee has been briefly addressed above (using a global lock).
- Weak progressiveness: a transaction can abort only if a conflict with a concurrent transaction arises in an access.
- Strong progressiveness: stronger than weakly progressive as it requires that, among a group of transactions whose accesses conflict on a transactional variable, at least one of them commits.

3.2.3. Permissiveness

A TM is permissive with regard to a correctness criterion C (where C may be opacity for example) if it never aborts a transaction unless necessary for maintaining safety according to that criterion. Note that a TM may be seen as an online algorithm in the sense that, on each operation that it executes, it has to decide on its influence on the overall correctness of an incomplete transaction with operations that may yet be performed. Consequently, ensuring C-permissiveness may yield a very expensive algorithm complexity wise. As a matter of fact, it has been shown that it is impractical to achieve permissiveness deterministically [37].

Therefore, an alternative notion has been suggested in the literature: probabilistic C-permissiveness [37] in which some randomization takes place that eventually leads to the acceptance of C-safe histories by the TM. The underlying idea builds on the following example: if T_k and T_i access the same transactional variable where T_k writes and T_i reads, even if T_k commits first (but after the concurrent read took place), T_i may still commit if its serialization point is before T_k's. For this to be possible, transactions may adaptively validate themselves by maintaining a possible interval of serialization. At commit time, they randomly choose a point within that interval to serialize themselves, allowing transactions to commit probabilistically in the past, or in the future.

On a slightly different setting one may also use multi-version-permissiveness: a relaxation in which only read-write transactions may abort and in which case it has to conflict with a concurrent read-write transaction [32]. Therefore, read-only transactions must always commit. This guarantee suites TMs that maintain multiple versions of transactional variables. However, it has been shown that single version TMs may also be mv-permissive [57].

4. Existing Systems regarding Nesting

In this Section we begin by describing several design choices have been discussed in the related work. It is important to understand these as they are the building bricks of a TM. Following, we address nesting in detail. Firstly, achieved by nesting transactions to provide composability. Secondly, linear nesting and finally parallel nesting. Orthogonally, a nested transaction is either closed or open, which we explain when presenting linear nesting.

4.1. Transactional memory design choices

Among the TM systems that have been proposed, many different design decisions were promoted. To start with, a TM can be implemented either in hardware or software, as well as in various hybrid approaches that dynamically switch between hardware and software execution modes. However, they all have common issues that have been solved very differently across the literature. In the next Sections we present those characteristics. We also summarize several TM implementations by characterizing them according to these design choices in Fig. 2.

4.1.1. Update Policy

The update policy establishes how the system manages both stable (valid when the transaction had started) and speculative values (attempting commit) of the transactional shared variables. The former ones are used when the transaction aborts whereas the latter ones are used in case it commits.

One strategy is called lazy (also known as deferred) update, in which all writes performed within a transaction are buffered until commit time. These writes may be stored as values in a list or applied to some tentative copy of an object. Upon commit, these buffered writes are publicized, meaning that they are written to the proper address corresponding to the transactional value. Conversely, if the transaction aborts, it suffices to discard the local tentative writes.

On the other hand, there are eager (also known as direct) updates that are directly applied to the transactional variable/object instead of some shadow copy or temporary buffer. To return the global state to a consistent one when a transaction aborts, these writes must still be logged in what is usually referred to as an undo log. However in this case, the log will contain the previous values that were overwritten by the tentative values. This way, upon abort, they may be retrieved and rewritten in the global state while at commit time it suffices to clean the undo log.

System	Granularity	Conflict Detection	Update Strategy	Synch Strategy	Nesting models	Isolation	Read Visibility
DSTM [1]	Object	Eager	Direct	O-F	Flat	Weak	Invisible
OSTM [38]	Object	Lazy	Deferred	L-F	None	Weak	Invisible
ASTM [21]	Object	Mixed	Deferred	O-F	None	Weak	Invisible
RSTM [20]	Object	Mixed	Deferred	O-F	Flat	Weak	Visible
McRT-STM [22]	Obj or CL	Mixed	Direct	L-B	Closed	Weak	Visible
TL2 [19]	W or Obj	Mixed	Deferred	L-B	None	Weak	Invisible
LSA [18]	W or Obj	Eager	Deferred	O-F	None	Weak	Invisible
DracoSTM [39]	Object	Mixed	Both	L-B	Closed	Weak	Visible
TinySTM [17]	Word	Eager	Both	L-B	None	Weak	Invisible
SwissTM [16]	Word	Mixed	Deferred	L-B	None	Weak	Invisible
StrAtomic [40]	Object	Mixed	Direct	L-B	Closed	Strong	Invisible
Elastic-STM [7]	Word	Eager	Deferred	L-B	Closed	Weak	Invisible
NoRec [23]	Word	Lazy	Deferred	L-F	Closed	Weak	Invisible
JVSTM [12]	Word	Lazy	Deferred	L-F	Closed	Strong	Invisible
LockBased [15]	Object	Mixed	Deferred	L-B	None	Weak	Visible
CWSTM [2]	Word	Eager	Direct	L-B	Parallel	Weak	Visible
NeSTM [3]	Word	Eager	Direct	L-B	Parallel	Weak	Invisible
HparSTM [5]	Object	Mixed	Deferred	L-B	Parallel	Weak	Visible
NePalTM [6]	Word	Eager	Direct	L-B	Parallel	Weak	Both
PNSTM [52]	Word	Eager	Direct	L-B	Parallel	Weak	Visible
SSTM [13]	Word	Lazy	Deferred	L-B	Parallel	Weak	Invisible

Figure 2. Characterization of several STMs according to their properties. O-F stands for Obstruction-Free, L-F for Lock-free, and L-B for Lock-based synchronization strategies. Also, CL stands for Cache Line granularity level.

4.1.2. Conflict detection and resolution

To detect conflicts, each transaction needs to keep track of its read- and write-sets. On one hand, lazy conflict detection and resolution (also referred to as late, optimistic, or commit-time) is based on the principle that the system detects conflicts when a transaction tries to commit, i.e., the conflict itself and its detection occur at different points in time. To do so, one possibility is to have a committing transaction T_i to ensure that no write-set of a recently committed transaction T_k intersects with T_i's read-set. A transaction T_k is recently committed with regard to T_i if it committed after T_i started and before T_i committed. If there is an intersection, there is a read-write conflict, which leads the transaction attempting commit to abort. This strategy promotes more concurrency (by causing less conflicts) because a read-write conflict may not be troublesome if the reader transaction commits before the writer. However, conflicts are detected late, which may result in fruitless computation.

