Nested parallelism in transactional memory

Ricardo Filipe and João Barreto

Abstract We are witnessing an increase in the parallel power of computers for the foreseeable future, which requires parallel programming tools and models that can take advantage of the higher number of hardware threads. For some applications, reaching up to such high parallelism requires going beyond the typical monolithic parallel model: it calls for exposing fine-grained parallel tasks that might exist in a program, possibly nested within memory transactions.
While most current mainstream transactional memory (TM) systems do not yet support nested parallel transactions, recent research has proposed approaches that leverage TM with support for fine-grained parallel transactional nesting. These novel solutions promise to unleash the parallel power of TM to unprecedented levels. This chapter addresses parallel nesting models in transactional memory from two distinct perspectives.
We start from the programmer's perspective, studying the spectrum of parallel-nested models that are available to programmers, and giving a practical tutorial on the utility of each model, as well as the languages, tools and frameworks that help programmers build nested-parallel programs. We then turn to the perspective of a TM runtime designer, focusing on state-of-the art algorithms that support nested parallelism.

1 Introduction

Harnessing the parallel power of today's computers calls for concurrent programs that expose and exploit as much parallelism as the ever increasing hardware thread count. More than easily coding concurrent programs that yield *some* parallelism, we

Ricardo Filipe
Instituto Superior Técnico, Universidade de Lisboa /INESC-ID e-mail: rfilipe@gsd.inesc-id.pt

João Barreto
Instituto Superior Técnico, Universidade de Lisboa /INESC-ID e-mail: jpbarreto@gsd.inesc-id.pt

want concurrent programs that expose *as much parallelism as the ever increasing hardware thread count*.

This goal becomes dramatically more challenging as affordable multicore machines include more and more cores each year. While 4-core processors supporting up to eight simultaneous hardware threads are already regarded as commodity hardware, 8-core, 16-core and even chips with tens or hundreds of cores promise to be an affordable reality soon [1].

Achieving such parallelism levels will not always be possible with the traditional monolithic organization of coarse-grained parallel threads. For many real applications, the programmer may not be able to find enough coarse-grained top-level parallelism to fork. Hence, the alternative is to recursively expose the fine-grained parallel tasks that might exist within coarser-grain parallel tasks in the program. This leads to *nested-parallel* programs.

As a motivational example, let us assume that a programmer building an application finds different tasks that, according to the application semantics, can safely run in parallel threads. Furthermore, inside such tasks, the programmer finds that some sub-tasks of a same task can also be parallelized in a fork-join fashion. Proceeding recursively with this approach, the final application will comprise a dynamic tree of nested fork-join tasks, each of which can run in concurrent threads to exploit the available hardware resources. This tree can even be deeper if we consider that some tasks may invoke functions from other modules (e.g., a library call) that may themselves be implemented by nested-parallel programs.

If the tasks work on shared data, then the above application will most likely have concurrent accesses to that data. Concurrency in traditional parallel programming is well known to be a hard problem to tackle, as we need to correctly synchronize access to shared data. Shifting to nested-parallel programming can further complicate synchronization to dantesque levels.

Nested-parallel programs comprise dynamic trees of tasks, running at concurrent threads, where correct synchronization depends on ancestor-descendant relations. On the one hand, data contention between concurrent threads needs to be synchronized. But, on the other hand, tasks that are ancestor/descendant of each other need to be treated differently: for instance, a nested task trying to access some memory location locked by some ancestor may be allowed to proceed with the access. Further, deadlock situations are more likely and harder to prevent, as they may happen between tasks at any nesting depths.

Relying on the programmer to explicitly solve such intricate synchronization challenges (e.g., using lock-based programming) is usually unrealistic for the average programmer. Except for embarrassingly parallel programs, the programmer is strongly discouraged to explore into the possibilities of nested-parallel programming.

Memory transactions, in contrast, are an elegant and effective solution to hide the hard synchronization parallel programming, especially if nested, away from the programmer. This makes transactional memory (TM) a promising paradigm to leverage fine-grained nested parallelism in tomorrow's multi/many-core machines.

Hereafter, let us designate the original non-nested TM programming model as *flat-parallel* (in contrast to the nested-parallel counterpart). The key insight is that the flat-parallel TM programming model is easily extensible to support nested transactions [2], an extension that has been introduced well earlier in the context of database transactions [3]. Essentially, a nested transaction is one whose execution is contained inside another transaction's execution. A program may hence recursively create nested transaction trees while executing.

When building a parallel-nested program, the programmer simply needs to apply the same rule that she was required to follow in traditional flat-parallel programming: to identify each code region that needs to run atomically and wrap it inside a transaction. Since transactions are composable [4], if all the atomic regions in a task have been properly defined, then executing such a task nested within a nested-parallel program will be correct. This holds true even if some of the tasks in a nested-parallel program belong to different modules, whose implementation the programmer does not know about (e.g. a call to a parallelized library function).