On the other hand, eager conflict detection and resolution is based on the principle that the system checks for conflicts during each load and store, i.e., the system detects a conflict directly when it occurs. As we describe in Section 4.3.2., there have been attempts to combine eager write-write detection and lazy read-write, in what is named mixed invalidation scheme.

4.1.3. Granularity of conflict detection

Despite how conflicts are managed, one may perform the conflict detection at different granularities. The most fine-grained one registers memory addresses in the read and write-sets and performs verifications by comparing the memory words. The major drawback is the extensive overhead regarding the fine-grained mapping that is created to cover all the addresses. An advantage is that this strategy avoids false sharing.

A midterm alternative requires less time and space, by using cache line granularity, but risks having false sharing which may lead to unnecessary aborts. This strategy is usually applied to HTM systems that can leverage on existing coherence mechanisms.

Finally, there is an alternative that promotes object granularity, in which the sets maintain the objects whose field(s) are read or written. This strategy may also yield false sharing if two concurrent transactions access different fields in the same object. Note that there is room for different granularities besides the ones explicit as in practice the conflict detection may employ more than one memory word or cache line.

4.1.4. Types of atomicity

In most STM implementations, there is typically an API that the programmer uses to manipulate transactional data. However, the truth is that (usually) there is nothing stopping him from directly accessing the data without the mediation of the transactional system. This may happen because the access did not follow the aforementioned convention or because it was performed outside a transaction. This leads to another design decision of whether non-transactional code can read non-committed updated values within an ongoing transaction.

Weak atomicity consists of non-transactional code that can read non-committed updates, while the opposing strong atomicity refers to the impossibility of performing that action. It is difficult to argue against strong atomicity from the semantic point of view that is provided to the programmer. However, in practice, it is largely more difficult to implement efficiently. As a consequence, most of the STMs presented in Fig. 2 provide weak atomicity. The drawback is that the programmer is allowed to do what is most likely a mistake and access shared data outside a transaction possibly causing a race condition.

4.1.5. Read visibility

Some STM systems perform an optimization regarding transactional read accesses in which the transaction records the read in its local storage, but does not incur in any synchronization action to do so. The downside of this lightweight read operation, also known as an invisible read, is the following: if T_i performs an invisible read on variable x and later T_k writes to x and commits before T_i does so, then T_i will be doomed to abort without having such knowledge. More importantly, T_i may now access some other variable y that was also written by T_k. This way T_i would had read an inconsistent state that was no longer serializable. The traditional solution is for T_i to validate its read-set every time it performs a new read, i.e., ensuring that no variable read has been overwritten since it was read. This solution is avoided in multi-version STMs such as JVSTM. Conversely, a visible reader would had left some trace of its access in x that would had been used by T_k to abort T_i (also known as invalidation).

4.1.6. Contention Managers

In the previous section when a conflict was detected, the solution used was always to abort the transaction that detected the conflict. However, that decision can be taken either way or may not require a transaction to abort at all. A contention manager typically implements one or several contention management policies to decide which transaction should win a conflict. This concept was initially suggested in [1] as a modularized policy that could be adapted accordingly to the expected workload, as there is not a single policy to rule them all. Some existing variants are as follows:

- Greedy: Guarantees that each transaction commits within a finite or bounded time.
- Karma: Considers a rough measure of the computation that the conflicting transactions have done so far. The one with the least amount of work done is aborted.
- Polite: Uses an exponential back-off strategy to resolve the conflict. When a transaction unsuccessfully has tried to commit a specific number of times, the contention manager aborts the competing transaction(s).
- Polka: Backs-off for different intervals proportional to the difference in priorities between the transaction and its enemy.
- Time-based: Records the start time of a transaction and, in case of a conflict, it aborts the transaction(s) with the newest time stamps.
- Timid: Always aborts a transaction whenever a conflict occurs.

4.2. Disjoint-Access Parallelism

Consider the transactions T_i and T_k whose executions only require access respectively to the transactional variables x and y. Conceptually, both transactions should progress independently without contending to any common data. This property is named disjoint-access parallelism (DAP) and formally requires that operations on disconnected data should not interfere with each other (namely, by contending on data structures of the TM).

This concept is easily visualized in a graph representing conflicts of transactions that overlap in time. Informally, the vertices of the conflict graph correspond to data items and there is an edge between data items if they are accessed by the same transaction. Two transactions T_i and T_k are disjoint-access if there is no path between an item in the data set of T_i and an item in the data set of T_k, in the conflict graph of the minimal execution interval containing the intervals of both transactions. The concurrent transactions are said to contend on a base object if they both access it and at least one of them performs a write. Therefore, a TM is weakly DAP if there exists no pair of concurrent transactions, such that they are not disjoint-access, contending on the same base object [35]. This relaxation allows transactions to perform reads on the same base object even if they are disjoint-access.

This property is crucial for the scalability of a TM: when it is not ensured, as the number of concurrent threads increases, more contention will follow on the artificial hot spots induced by the TM algorithm.

For instance, the JVSTM [12] requires write-transactions to increment a global counter on commit. That counter represents an artificial hot spot of contention and consequently the JVSTM is not disjoint-access parallel. Note that in practice weak DAP does not hinder scalability as the read operation will not require the accessed data to be held in exclusive mode in the processors' private cache.

4.3. Nesting by flattening

As we have seen, nesting of transactions is a requirement to support software composability. The simplest way to provide it is by flattening transactions into the outermost level. Yet, some implementations have failed to support even this model [38, 21, 19, 18, 17, 15, 16].

A possible implementation of such model is for a transaction to maintain a counter regarding the depth of nesting. This way, instead of creating a nested transaction, the counter is incremented. When a commit is reached, the counter is decremented. The actual commit is only performed when the counter corresponds to the top level.

In this setting, the code that conceptually belongs to a nested transaction is actually behaving as if it were in the top-level transaction. Therefore, all the bookkeeping performed during the accesses is maintained in the top-level transaction's structures. Next, we present some implementations that have addressed nesting by flattening the transactions.