Programming in a nested-parallel fashion using TM, when compared to flat-parallel programming, introduces new challenges that programmers must be aware of in order to build correct and efficient programs. Firstly, the nested-parallel model is more complex than the flat one. Secondly, starting nested tasks may be cumbersome, error-prone and lead to inefficient, slow and not scalable programs if is not handled correctly. Finally, not of all today's mainstream TM runtimes support a fully nested-parallel model. Instead, many TM runtimes support limited nesting models, which need to be taken into account by the programmer.

This chapter approaches parallel nesting models in TM from two distinct perspectives. We start from the programmer's perspective. Section 2 studies the spectrum of parallel-nested models that are available to programmers, and gives a practical tutorial on the utility of each model. Section 3 then surveys languages, tools and frameworks that help programmers build nested-parallel programs in TM. Section 4 then focuses on the inner works of TM runtimes that support parallel nesting, describing state-of-the art algorithms. Finally, Section 5 summarizes.

2 Nested parallelism models in transactional memory

In theory, the TM model is extensible to support parallel-nested programs [5]. This extension implies redefining the correctness guarantees that were originally defined in the context of flat-parallel programming in TM.

The key insight is that correctness must now consider the ancestor-descendant relationships between parallel transactions, as we shall detail next.

However, supporting parallel nesting also implies changes to the TM runtime that may introduce substantial overheads or limit scalability. For this reason, many current mainstream TM runtimes opt for limited nesting models. For the sake of efficiency of flat-parallel programs, these typically restrict the nested parallelism that programmers can actually extract from their programs.

Algorithm 1: Example of parallel nested transactions

```
1 function sb7-longTraversal(root)
2     atomic {
3         parallel {
4             sb7-traverseComplexAssembly(root.leaf1);
5             sb7-traverseComplexAssembly(root.leaf2);
6             sb7-traverseComplexAssembly(root.leaf3);
7         }
8     }
```

Hence, the reality is that, instead of a single nested-parallel model, TM runtimes actually offer a spectrum of models. It is, of course, important that the programmer understands each model in order to produce programs that, while ensuring correctness, are able to fully exploit the model supported by the underlying runtime.

This section presents and discusses each different model in the spectrum of nested models for TM. We start by focusing on the pure parallel-nested model, before delving in restricted variants of such a model in the subsequent subsections.

2.1 Parallel Nesting

Simply put, the nested-parallel model for TM means that the TM runtime supports nested transactions and allows the child transactions of a common parent to run in parallel. This model is a straightforward extension of the closed nesting model proposed by Moss and Hosking [2].

Conceptually, the execution of a nested-parallel program yields a dynamic tree of active transactions, inter-connected by child-parent relations. At any moment, some of the transactions will be running, while others will be waiting (for instance, for some processor to become available, or waiting for their children to commit).

We illustrate with an excerpt taken from a modified long transaction of the popular STMBench7 benchmark [6], presented in Algorithm 1. Method sb7-longTraversal includes an atomic region (i.e. encloses a transaction), which calls the sb7-traverseComplexAssembly method for each leaf of the root data item. The sb7-traverseComplexAssembly method also executes a transaction inside of it. In this example the programmer is calling methods within the same program, but they could be calls to an external library.

In the above example, the programmer calls the sb7-traverseComplexAssembly methods in parallel threads, thereby building a nested-parallel program. These methods will perform accesses that may conflict with transactions running concurrently at other threads. Furthermore, the parallel-nested threads may also contend for shared memory locations. If that is the case, the programmer should have identified the code regions at the called methods that need to run atomically and created transactions to ensure the necessary synchronization.

The transactions defined by the atomic regions within sb7-traverseComplexAssembly will compose with the parent transaction initiated at method sb7-longTraversal. In other words, the sb7-traverseComplexAssembly methods will start nested transactions. We call each such nested transaction a *child* of the parent transaction from sb7-longTraversal. By extension, we say that two transactions are *siblings* if they have a common parent transaction. Furthermore, we define that transaction t is an *ancestor* of transaction s if t is included in the path from s's node to the root node in the tree of nested transactions.

For model simplicity, most definitions of the nested-parallel transactional model (e.g. [5]) assume that the parent transaction halts until all the threads that it spawned (and the inherent nested transactions) complete. Only after all children tasks finish does the parent's execution continue. We adopt such assumption too. Hence, when a given nested task is running, all its ancestors' threads are waiting. Accordingly, when a nested transaction is active, all its ancestor transactions are waiting.

A nested transaction is seen as executing after all the accesses that its ancestors have performed so far. In particular, when some transaction t reads from a memory location that has been written by any of its ancestors, t should observe the most recently written value by its ancestors.

Each nested transaction runs in isolation relatively to any other concurrent transaction. More precisely, the concurrent transactions of a given transaction include its own siblings and all its ancestors' siblings and their descendants (including the root transactions that are concurrent to the transaction's root ancestor).Note that a transaction never runs concurrently with its descendants, as it waits for the descendants to complete.