4.3.1. DSTM

The Dynamic Software Transactional Memory (DSTM [1]) overcame the deficiency of previous STM systems [28, 48] where the transaction size and memory requirements were statically defined in advance. DSTM provides C++ and Java APIs for programming dynamic data structures, such as lists and trees, for synchronized applications without locks. It employs non-blocking synchronization (obstruction-freedom), in-place update and eager conflict detection at an object level. The concept of contention manager was also motivated by their obstruction-free design, as they claimed that the livelock-freedom guarantee should be provided by a modularized manager queried upon conflict for its resolution.

DSTM simplified composability of transactions by flattening nested transactions in the outer transaction as previously described. It also provided a novel and powerful (albeit dangerous) way to reduce conflicts. Before it commits, a transaction may release objects that it has read, effectively removing them from its read-set. Once an object has been released, other transactions accessing that object do not conflict with the releasing transaction over the released object. The programmer must ensure that subsequent changes by other transactions to released objects will not violate the linearizability of the releasing transaction. Therefore, a transaction may observe inconsistent state. Clearly, the release facility must be used with care; careless use may violate transaction linearizability. This mechanism partially resembles open nested transactions that we describe in Section 4.4..

4.3.2. RSTM

The Rochester Software Transactional Memory (RSTM [20]) developed an obstruction-free STM because lock-based, although performing better in some cases, is still vulnerable to priority inversion, thread failure, convoying, preemption, and page faults. It is another example of an STM that provides only flattened transactions to support nesting. The RSTM design also used object-based conflict detection because it fits better in STMs: rather than instrumenting all memory accesses, it suffices to provide an API for the programmer to open transactional wrappers on shared objects (similar to DSTM). The authors argue that write-write conflicts should be detected eagerly because neither of them will be able to commit for sure. However, read-write conflicts should be detected lazily as they may still be innocuous in case the reader transaction commits first. This is what they named a mixed invalidation strategy.

Visible readers are used to avoid the aggregate quadratic cost of incrementally validating invisible reads private to transactions on every access. Consequently, a writer aborts all visible readers before acquiring an object.

4.4. Linear Nesting

Almost all recent related work builds on the model presented by Moss and Hosking [42]. The linear nesting model imposes that a transaction may have only one nested transaction active at a given time. Conversely to flattening, an atomic action enclosed in the control flow of an active transaction T_i will effectively create a nested transaction T_k. The parent of T_k is T_i. The definition is recursive in the sense that a nested transaction is merely a more specific term for a general transaction. Therefore, a nested transaction may also be the parent of another nested transaction. A top-level transaction may now be easily defined as a transaction without a parent. The ancestor set of a top-level transaction is empty whereas the ancestor set of a nested transaction T_k is its parent T_i plus T_i's ancestor set. Moreover, when T_k, a linear nested transaction, attempts to read variable x, it must obtain the value it previously wrote to x. If T_k never wrote to x, then it does the same but on its parent T_i instead. If T_k has no parent, and thus is top-level, it obtains the globally known value.

As we shall see, we may define two types of nested transactions [42]: Closed and open. However, regardless of its type, a nested transaction accessing a variable will always obtain the most recent value known by itself (in case it has written to it) or by its ancestors. What differs between closed and open nested transactions is what happens when they attempt to commit.

On one hand, a closed nested transaction commit results in the union of its read- and write-sets with its parent's. This type of nesting is provided in [23, 8, 22]. Open nested transactions (ONTs), provided in [11, 4], allow a committing inner transaction to release isolation immediately: The commit is partially performed as if it was a top-level transaction. This means that the writes are made globally visible and both its read and write sets are discarded. However, one of its ancestors may yet abort, in which case the ONT's write set "escaped" the control of the abort mechanism and was made visible incorrectly. The workaround depends on the TM design. One example may require the nested transaction to propagate an undo-set to its parent [41].

It is not straightforward how some applications can deal with ONTs [4]. The issue lies in the fact that opacity is broken since some other transaction may read a value publicized by a transaction that aborts. Moreover, it is also difficult to deal with states in which an open nested transaction commits values that depend on some state that was written by one of its ancestors but is not yet committed. For instance, consider that $y = 2 * x$ where x and y are shared variables in our program. Consider the history:

$W_{T_1}(x, 1); S_{T_1}(T_2); R_{T_2}(x, 1); W_{T_2}(y, 2); C_{T_2}(ok)$

where we add $S_t(t_1, t_2, ..., t_n)$ to the notation meaning that transaction t spawns the nested transactions $t_1, t_2, ..., t_n$. Moreover, T_2 is an open nested transaction and variables are initialized to 0. By the moment that T_2 commits, any concurrent transaction that reads x and y will see an inconsistent state where $y = 2$ and $x = 0$ until T_1 commits. This leakage of uncommitted state is traditionally avoided by the following rule of thumb: ONTs' footprint (i.e., the union of read- and write-sets) should not intersect with its ancestors' footprint. This approach was proposed as a way to increase concurrency and decrease false conflicts (in the sense that to the application logic those conflicts were not relevant) [8]. The DSTM's eager release mechanism presented in Section 4.3. is another way of achieving a similar goal.

A common motivation for open nested transactions is based on part of an atomic action that has a very high chance of conflicting with concurrent transactions. For instance, if all transactions in some program have to increment some statistical counters, this indirectly causes one transaction only to succeed, which is the one that manages to commit without its accesses being invalidated. Therefore the usual proposal is that the changes on the counters are encapsulated in an open nested transaction.

Next, we describe some STMs that use closed/open linear nested transactions.

4.4.1. Haskell STM

More than performance, the main goal of the Haskell STM [9] design was to explore the type system of the Haskell language to ensure the safety of the operations executing within a transaction, ruling out operations with side-effects. In part, it is the Haskell type system that allows some particularly interesting features that distinguished this work from its counter-parts.

More than performance, the main goal of the Haskell STM design was to explore the type system of the Haskell language to ensure the safety of the operations executing within a transaction, ruling out operations with side-effects.

For once, the possibility of blocking, i.e., conditional signaling between threads for producer-consumer patterns, is supported with the `retry` construct. Its main idea is that the current transaction read-set must be, at least partially, representative of the condition that the thread is depending on. Therefore, the transaction is aborted but only retries once some transactional variable in its read set is written by a new commit.