Conceptually, a nested transaction has its own read set and write set. This enables rolling back the nested transaction without having to roll back its entire root transaction.

On commit, a nested transaction's read and write sets are inherited by the transaction's parent. In other words, the reads and writes of the committed nested transaction are, from that moment on, considered to have been performed on behalf of the parent transaction.

Committing a nested transaction does not make its writes visible to the rest of the world. [1] Instead, committing a nested transaction means that the committed writes become visible to its active siblings and to its ancestors (which are blocked until all children commit). Following this rule recursively, the writes of a nested transaction become gradually visible to other transactions, starting at the set of siblings of the transaction and then going upwards the nesting tree.

The nested-parallel transactional model is very powerful to ease programmers' lives when exploiting nested parallelism in their programs. The key insight is that the nested-parallel model retains the composability of the traditional flat model. Hence, when shifting from the flat-parallel model to the nested-parallel one, the program-

[1] This means that we consider only a closed nesting model. An alternative is the open nesting model introduced briefly in Section 2.3.2. We leave that alternative out of the scope of this chapter, since no research work on parallel nesting support includes open nesting. In theory, however, open nesting is applicable to both parallel and linear nesting models.

mer is required to apply the very same principle as before: to identify regions within the program that are atomic and wrap them in transactions. Having done that, correct synchronization is ensured by the TM runtime even for a program that has been structured in a nested-parallel fashion. This holds true even if some of the tasks in a nested-parallel program belong to different modules, whose implementation the programmer does not know about (e.g. a call to a parallelized library function).

However, porting a flat parallel program with monolithic coarse-grained threads to a nested-parallel alternative that exposes more fine-grained parallelism is not transparent and requires caution from the programmer. Let us consider a thread that executes a seguence of tasks. Before parallelizing such tasks, the programmer needs to carefully confirm that:

- The candidate tasks to parallelize safely commute. Parallelizing them can yield executions where the serialized order of the transactions within the parallelized tasks is different than the serial order in the original flat thread's program. Whether such a reordering of such tasks is safe or not depends on the semantics of the operations being performed at each candidate task.
  When two or more tasks are not commutable, spawning them inside nested-parallel threads is not a safe choice.
- The tasks to parallelize should be long enough to compensate the overheads associated with nesting. Namely, the cost of forking/joining the new threads to run each task in parallel, the costs of beginning and committing nested transactions, the additional overheads of deeper nesting in the transactional tree, among others. Nested tasks should only be parallelized when the associated speed-ups clearly compensate the above costs.
- There are available hardware contexts to run each task in parallel. Of course, exposing additional fine-grained parallelism is advantageous as long as there are idle hardware contexts to run the spawned nested tasks. Blindly spawning nested tasks may lead to pathological executions where spawned tasks are actually condemned to spend substantial periods waiting for an available core. Furthermore, it increases thread preemption cost.

Ensuring the above conditions is not trivial and is, perhaps, the key obstacle to building efficient nested-parallel programs. There are, however, tools, frameworks and language support that assist the programmer with some of the above issues. We describe some examples of such items in the following sections.

Although appealing in theory, only a few of today's state-of-the-art TMs support this nested-parallel model. As we shall discuss in Section 4, the nested-parallel model brings about a number of technical challenges that can substantially complicate the implementation of a TM runtime. Hence, many TMs offer support for nesting but introduce restrictions that do not exist in the pure nested-parallel model we described previously.

We address such restricted models next.

2.2 Shallow parallel nesting

Volos et al. [7] define one poorer variant of the nested-parallel model, which they call *shallow nesting*. In shallow nesting, a transaction can have several threads executing, in parallel, parts of the transaction's code. However, no nested transactions are allowed.

The memory accesses performed by the threads running on behalf of a common (parallelized) transaction are added to the transaction's read and write set. However, the TM does not guarantee that such threads run in isolation.

Hence, the programmer's role is harder, since shallow nesting places the burden of ensuring correct synchronization among the parallel threads running on behalf of a same transaction. Shallow nesting is, though, a nice fit for parallelizing long transactions that perform multiple independent operations (e.g. a loop on disjoint data).

2.3 Nesting with restricted parallelism

Other variants of the nested-parallel model restrict the allowed parallelism among nested tasks (and transactions).

2.3.1 Hierarchical Lock Atomicity

One such model consists in disallowing sibling transactions (i.e., nested transactions descending from a common parent) to run in parallel. Volos et al. [7] define this as the *Hierarchical Lock Atomicity* (HLA) model. In concept, it is as if each parent transaction has a single lock, which the children transactions need to obtain before proceeding. More precisely, let us consider that some transaction spawned a set of threads. When any of such threads wishes to begin a (nested) transaction, it needs to wait until there is no other sibling or any sibling's descendant transaction running.