Moreover, the construct `orElse` is provided to allow a failed transaction to recover in an alternative way: if the first atomic action aborts, the second one is attempted. This is repeated until nested `orElse` blocks have all been attempted or one of them committed. In the former case the whole enclosing transaction is retried. This construct motivated the application of a closed nesting model so that each `orElse` block corresponds to a new closed nested transaction.

4.4.2. McRT-STM

The Multicore Runtime STM [22] is implemented in both C++ and Java and is built on top of the McRT system as yet another component taking part in it. It uses closed nested transactions, employs a direct update strategy in combination with eager conflict detection for writes and late conflict detection for reads, and supports conflict detection at both cache line and object level. Moreover, it uses strict two phase locking and thus may incur in deadlock. Consequently, if a thread spins for too long to acquire a lock, it aborts and retries.

McRT-STM maps objects or cache lines to a set of locks. Its locking scheme uses multiple readers-single writer, i.e., the acquisition of a lock in read-mode prevents writers from modifying its content; its acquisition in write-mode disables readers and other writers from accessing the value.

They chose to make writes in-place for reducing the commit time that should be adequate when the fraction of aborting transactions is small. Moreover, the read-after-write is trivially solved: when a transaction accesses a variable for reading for which it has written a value, instead of seeking it in its write-set, it may immediately look up the variable as it previously directly wrote the new value there and not in some temporary location.

4.4.3. NORec

The No Ownership Records STM (NORec [23]) was specifically designed with a minimalistic intent to obtain the least overhead possible even if it entails the impossibility of scaling to hundreds of threads. It is a lock-free, word-based STM with lazy conflict detection and deferred write application. It also provides linear closed nested transactions for composability.

NORec uses a global logical clock for validation of its invisible reads by performing a read, a validation and re-read step to ensure it executed the process in a consistent manner without concurrent threads committing in the meantime. This step happens on every read access to ensure consistency across the transaction. Therefore, on commit, successfully validated transactions have to atomically increment that global clock and write-back its buffered values.

4.4.4. LogTM

The LogTM [8] authors emphasize that ideal software composition should not require programmers to have deep knowledge of modules' internals. Contrary to most of the previously described implementations, LogTM is an HTM system. The authors exploited a design with old value logging in thread local storage and in-place direct writing that was different from most HTMs at the moment [44].

The authors consider that aborts should be rare and thus they produce an hardware interrupt that is handled in software: this allows walking the undo log and replace the correct state of memory upon aborts.

LogTM provides both open and closed nested transactions. This is achieved by segmenting the undo log and replicating control bits for conflict detection in an activation record held in a stack such that each level is a nesting depth. Read and write sets are merged with the parent upon commit of a closed nested transaction. Open nesting works by removing the inner log segment upon commit and adding compensating action records to its parent log.

4.4.5. The Atomos Transactional Programming Language

Atomos [11] is an extension of Java with the traditional blocking synchronization mechanisms replaced with implicit transactions and strong atomicity. Atomic blocks delimit transactions whereas the `volatile` and `synchronized` modifiers are replaced by an `atomic` modifier that ensures that accesses to the corresponding field are encapsulated in a transaction. Beyond linear nesting composition of transactions, Atomos also allows open-nested transactions with the `open` keyword. Compensating actions are ran in transactional handlers that are hooked on key points of a transaction: commit or abort.

Another feature provides the `wait` keyword that allows to watch some condition and resume the transaction when it is verified that the condition has changed. The underlying scheduler retries only once it believes those conditions have changed due to some other commit. This assumes that the TM implementation has control over the runtime scheduler thread that should be always running and responsible for detecting changes. For that, inter-thread communication is performed using an open nested transaction as a means for escaping possible enclosing transactions.

4.4.6. Alternative approaches

The following transactional designs promote alternatives to open nesting transactions or seek the same objective: to reduce benign conflicts that frequently happen at operation level but have no influence in the semantics of the application. Recall the example that we have presented of incrementing some shared counter inside all transactions. Although all the transactions will conflict on the increment, that operation is actually commutative and the conflicts could be avoided. Note that an atomic increment operation is commutative with another atomic increment whereas the reads and writes that compose them are not.

Abstract Nested Transactions [14] (ANTs) were proposed to alleviate those benign conflicts. Its main idea is that some predictable contending block of code may be re-executed during the commit of its enclosing top-level transaction. This block of code is identified by the programmer as an ANT. Accesses performed within an ANT are stored separately from its enclosing transaction (similarly to traditional nested transactions). Upon commit, normal transactional accesses are validated, after which the correct usage of ANTs is verified: if some access in the ANT log intersects with its enclosing parent access log, then parent restarted and that nested transaction (once retried) will no longer execute as an ANT but rather as a normal closed transaction. If that case is not verified, then we're in the normal usage of ANTs. Then, if some ANT log entry fails validation, that ANT is restarted. If its re-execution produces a different return than what it had stored upon its original execution, the whole top-level transaction restarts.

Another approach are Elastic Transactions [7] that provide an insightful and common scenario with benign conflicts. Considering a set of sorted data elements, concurrently inserting two new elements in transactions may cause read-write conflicts. However, from a conceptual point of view, no conflicts may ever happen between those two operations. Therefore, the authors provide elastic transactions that may cut themselves and commit what has been done so far upon detecting a conflict. The rest of the transaction will itself form a new elastic transaction. The intuition is quite similar to the eager release mechanism provided by the DSTM.

On another setting, Transaction Communicators [10] are presented as variables through which concurrent transactions can communicate: Changes to a communicator by one transaction can be seen by other transactions before the first transaction commits. Such communication compromises isolation because the transaction may yet be aborted. The authors limit the influence of this compromise by preventing a transaction from committing unless every other transaction whose effects it has seen also commits. For example, two transactions that see the effects of each other must either both commit or abort. This communication is mediated through Communicator-Isolating Transactions (CIT). Normal transactions are flattened if nesting arises whilst CITs are implemented in nested transactions. The commit of a CIT actually resembles an ONT's: During the transaction its reads are invisible but writes acquire the ownership of the variables' locks (just like normal transactions); upon commit, those locks are released, effectively publicizing the writes to the communicators in concurrent CITs. The communicator variables undo log is merged in the enclosing (normal) parent transaction to enable rollback in case of abort.

Finally, Agrawal et al suggested another approach towards open nesting that safeguards against many of the pitfalls that result in unexpected behavior. Traditional solutions require some convention to be followed by the programmer of the application to ensure that objects accessed by an open nested transaction are disjoint from its parents' accesses. However, these ad-hoc solutions are inadequate. Thus, the authors created XModules [4], which is a framework that formalizes ONT semantics and its use. A concrete set of guidelines for data sharing and interactions between Xmodules is presented, which is enforced by a specified type system that could be applied as a language extension for some

Java-like language of choice. Given this setting, the authors are able to prove that if the proposed guidelines are followed, then serializability by modules is guaranteed, which is a generalization of serializability by levels used in database transactions.

4.5. Parallel Nesting in TMs

The linear nested transactions that we have presented in the previous Section 4.4. may be represented in a tree structure: Each transaction is a node; the parenthood relations are established by directed edges from the child to the parent transaction; and the root is a top-level transaction. Given this representation, in linear nesting, only one of the branches of the tree may be active at a given time. Conversely, in parallel nested transactions, we may have an arbitrary number of branches in the nesting tree with active transactions because a parent may have multiple nested transactions active at any given time.

Note that reading a variable in parallel nested transactions works the same way as for linear nested transactions as previously described in Section 4.4.. However, the fact that we may now have parallel siblings and different branches of the nesting tree active at a given time make the implementations more complex in practice: In linear nesting a nested transaction can always assume that the write-sets of its ancestors will never change during the nested transaction lifetime whereas for parallel nested transactions that is not true due to concurrent nested commits. Next, we describe some STMs that have parallel nested transactions and in which this difficulty and other concerns will be explained.

4.5.1. NeSTM

The Nested STM (NeSTM [3]) is based on McRT-STM as a blocking, eager conflict detection, word-granularity TM with undo logs for writes and a global version clock for serializability. In the original TM, each address is mapped, using a hashing function, to a variable that acts either as a lock or as a 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. Moreover, every transaction is uniquely identified by an identification number.

In the extension of this system to support parallel nesting, the authors argue that the most important point is that it should not interfere with the performance of workloads in which nesting is not used. They were also driven by the intent of keeping the memory footprint as close to constant as possible, regardless of the nesting depth in use. Also, the assumption that no other transaction could access a locked variable in the original system is no longer true: due to the parallel nested transactions, other transactions can correctly access the locked object as long as they are descendants of the owner. To allow this, the ownership information was always made available in the lock to query the ancestor relationship at any time. Similarly, the version number must also be visible at all times to serialize the conflicting transactions. Consequently, the lock variables now reserve some bits to identify the transaction owning it, whereas the rest are used for the version number, allowing visible readers despite the current lock mode. This leads to two practical consequences: There is a maximum number of concurrent transactions at a given time and the transaction identifier overflows several orders of magnitude faster than normal.

At transaction start, the global clock is used to timestamp the transaction. Reads will cause an abort if a variable was written since the transaction started. This might cause unnecessary aborts: T_i did not perform any access, T_k commits values, T_i reads one of the values and will abort.

When writing a value, the transaction will attempt to acquire the lock corresponding to the variable and then it will validate the object: The transaction attempting to write, as well as its ancestors, must not have a timestamp smaller than the object'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. In Fig. 3, we present an example in which the nested transaction D attempts to acquire the lock corresponding to a variable that was previously owned by D's ancestor. In this case, the validation process will only be performed for D and C due to the optimization described. Yet, this mechanism with yields considerable costs in terms of computation at high depth levels.

Given that 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 serializability. This was solved by introducing a lock at each transaction and make nested transactions acquire their parent's lock in mutual exclusion with their siblings.

Moreover, NeSTM is subject to livelocks at the level of nested transactions. If T_1 writes to x and T_2 writes to y, they will both have acquired their ownership. 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 since those variables are owned neither by them or their ancestors. 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.

4.5.2. HParSTM

The Hierarchy-based Parallel STM (HParSTM [5]) is based on Imbs' STM [15], thus obeying opacity and progressiveness. The novelty of this work is that it allows a parent to execute concurrently with its children nested transactions. The advantage is that it allows more nodes in the transactional tree to be active in computations concurrently that enhances the distribution of tasks and requires less depth of nesting due to useless parents standing-by.

The same protocol used for top-level transactions is extended for nesting by replicating most control data structures. The baseline STM design promotes a mixed invalidation strategy with visible readers and lazy lock acquisition and write-back on commit time. To achieve this, a global structure is used where doomed transactions are registered. This is achieved by having a transaction's commit procedure to invalidate active readers of objects that it is writing-back in the aforementioned structure. Any transaction has to check that it does not belong to the doomed transactions prior to commit. Moreover, this information is also scattered across the shared objects which have a forbidden set associated to them: if T_1 read x and T_2 wrote x and y followed by commit, it not only adds T_1 to

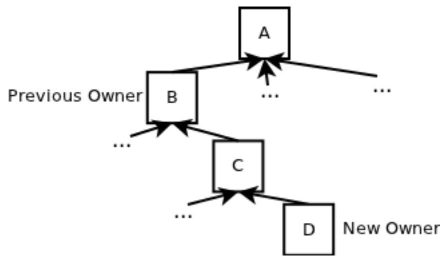


Figure 3. Nesting tree in which A is the top level transaction. In this example a transactional variable, say x, was held by B. When D attempts to acquire the ownership of x, it is able to do so because B is an ancestor of D. Some branches were omitted to simplify the example.

the global doomed set, but also to the forbidden set of x and y. If T_1 attempts to read y it will fail to do so, as otherwise, that would be an inconsistent view state. This procedure is used by nested transactions, except that they must ensure that these invalidation sets contain neither its id or any of its ancestors'.

The extension performed for nesting parallel transactions also synchronizes merges in a parent transaction by concurrent siblings (and the parent's execution itself) with mutual exclusion.

4.5.3. NePalTM

The Nested Parallelism for Transactional Memory (NePalTM [6]) provides in-place updates with strict two-phase locking for writes. Memory addresses are mapped to transactional records that may cover several addresses. The transactional records may be read in two modes: Version timestamps are used for optimistic readers whereas in pessimistic mode they have to acquire the lock in read-mode. Therefore it actually provides both visible and invisible readers.

The NePalTM was built on top of OpenMP[2] and Intel's STM[3] to integrate parallel and atomic blocks.