Note that, like shallow nesting, HLA also allows a transaction to effectively run in parallel - as long as such parallel threads do not begin simultaneous nested transactions. Hence, in long transactions that can be parallelized into tasks that contain few and short transactions, HLA is able to yield parallel executions that resemble those of the pure nested-parallel model.

2.3.2 Linear nesting

For implementation simplicity, many mainstream TMs support nested transactions but simply disallow a transaction to spawn any threads. In other words, if some parent transaction creates child transactions, then the children will run in the same

thread that runs the parent transaction, one after another. This is called the *linear nesting* model.

Linear nesting imposes a decisive limitation on the potential parallelism that is made available to programmers, who can only create threads in code locations that lie outside atomic blocks. Hence, it severely restricts composability of parallel programs [16], as a program cannot call a parallel library function from inside a transaction without serializing the function [1]. Or, alternatively, the programmer cannot decompose long transactions into parts that do not conflict among each other (at least not too much).

We can actually identify three main variants of the linear nesting model, as follows:

- Flat Nesting.
  The parent transaction sees all modifications to program state made by inner transactions, since child and parent transactions are coupled onto a single transaction. This is the simpler approach, since aborting the child transaction will also abort the parent, but committing the child transaction has no effect until the parent transaction also commits. Flattened transactions are easy to implement, since there is only one transaction in execution coupled with a nesting depth counter. However, this is a poor programming abstraction, since if an explicit abort is issued in a library routine that contains transactions, all surrounding transactions must terminate execution.
- Closed Nesting.
  A closed transaction behaves similarly to a flattening one, except the inner transaction can abort without terminating its parent transaction. When a closed transaction commits or aborts, control passes to its parent. If the inner transaction commits, its changes become visible to the parent. However, they only become visible to other threads when the parent transaction commits. Hence, closed nesting ensures the same correctness properties as flat nesting.
- Open Nesting.
  When an open transaction commits, its changes become visible to all other transactions in the system, even if the parent transaction is still executing. Furthermore, if the parent transaction aborts, the results of the nested open transactions remain committed. Thus, open nesting allows greater concurrency between transactions. For example, it allows concurrent transactions to increment a shared counter without provoking a conflict for the whole parent transaction. While using open transactions allows for greater concurrency in the application, they can subvert the isolation of a parent transaction, thus requiring extra care. For instance, consider the case where a child transaction reads data tentatively written by the parent; then the child transaction commits but the parent transaction later aborts. Now there is some inconsistent global state which depends on a write operation that actually never occurred. Another problematic case is the one where the parent transaction reads some location that the child transaction writes to. The child can commit a new value to that location, and then the parent may abort and read the value that was updated by its child transaction upon re-execution.

2.4 Nested-parallelism with thread-level speculation

As discussed earlier in this section, the nested-parallelism model requires careful reasoning about the semantics of the parent task being parallelized. Namely, the programmer must assert if the work performed by the parallel children tasks is actually commutative.

This assertion may not be trivial for all applications. For the average programmer, this may pose a significant effort and introduce a non-negligible risk of errors due to parallelizing tasks that, after all, were not semantically commutable. At the end, most programmers will most likely feel discouraged from exposing fine-grained parallelism lying within their applications.

Furthermore, some tasks are simply not commutable, as the application's semantics require them to run accordingly to the sequential program's order. That is, any task reordering that leads to different results is simply prohibited by the semantics. However, this does not mean that running the tasks in parallel will always lead to such undesirable executions. Consider, for instance, a sequence of tasks that work on some shared data structure (e.g., a large array or matrix) such that some tasks may occasionally read or write to the same elements in the shared structure. Any task reading from an element that other tasks in the sequence write to should obtain the value updated by the most recent task that, in program order, precedes the reader. Hence, parallelizing these tasks as sibling nested transactions may violate this condition, as the nested-parallel model may serialize siblings in a different order than that of the original program.

A recent research direction has proposed a variant of the nested-parallel model that address the two above issues [8]. This new model combines TM and thread-level speculation (TLS) [9].

As in the nested-parallel model, the programmer can sub-divide a transaction into parallel tasks. The key difference in the hybrid TM+TLS model is that runtime is responsible for ensuring that any data dependencies stemming from the original sequential program order are respected in the speculatively parallelized execution.

This hybrid model eliminates the two issues discussed above. On the one hand, the programmer in doubt about task commutativity can conservatively parallelize a transaction using this hybrid model. Since the underlying runtime guarantees that the parallelized execution will be equivalent to a sequential execution of the same transaction, the parallelized program is correct no matter if the tasks were actually commutable or not. On the other hand, situations where the sequence of tasks in a transaction is not commutable may now be safely parallelized, since the TM ensures that such tasks will be serialized according to program order.

It is thus pertinent to compare the the hybrid TM+TLS model with the nested-parallel model. The TM+TLS model is perhaps more appealing to the average programmer, as it strongly simplifies programming fine-grained parallel programs where the tasks do not commute or the programmer simply is not sure that they commute.