The execution model in use is Single Lock Atomicity: The atomic blocks behave as if they were protected by a single lock. This model was extended to Hierarchical Lock Atomicity to allow deep nesting of transactions. An abstract lock [50] is used to serialize executions of transactions with the same transactional parent by having direct children of top root atomic blocks (shallow nested transactions) to proceed optimistically resorting to transactions while deep nested parallel transactions (spawned by a given nested transaction) run sequentially in mutual exclusion.

Therefore, NePalTM allows only optimistic concurrency between nested transactions that are directly connected to a top-level transaction (shallow nesting). Should such nested transactions have their own nested transactions (deep nesting), NePalTM has the severe limitation of requiring such sibling transactions to run in mutual exclusion. In other words, it does not support fully-parallel nesting. Their choice was motivated by Agrawal et al [2] who argued that the design of nested parallel transactions is so complex that its performance may never be worth it.

4.5.4. CWSTM

This approach builds on the Cilk [51] language that allows the programmer to use constructs such as `spawn` to create new threads with assigned tasks. The CWSTM [2] dynamically unfolds the program execution into a computation tree that is used for eager conflict detection. This structure serves as the basis for a work-stealing algorithm that will allow exploring transactions' inner parallelism by available threads in the system.

The work-stealing [45] technique is a means of distributing tasks to threads: Each thread maintains its double-ended queue (also known as a dequeue) of tasks; when it runs out of work, it reaches the top of another thread's dequeue and steals a task that it will accomplish on its behalf. Given the uniform random access for stealing, there should never exist any contention in accessing a dequeue as long as there is work left to be done.

The extension provided a new keyword, `atomic`, which allows specifying parallel blocks in conjunction with `spawn`, providing it with the atomicity property. Moreover, the CWSTM uses the aforementioned computation-tree for eager conflict detection in a way that is independent of the nesting depth. Each

[2]http://openmp.org/wp/
[3]http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/

transactional object has an associated access stack in which entries correspond to accesses performed by active transactions. The contents of these stacks is a form of multiple-readers-single-writer locking scheme: The last entry always corresponds to the youngest descendant writer transaction or a set of reader transactions all descendant of a common writer ancestor. Therefore, bellow the first entry there may only exist accesses of descendants of the last access owner. This way, as soon as a transaction accesses an object, it may eagerly detect a conflict.

4.5.5. PNSTM

The Parallel Nesting STM (PNSTM [52]) was based on the ideas that CWSTM pioneered. However, the algorithm that had been proposed was considered to be too complex: No practical implementation was provided.

Therefore, the PNSTM provides a simpler work-stealing approach with a single global queue such that the programs' blocks may be enqueued for concurrent transactional execution. Similarly to CWSTM, a stack is also associated with each transactional object containing accesses (non distinguished between reads or 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.

To achieve constant time queries for eager conflict detection, regarding ancestor ownership of a shared variable, a set of transactions may be represented in a memory word by having each bit assigned to a transaction. This way, when T_i accesses a variable last accessed by T_j, a conflict is detected by operating on both transactions' bit vectors and deciding if one of them is ancestor of the other using bitwise operations: Assuming vec_i is the bit vector corresponding to T_i, we have a conflict when $(\overline{vec_i} \wedge (vec_i \bigoplus vec_j) \neq 0)$. In Fig. 4 I present an example where these operations are used to validate accesses to a variable x by various transactions of a nesting tree.

On one hand, this limits the maximum number of transactions on the system at all times. To work around that, they introduced the concept of epochs, such that a transaction identifier only has meaning when paired with the corresponding epoch. Moreover, the system would be limited to a given maximum number of concurrent transactions. The authors claim that no more parallelism would be attained over that limit if it is larger than the maximum number of worker threads. Consequently, they build on that assumption and provide some ways of reusing identifiers and making it harder to reach the limit.

When a transaction commits, it leaves behind many 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 performs 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 committed but not yet reclaimed transaction. The authors show that it

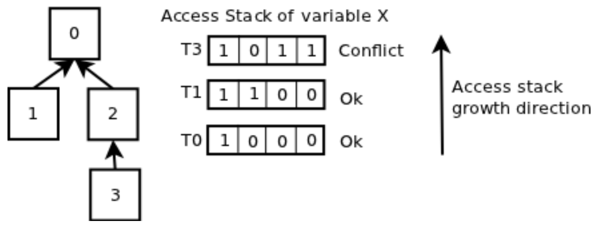


Figure 4. Nesting tree in which transactions are identified by bit nums according to the PNSTM. The access stack for variable x is shown when T_0, T_1 and T_3 access x in that order. Note that the last access creates a conflict.

is possible to avoid it by resorting to a global structure maintaining data about all the committed transactions and some lazy cleaning up.

4.5.6. SSTM

In a different setting, the Sibling STM (SSTM [13]) considers that sibling, concurrent, nested transactions may have relationships and be dependent among each other. The authors present the notion of coordinated sibling transactions, a more powerful and expressive generalization of the traditional model of parallel closed-nested transactions.

The programming constructs created to facilitate expressing intra-transaction concurrency, allowing siblings to affect each others' outcomes, are:

- OR: Two nested transactions that may be parallelized independently.
- AND: Both nested transactions have to be successful, which corresponds to segments of a higher level atomic action that has been split for smaller recovery in case of conflicts. It is a programming error to have nested transactions that may conflict with each other connected by this construct.
- XOR - Only one transaction may be successful. This operation is motivated by speculative actions from which only one may be able to make its results public.

This algorithm was implemented over the Microsoft's Common Language Runtime execution environment. In particular, it makes use of the baseline two-phase commit protocol (2PC) that is provided in the runtime libraries. Consequently, there are many overheads inherent to the distributed nature of the 2PC that could have been avoided.

4.6. Summary of Parallel Nesting Implementations

It is undeniable that providing nesting models along with inner parallelism may unveil yet more concurrency in our programs. Some of the parallel nesting implementations that we have seen attempted to present and describe how it will actually be used by the programmer with seamless integration of thread creation and transaction nesting but end up with depth-dependent algorithms.

For instance, the SSTM explored a unique perspective in which nested transactions may interfere with each other's outcome. However, their algorithm is not provided in a detailed manner and is more interested on how to make use of the underlying runtime of choice.

The CWSTM also took into account the composition of atomic and parallel blocks in the language. In addition to that, it was the first one to show a nesting depth independent algorithm, but did not provide any implementation or evaluation.

On the other hand, the NePalTM provided a model that is not too powerful, but still allows unveiling some concurrency in transactions while maintaining composability (that has sequential bottlenecks given the mutual exclusion on deep nesting). Despite this, they provide a compiler-assisted STM that allowed for a seamless integration of atomic and parallel blocks.

Conversely, the NeSTM presented many of the difficulties that come up when providing nested parallelism, but some of them (which may break opacity) are only solved heuristically. The HParSTM design informally proved some guarantees that we have described but did not present any evaluation.

It is likely that some of the global structures they used inhibit scalability as it breaks the disjoint-access parallelism property and are intensively used for conflict detection.

Finally, the PNSTM provided an efficient algorithm but all accesses are assumed to be writes that precludes some read-read potential concurrency.

All these STMs are lock-based and single-version. Consequently, what is left to explore is providing similar parallel nesting models in the context of a lock-free multi-version STM. Moreover, although Agrawal et al [2] suggested that lazy conflict detection required work proportional to the depth, that is yet left to be confirmed for a different setting where the STM keeps multiple versions of the same object.

4.7. Adapting a TM for Parallel Nesting

We have briefly introduced parallel nesting in Section 4.5. based on work regarding TM implementations that provided it. Yet, we provided mostly intuitions because there has not been a specific attempt to define them more formally for parallel nested transactions. Consequently, we now present those intuitions more carefully.

We use closed nested transactions [42] as the basis for parallel nested transactions. Recall that, on commit, a closed nested transaction merges its read- and write-sets with its parent's. If the closed nested transaction aborts, it may rollback only the atomic action corresponding to itself rather than the whole top-level, depending on the conflict that caused the abort.

In parallel nesting, a transaction may have multiple nested transactions running concurrently. Two nested transactions are said to be siblings if they have the same direct parent. Each top-level transaction may now unfold a nesting tree with the following characteristics:

- Every node in the tree may have an arbitrary number of active children nodes. The parent and their children are connected by edges representing a parenthood relation directed from the children to their parent.
- A node performs transactional accesses only when all its children nodes are no longer active.
- The ancestor set is calculated in the same way as for the linear nesting model.

A nested transaction can read a transactional variable in the same way as for linear nesting (described in Section 4.4.) but with the following difference: When a nested transaction T_i finds out a write in its ancestor T_k private write-set it does not necessarily guarantee that it is safe to read it. Consider the following execution: $W_{T_k}(x,1)$; $S_{T_k}(T_i,T_j)$; $R_{T_i}(x,1)$; $W_{T_j}(x,2)$; $C_{T_j}(ok)$; $R_{T_i}(x,?)$. In this example, the last read performed by T_i would find the value 2 for x in T_k but returning it would break the correctness criterion (assuming opacity described in Section 3.1.). The alternatives are to return the value 1 if the TM is multi-version or to abort T_i.

We consider that the need for a parent to execute concurrently with its children may be satisfied by having the parent spawn a nested transaction to execute the following code that belonged to the parent. Therefore, in practice, the parent thread executes a nested transaction encapsulating the parent code. If the parent code spawns any further transactions in the following code, a barrier is placed so that the previously spawned nested transactions finish together with the artificially created nested transaction on the parent.

5. Conclusions

The widespread growth of parallel computation power has unveiled the concern for the development of scalable applications. In the context of synchronizing access to shared data in these applications, we have addressed the pitfalls of mutual exclusion resorting to locks, and how the Transactional Memory abstraction may solve those issues.

This abstraction enables transactions to compose with each other. This nesting capability provides more flexibility to an application using a TM. However, it is still limited when code that runs under an atomic assumption makes use of multi-threading.

We have shown that parallel nested transactions may be used to work around those concerns. Moreover, we have elaborated how this nesting model may prove useful in highly contending write-dominated workloads. We are now left with the challenge of finding out whether it is possible to design and implement such idea in a way that it effectively produces better results than the existing alternatives.

References

[1] M. Herlihy, V. Luchangco, M. Moir and W.N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the* 22nd *Symposium on Principles of Distributed Computing (PODC '03), 92–101*, 2003.

[2] K. Agrawal, J. T. Fineman and J. Sukha. Nested parallelism in transactional memory. In *Proceedings of the* 13th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08), 163–174*, 2008.

[3] W. Baek and C. Kozyrakis. NesTM: Implementing and Evaluating Nested Parallelism in Software Transactional Memory. In *Proceedings of the* 9th *International Conference on Parallel Architectures and Compilation Techniques (PACT '10), 253–262*, 2010.

[4] K. Agrawal, I. Lee and J. Sukha. Safe open-nested transactions through ownership. In *Proceedings of the* 12th *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '08), 151–162*, 2008.

[5] R. Kumar and K. Vidyasankar. HParSTM: A Hierarchy-based STM Protocol for Supporting Nested Parallelism. In *the* 6th *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '11)*, 2011.

[6] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian and R. Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *Proceedings of the* 23rd *European Conference on Object-Oriented Programming (ECOOP '09), 123–14*, 2009.

[7] P. Felber, V. Gramoli and R. Guerraoui. Elastic transactions. In *Proceedings of the* 23rd *International Symposium on Distributed Computing (DISC '09), 93–107*, 2009.

[8] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift and D. A. Wood. Supporting Nested Transactional Memory in LogTM. 12th International Conference on Architectural Support for Programming Languages and Operating Systems In *SIGPLAN Notices (Proceedings of the 2006 ASPLOS Conference)*, 41(11):359–370, 2006

[9] T. Harris, S. Marlow, S. Peyton-Jones and M. Herlihy. Composable memory transactions. In *Proceedings of the* 10th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05), 48–60*, 2005.

[10] V. Luchangco and V. J. Marathe. Transaction communicators: enabling cooperation among concurrent transactions. In *Proceedings of the* 16th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '11), 169–178*, 2011.

[11] B. Carlstrom, A. Mcdonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis and K. Olukotun. The ATOMOS Transactional Programming Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06), 1–13*, 2006.

[12] S. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the* 16th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '11), 179–188*, 2011.

[13] H. Ramadan and E. Witchel. The xfork in the road to coordinated sibling transactions. In *the* 4th *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '09)*, 2009.

[14] T. Harris and S. Stipic. Abstract Nested Transactions. In *the* 2nd *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '07)*, 2007.