However, the main question is which model is able to actually deliver higher parallelism. In fact, each model can, in theory, achieve more parallelism than the other,

Algorithm 2: Example of nested-parallel programs with TFJ

```
9  function sb7-longTraversal-TFJ(root)
10     transaction(proc, params) {
11         onacid;
12             proc(params);
13         commit;
14     }
15     onacid;
16         spawn transaction(sb7-traverseComplexAssembly, root.leaf1);
17         spawn transaction(sb7-traverseComplexAssembly, root.leaf2);
18         spawn transaction(sb7-traverseComplexAssembly, root.leaf3);
19     commit;
20     }
```

depending on the program being parallelized. As discussed above, the TM+TLS model can expose parallelism in situations where the pure nested-parallel model cannot.

However, in situations where the nested-parallel tasks are commutable, the TM+TLS model is limited. Whereas the pure nested-parallel model is free to serialize the sibling tasks in any order, the TM+TLS model will always enforce the sequential program order. Unfortunately, the sequential program order may not be the serialization order that allows for highest parallelism, when considered among the remaining possible serialization orderings.

3 Support

In order to aid the programmer in building nested parallel programs it should be easy for him to: i) create nested tasks in a fork-join pattern; ii) protect the accesses to regions of shared data using transactions. Recently several frameworks in different programming languages have added support for such mechanisms, which we will now address.

The flat-nesting TM API makes use of functions to start and end transactional code, e.g. tx-begin() and tx-commit(), or simply use an annotation or construct that surrounds the transactional code, e.g. *@Atomic* or *atomic* { }. When using nested transactions there is, usually, a need for an extended TM API that supports each of the models described in Section 2.

The first framework support for parallel nested transactions was proposed by Vitek et al. in Transactional Featherweight Java (TFJ) [10]. TFJ used a *spawn* keyword to create a new thread for executing a transaction, an *onacid* keyword that represents the start of a transaction and a *commit* keyword for ending a transaction (example Algorithm 2). They proceed to define the semantics in which such keywords can be used to program parallel nested applications. Then, they describe

Algorithm 3: Example of nested-parallel programs with Cilk

```
21 function sb7-longTraversal-Cilk(root)
22     atomic {
23         parallel {
24             atomic {
25                 traverseComplexAssembly(root.leaf1);
26             }
27             atomic {
28                 traverseComplexAssembly(root.leaf2);
29             }
30             atomic {
31                 traverseComplexAssembly(root.leaf3);
32             }
33         }
34     }
```

theoretical proofs that validate these keywords as building blocks for any model of nested transactions.

The work on TFJ was followed by Agrawal et al. [5] implementing similar constructs in Cilk, a dynamic multi-threaded language. Cilk already supported executing parallel sections of code, using a *parallel* { } construct, to tell the runtime that there exists a possibility for parallelism, and transactions, using the *atomic* { } construct. The combination of these two constructs allowed for the specification of parallel nested transactions, with an unbounded nesting depth (example Algorithm 3).

The support for parallel nested transactions on TFJ and Cilk executed all sibling transactions independently, as most parallel nested transactions' models require. However, Ramadan et al. [11] argued that this execution model was not expressive enough, and that siblings should affect each other's outcomes. They introduced coordinated sibling transactions in Xfork, a programming construct that allowed TM programmers to express intra-transaction concurrency. Inside an *atomic* { } construct, a TM programmer could define parallel transactions with the construct *xfork (form, numForks, xforkProcedure, data)*, where:

- form : the form of sibling coordination (AND, OR, XOR)
- numForks: the number of concurrent sibling transactions to spawn
- xforkProcedure: a list of procedures to execute inside sibling transactions
- data: a list of arguments for each of the procedures

Xfork supports three forms of coordinated sibling transactions:

- AND: All sibling transactions must succeed, or none succeed
- OR: Sibling transactions succeed or fail independently
- XOR: Only one sibling transaction must succeed

The AND form is used for regular nested parallel transactions (Example Function 4). The OR form emulates independent nested transactions, where all success-

Algorithm 4: Example of nested-parallel programs with xFork

```
35 function sb7-longTraversal-xFork(root)
36     atomic {
37         xfork (AND, 3, { traverseComplexAssembly, traverseComplexAssembly,
       traverseComplexAssembly }, {root.leaf1, root.leaf2, root.leaf3});
38     }
```

Algorithm 5: Example of nested-parallel programs with JVSTM

```
39 @Atomic
40 function sb7-longTraversal-JVSTM(root)
41     @Parallel
42     for each leaf in root do
43         traverseComplexAssembly(leaf);
```

fully completed siblings will commit. The XOR form allows for speculative parallel nested transactions, where if some sibling is successful the parent is also successful. Non-speculatively, the XOR form can execute several transactions in parallel when the programmer knows that only one sibling will commit successfully (e.g. when doing a parallel search for an item on a data structure).

Finally, the work by Diegues et al. [12] uses the annotations *@Atomic* and *@Parallel*, identical to the constructs of Agrawal et al. and DeuceSTM [13], in the Java programming language. These annotations are enough to fully program parallel nested transactions, with an unbounded nesting depth, in JVSTM [12] (example Algorithm 5).

4 Algorithms

Extending a TM runtime with parallel nested transactions support is not trivial. Conflict detection, in particular, becomes much more complex. Not only does the TM need to detect conflicts between concurrent running transactions accessing the same data object, but now the TM must also allow accesses from child transactions to objects written to and commited by its siblings. Handling such accesses in an efficient manner requires a re-organization of the TM data structures.

Therefore, for a TM runtime to fully support nested parallel transactions it has to tackle several challenges that did not exist in the traditional flat nesting scenario:

1. To support partial rollback of child transactions, without affecting the parent
2. To handle concurrent data structures correctly, such as the parent-child read and write sets
3. To coordinate the commit or abort of parent and child transactions

4. To detect conflicts by verifying ancestor-descendant relationships, which may be complicated for deep nested trees

This section addresses several state of the art algorithms for the nested parallel transactions models we presented in Section 2. Since this chapter focuses on parallel nesting models, we omit algorithms that support only linear nesting. A survey of linear nesting algorithms can be found in the technical report of Diegues [14].

Each of the following algorithms solves some or all of the previous challenges in different ways, with different complexity degrees. As discussed in Section 2, some solutions opt for limited models in exchange for better performance or scalability.

4.1 CWSTM

This approach builds on Cilk, a dynamic multi-threaded language that allows the programmer to use special constructs to create new threads with assigned tasks. The CWSTM [5] dynamically unfolds the program execution into a computation tree that is used for conflict detection. This structure serves as the basis for a work-stealing algorithm that allows the exploration of a transaction's inner parallelism.

The work-stealing technique is a means of distributing a set of tasks to threads: Each thread maintains a double-ended queue of tasks; when the thread runs out of work, it reaches the top of another thread's dequeue and steals a task to execute on that thread's 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.

CWSTM uses the aforementioned computation tree for eager conflict detection, with a computational intensity that is independent of the nesting depth. Each transactional object has an associated access stack in which entries correspond to accesses performed by active transactions. The content 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, below the first stack entry there may only exist accesses of descendants of the last access owner. This way, as soon as a transaction accesses an object, that transaction may eagerly detect a conflict.

However, maintaining these per-object stacks is very inefficient. Hence, their effort only resulted in providing a STM specification and a theoretical upper bound for the execution time of a parallel nested transaction. No complete implementation of such design was achieved for this paper, albeit the proposed design solves all of the challenges we described.

4.2 PNSTM

The Parallel Nesting STM (PNSTM) [15] followed the approach of Agrawal et al. and succeeded in implementing an algorithm for parallel nested transactions support. PNSTM provides a simple work-stealing approach with a single global queue, into which the application's blocks may be enqueued for concurrent transactional execution.

Moreover, each transactional object is associated with a stack that contains all the accesses (both reads and writes) performed by active transactions. To achieve constant time ancestor queries for eager conflict detection, the per-object stack is represented by a memory word that has each bit assigned to a transaction (called a *bitnum*). When two transactions access the same object, a conflict is easily detected by performing a bitwise operation on the object's stack.

By using a memory word for this representation they achieved performance improvements but limit the maximum number of transactions on the system at all times. As a workaround, PNSTM uses a mechanism that allows for new transactions to reuse *bitnums* of completed transactions.

The system is limited to a determined maximum number of concurrent transactions. However, PNSTM claims that no more parallelism would be achieved over that limit if it is larger than the maximum number of worker threads.

When a transaction commits, it leaves behind traces in all the objects it accessed, namely the stack frames stating its ownership. To avoid having to go through all the objects in the write-set by locking and merging the frame with the previous entry, PNSTM does that lazily, similarly to Agrawal's algorithm. 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 is possible to avoid it by resorting to a global structure maintaining data about all committed transactions.

This was the first implementation of parallel nesting with constant time ancestor queries, for an arbitrary nesting depth. It solves all of the challenges we presented in a more efficient way, at the cost of a bound in the active threads count.

4.3 NePalTM

The Nested Parallelism for Transactional Memory (NePalTM) [16] provides in-place updates with strict two-phase locking for writes. Memory addresses are mapped to transactional records with a granularity of several addresses.

The transactional records may be read in two modes: in pessimistic mode they have to acquire a lock in read-mode, or by using version timestamps which are accessed by optimistic readers. Therefore, it actually provides both visible and invisible readers.

NePalTM supports the Shallow Nesting model, described in Section 2, by having each member of an atomic region store its own transactional logs (read, write and

undo logs). This way, no synchronization is required to access the logs of an atomic region, and they are all used only at commit time of that atomic region.