[15] D. Imbs and M. Raynal. A Lock-Based STM Protocol That Satisfies Opacity and Progressiveness. In *Proceedings of the* 12th *International Conference on Principles of Distributed Systems (OPODIS '08), 226–245*, 2008.

[16] A. Dragojevic, R. Guerraoui and M. Kapalka. Stretching transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), 155–165*, 2009.

[17] P. Felber, C. Fetzer and T. Riegel. Dynamic performance tuning of Word-Based Software Transactional Memory. In *Proceedings of the* 13th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08), 237–246*, 2008.

[18] T. Riegel, P. Felber and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the* 20th *International Symposium on Distributed Computing (DISC '06), 284–298*, 2006.

[19] D. Dice, O. Shalev and N. Shavit. Transactional Locking II. In *Proceedings of the* 20th *International Symposium on Distributed Computing (DISC '06), 194–208*, 2006.

[20] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III and M. Scott. Lowering the overhead of Software Transactional Memory. In *the* 1st *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '06)*, 2006.

[21] V. Marathe, W. Scherer III and M. Scott. Adaptive Software Transactional Memory. In *Proceedings of the* 19th *International Symposium on Distributed Computing (DISC '05), 354–368*, 2005.

[22] B. Saha, A.-R. Adl-Tabatabai, R.L. Hudson, C. Cao Minh and B. Hertzberg. McRT-STM: a high performance Software Transactional Memory system for a multi-core runtime. In *Proceedings of the* 11th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06), 187–197*, 2006.

[23] L. Dalessandro, M. Spear and M. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the* 15th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10), 67–78*, 2010.

[24] A. Oram, G. Wilson and S. Jones. Beautiful Code: Leading Programmers Explain How They Think. Published by O'Reilly, 2007.

[25] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming. Published by Morgan Kaufmann, 2008.

[26] C. Papadimitriou. The serializability of concurrent database updates. In *Journal of the ACM, 26(4), 631–653*, 1979.

[27] Transactions in Transactional Workflows. D. Worah and A. Sheth. In *Advanced Transaction Models and Architectures, 3–34*, 1997

[28] M. Herlihy, J. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the* 20th *Annual International Symposium on Computer Architecture (ISCA '93), 289–300*, 1993.

[29] A. Dragojevic, P. Felber, V. Gramoli and R. Guerraoui. Why STM can be more than a research toy. In *Communications of the ACM, 54(4), 70–77*, 2011.

[30] V. Pankratius and A. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the* 23rd *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11), 43–52*, 2011.

[31] 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), 175-184*, 2008.

[32] D. Perelman, R. Fan and I. Keidar. On maintaining multiple versions in STM. In *Proceedings of the* 29th *Symposium on Principles of Distributed Computing (PODC), 16–25*, 2010.

[33] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the* 12th *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '08), 304–313*, 2008.

[34] P. Kuznetsov and S. Ravi. On the Cost of Concurrency in Transactional Memory. In *Proceedings of* 15th *International Conference On Principles Of Distributed Systems (OPODIS '11), 112–117*, 2011.

[35] H. Attiya, E. Hillel and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the* 21st *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '09), 69–78*, 2009.

[36] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *Proceedings of the* 36th *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09), 404–415*, 2009.

[37] R. Guerraoui, T. Henzinger and V. Singh. Permissiveness in Transactional Memories. In *Proceedings of the* 22nd *International Symposium on Distributed Computing (DISC '08), 305–319*, 2008.

[38] K. Fraser. Practical lock-freedom. Ph.D. Thesis, Cambridge University Computer Laboratory, also available as Technical Report UCAM-CL-TR-579, 2004.

[39] J. Gottschlich and D. Connors. DracoSTM: a practical C++ approach to Software Transactional Memory. In *Proceedings of the Symposium on Library-Centric Software Design (LCSD '07), 52–66*, 2007.

[40] M. Abadi, T. Harris and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the* 14th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09), 185–196*, 2009.

[41] J. Moss. Open nested transactions: Semantics and support. In *Poster presented at Workshop on Memory Performance Issues (WMPI '06)*, 2006.

[42] J. Moss and A. Hosking. Nested Transactional Memory: Model and Architecture Sketches. In *Science of Computer Programming, 63(2), 186–201*, 2006.

[43] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. Murphy, B. Saha and T. Shpeisman. Compiler and runtime support for efficient Software Transactional Memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06), 26–37*, 2006.

[44] K. Moore, J. Bobba, M. Moravan, M. Hill and D. Wood. LogTM: log-based transactional memory. In *Proceedings of the* 12^{th} *High-Performance Computer Architecture International Symposium (HPCA '06), 254–265*, 2006.

[45] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the* 35^{th} *Annual Symposium on Foundations of Computer Science (FOCS '94), 356–368*, 1994. Society.

[46] K. Olukotun and L. Hammond. The Future of Microprocessors. In *ACM Queue, 3(7), 26–29*, September 2005.

[47] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. In *Dr. Dobb's Journal, 30(3)*, 2005

[48] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the* 14^{th} *ACM Symposium on Principles of Distributed Computing (PODC), 204–213*, 1995.

[49] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. In *ACM Transactions on Programming Languages and Systems, 12(3), 463–492*, June 1990.

[50] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. Hosking, R. Hudson, J. Moss, B. Saha and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the* 12^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07), 68–78*, 2007.

[51] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing, 37(1), 55–69*, August 1996.

[52] J. Barreto, A. Dragojevic, P. Ferreira, R. Guerraoui and M. Kapalka. Leveraging parallel nesting in transactional memory. In *Proceedings of the* 15^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10), 91–100*, 2010.

[53] C. Rossbach, O. Hofmann, and E. Witchel. Is transactional programming actually easier?. In *Proceedings of the* 15^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10), 47–56*, 2010.

[54] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? In *ACM Queue, 6(5)*, 2008.

[55] R. Martin. SOLID Design Principles and Design Patterns. In *http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod*, last verified 30-12-2011, 2000.

[56] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the* 23^{rd} *International Conference on Distributed Computing Systems (ICDCS '03), 522–529*, 2003.

[57] H. Attiya and E. Hillel. Single-version STMs can be multi-version permissive. In *Proceedings of the* 12th *International Conference on Distributed Computing and Networking (ICDCN '11), 83–94*, 2011.

[58] M. Spear, V. Marathe, W. Scherer III and M. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the* 20th *International Symposium on Distributed Computing (DISC '06), 179–193*, 2006.