NePalTM also supports the Hierarchical Lock Atomicity model, defined in Section 2. In this case, NePalTM has a major limitation of requiring such sibling transactions to run in mutual exclusion. Hence, it does not support parallel nesting entirely. Thus, NePalTM solves the first challenge, of supporting partial rollback, since there is no concurrency between parent and child transactions. It also solves the second challenge, since in shallow nesting members of an atomic region are concurrently logging transactional data.

4.4 NeSTM

The Nested STM (NeSTM) [17] is based on McRT-STM [18]. McRT-STM is a traditional blocking STM, with eager conflict detection, with undo logs for writes at the word granularity. In the extension of McRT-STM to support parallel nesting, the focus point was 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.

The original McRT-STM assumed that no other transaction could access a locked variable. With nested-parallel transactions this is no longer the case: due to the parallel nested transactions, other transactions can correctly access the locked object as long as they are descendants of the owner. When a transaction accesses an object, it locks such an object. That object's lock includes a new field with information about its current owner. This way, when another transaction wishes to access the same object, it may confirm if it is a descendant of the lock's owner.

Similarly, the version number of an object must also be visible at all times, in order to serialize conflicting transactions. Consequently, the lock variable now has some reserved bits to identify the transaction owning it, and the rest of the bits are used for the version number. This scheme allows visible readers even when the object is locked. This leads to two practical consequences: first, there is a maximum number of concurrent transactions at a given time, since the transaction identifier is just a few bits long; second, 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 an object was written since the transaction started. This might cause unnecessary aborts: picture two transactions T_i and T_k; 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. Yet, this mechanism yields considerable costs in terms of computation at deeper 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 making nested transactions acquire their parent's lock in mutual exclusion with their siblings.

In addition, NeSTM is subject to livelocks at the level of nested transactions. Picture two transactions, T_1 who writes to x and T_2 who writes to y, they will both have acquired ownership of the respective objects. Now if the T_1 spawns $T_{1:1}$ while T_2 spawns $T_{2:1}$ and both these nested transactions cross-access y and x, respectively, they will abort since those variables are neither owned 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.

NeSTM solves all of the challenges we identified, in a more efficient manner than PNSTM, but still with several limitations. Baek et al. [19] and Liu et al. [20] studied how hardware acceleration could improve the performance of nested transactional systems, using NeSTM as a baseline.

4.5 HParSTM

The Hierarchy-based Parallel STM (HParSTM) [21] allows a parent to execute concurrently with its children nested transactions. The advantage of this is that it allows more nodes in the transactional tree to be active in computations concurrently, which enhances the distribution of tasks.

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 to register transactions that are doomed to abort. This is accomplished by having a transaction's commit procedure 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 list prior to commit.

Furthermore, this information is also scattered across the shared objects which have a forbidden set associated to them, better defined by an example: if T_1 read x and T_2 wrote x and y followed by commit, it not only adds T_1 to 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, in order to prevent an inconsistent view state.

This procedure is used by nested transactions, except that they must ensure that these invalidation sets contain neither the nested transaction's identifier or any of its ancestors'. The control data structures of nested parallel transactions are merged into the parent transaction by concurrent siblings (and the parent's execution itself) with mutual exclusion.

HparSTM goes even further in the design space of parallel nested transactions algorithms. Although it solves all our challenges, HparSTM still has some limitations when supporting higher levels of nested transactions.

4.6 JVSTM

The first STM to solve all challenges we described in an efficient manner was the work by Diegues et al. in JVSTM [12]. They extended the original JVSTM [22] with parallel nesting support, assuming that each top-level transaction may unfold a nesting tree in which a transaction performs transactional accesses only when all its children are no longer active.

Their approach is to extend VBoxes (JVSTM's placeholders for transactional locations' values) such that transactions may now write directly to the VBoxes, rather than having to maintain a private write set mapping each location written to its new value. In order to distinguish between globally committed values and the tentative values of ongoing transactions, a VBox now contains both values. A permanent value has been consolidated via a commit of some top-level transaction, whereas a tentative value belongs to an active top-level transaction (or any of its children nested transactions), and is thus part of its write-set.

Additionally, each tentative write points to an ownership record (*orec*) that encapsulates the transaction that owns it, the version of the write, and the status of the owner. Each writing transaction creates one such *orec* and propagates it to the transaction's parent when it commits. Through these *orecs* a nested transaction can perform the ancestor query, which depends only on the number of tentative writes on the location.

The algorithm proposed in this work has three major features that make it efficient: a fast path in the read operation that is performed in constant time (independently of the nesting depth); a fast mode for writing, backed up by a slow mode for fallbacks; and a commit operation that is independent of the write-set size.

The fast read path is achieved by checking if the read operation being performed is not a read-after-write. In that case the read operation can be done directly from the last permanent write, and avoid the ancestor query. The fast path for writing occurs when the transaction that is writing to a location already owns that location, thus it can simply overwrite the tentative value. The commit operation of nested transactions simply changes the ownership of *orecs* that the child transaction owns to its parent. The set of location *orecs* is usually smaller than the whole write-set.

4.7 TLSTM

TLSTM is the first algorithm to tackle the challenges of nested-parallelism using thread level speculation. TLSTM extends an existing STM, SwissTM [23]. The key insight is that a SwissTM transaction is used as the speculative execution unit that supports two concepts: STM transactions (defined by the user) and TLS speculative tasks (automatically created at compile time). An STM transaction is seen as a sequence of one or more TLS speculative tasks, which can run out-of-order in a speculative fashion, until they commit sequentially.

Most of the maintenance load of STM and TLS that typically dominates the associated execution overheads is, in fact, common to both approaches. Namely, conflict detection, speculative reads and writes, read-log and write-log maintenance, commit and rollback are issues that both STM and TLS must handle. Hence, by combining both STM and TLS in TLSTM, the overhead associated with the above aspects remains comparable to the overhead of stand-alone STM, rather than doubling.

Cross-transaction conflict detection follows the original approach of SwissTM: using eager, lock-based conflict detection for write/write conflicts, and lazy counter-based validation for read/write conflicts. Within each top-level transaction, cross-task conflict detection relies on the very data structures maintained for cross-transaction conflict detection, with the addition of a task read-set for speculative cross-task reads. TLSTM allows only one task to write on each location at a time, also using eager, lock-based write-write conflict detection. TLSTM validates the task and transaction read-sets at write and commit time, looking for cross-task Write after Read conflicts. Furthermore, TLSTM only allow speculative reads from completed tasks within a transaction.

5 Summary

For many real applications, harnessing the hardware parallelism of modern multi- and many-core machines calls for exposing fine-grained parallel tasks, possibly nested within memory transactions. Memory transactions, being a composable abstraction, are a promising way to enable the average programmer to exploit nested-parallel programming.

This chapter has given an insight into the concepts, techniques and challenges behind nested-parallel programming. We started with a view from the programmer's point of view, describing the nested-parallel model in transactional memory and its variants. Complementarily, we surveyed available support to build and run nested-parallel programs. We then turn to the perspective of a TM runtime designer, studying the state-of-the art algorithms that support currently nested parallelism.

References

1. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108 –109, feb. 2010.
2. J. E. B. Moss and A. L. Hosking, "Nested transactional memory: Model and architecture sketches," *Sci. Comput. Program.*, vol. 63, pp. 186–201, Dec. 2006.
3. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 1992.
4. T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, (New York, NY, USA), pp. 48–60, ACM, 2005.
5. 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, (New York, NY, USA), pp. 163–174, ACM, 2008.
6. R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: A benchmark for software transactional memory," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), pp. 315–324, ACM, 2007.
7. 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)*, pp. 123–147, 2009.
8. J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *Proceedings of the 13th International Middleware Conference*, pp. 187–207, Springer-Verlag New York, Inc., 2012.
9. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, (New York, NY, USA), pp. 521–532, ACM, 1998.
10. J. Vitek, S. Jagannathan, A. Welc, and A. Hosking, "A semantic framework for designer transactions," in *Programming Languages and Systems* (D. Schmidt, ed.), vol. 2986 of *Lecture Notes in Computer Science*, pp. 249–263, Springer Berlin Heidelberg, 2004.
11. H. Ramadan and E. Witchel, "The xfork in the road to coordinated sibling transactions," in *4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2009)*, 2009.
12. N. Diegues and J. Cachopo, "Practical parallel nesting for software transactional memory," in *Distributed Computing* (Y. Afek, ed.), vol. 8205 of *Lecture Notes in Computer Science*, pp. 149–163, Springer Berlin Heidelberg, 2013.
13. G. Korland, N. Shavit, and P. Felber, "Noninvasive concurrency with java stm," in *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.
14. N. Diegues and J. Cachopo, "Review of nesting in transactional memory," tech. rep., Technical Report RT/1/2012, Instituto Superior Técnico/INESC-ID, 2012.
15. J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka, "Leveraging parallel nesting in transactional memory," *SIGPLAN Not.*, vol. 45, pp. 91–100, Jan. 2010.
16. 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)*, pp. 123–147, 2009.
17. 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)*, 2009.

18. B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: A high performance software transactional memory system for a multi-core runtime," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, (New York, NY, USA), pp. 187–197, ACM, 2006.
19. W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "Making nested parallel transactions practical using lightweight hardware support," in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 61–71, ACM, 2010.
20. Y. Liu, S. Diestelhorst, and M. Spear, "Delegation and nesting in best-effort hardware transactional memory," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 38–47, ACM, 2012.
21. 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.
22. J. a. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Sci. Comput. Program.*, vol. 63, pp. 172–185, Dec. 2006.
23. A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pp. 155–165, ACM, 2009.