NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems

Haris Volos[1], Adam Welc[2], Ali-Reza Adl-Tabatabai[2], Tatiana Shpeisman[2], Xinmin Tian[2], and Ravi Narayanaswamy[2]

[1]University of Wisconsin
Madison, WI 53706

[2]Intel Corporation
Santa Clara, CA 95054

hvolos@cs.wisc.edu
{adam.welc,ali-reza.adl-tabatabai,tatiana.shpeisman,
xinmin.tian,ravi.naraynaswamy}@intel.com

Abstract. Transactional memory (TM) promises to simplify construction of parallel applications by allowing programmers to reason about interactions between concurrently executing code fragments in terms of high-level properties they should possess. However, all currently existing TM systems deliver on this promise only partially by disallowing parallel execution of computations performed inside transactions. This paper fills in that gap by introducing NePaLTM (**N**ested **PA**ralle**L**ism for **T**ransactional **M**emory), the first TM system supporting nested parallelism inside transactions. We describe a programming model where TM constructs (atomic blocks) are integrated with OpenMP constructs enabling nested parallelism. We also discuss the design and implementation of a working prototype where atomic blocks can be used for concurrency control at an arbitrary level of nested parallelism. Finally, we present a performance evaluation of our system by comparing transactions-based concurrency control mechanism for nested parallel computations with a mechanism already provided by OpenMP based on mutual exclusion.

1 Introduction

As the microprocessor industry transitions to multithreaded and multicore chips, programmers must use multiple threads to obtain the full performance of the underlying platform [24]. Transactional memory (TM), first proposed by Herlihy and Moss [13], has recently regained interest in both industry and academia [9–11, 18, 19] as a mechanism that seeks to simplify multithreaded programming by removing the need for explicit locks. Instead, a programmer can declare a section of code *atomic* which the TM system executes as a transaction; its operations execute atomically (i.e. all or nothing) and in isolation with respect to operations executed inside other transactions. While transactions appear to execute in some sequential order, their actual execution may overlap increasing the degree of available parallelism.

However as the number of processors increases, by Amdahl's law [3], the single transaction may become the sequential bottleneck hindering speedup achieved via parallelism. *Transactional nested parallelism*, that is the ability to use multiple threads inside a transaction, proves to be useful in removing this bottleneck. For example, resizing of a concurrent data structure constitutes a relatively long-lasting and heavyweight operation which nevertheless must be executed transactionally to prevent interference with other transactions concurrently accessing the same data structure. By parallelizing the resize operation within a transaction, we can still guarantee non-interference but without letting the sequential resize operation adversely affect overall performance.

Transactions are also meant to compose better than locks. Programmers should be able to integrate arbitrary library code into their own concurrent applications without fear of deadlock or unpredictable performance loss, regardless of how concurrency is managed inside the library. The existing TM systems deliver on this composability promise only partially as they do not support nested parallelism inside transactions and thus transactional code cannot take advantage of efficient parallel implementations of common algorithms, even if they are readily available in a packaged library form.

At the same time, dynamic (implicit) multithreading provided by languages such as Cilk [23] or libraries such as OpenMP [22] is becoming a widely used and efficient method of introducing parallelism into applications. An application programmer expresses the parallelism by identifying elements that can safely execute in parallel, and letting the runtime system decide dynamically how to distribute work among threads. Most of the systems supporting dynamic multithreading is based on the *fork-join* concurrency model which is simple to reason with and yet has great expressive power. For example, an important class of problems can be solved using the divide-and-conquer technique which maps well to the fork-join model: a problem is broken into sub-problems, and then these sub-problems can be solved independently by multiple threads whose partial computation results are ultimately combined into a complete problem solution. The parallel computation of the sub-problems can often proceed with little or no internal synchronization.

Despite the growing significance of dynamic multithreading, only few researchers have previously explored issues related to integration of TM constructs into the fork-join concurrency model. In particular, Agrawal et al. describe a high-level design for supporting nested parallelism inside transactions in the context of Cilk [2]. However, similarly to the first published design of a system supporting transactional nested parallelism (in a context of persistent programming languages) by Wing et al. [28], they provide neither implementation nor performance evaluation of their design. Integration of TM constructs into OpenMP has been explored by Baek et al. [4] and Milovanović et al. [16] but neither of these solutions allows nested parallelism inside transactions.

Our paper makes the following contributions:

1. We describe a programming model for a system where OpenMP's constructs enabling nested parallelism can be nested inside TM constructs used for

concurrency control (atomic blocks) (Section 3). Our programming model defines an execution model which is a logical extension of an existing transactional execution model to the case of nested parallelism.
2. We describe the design (Section 4) and implementation (Section 5) of the first TM system, NePaLTM (**N**ested **PA**ralle**L**ism for **T**ransactional **M**emory), where atomic blocks can be used for concurrency control at an arbitrary level of nested parallelism. We discuss in detail extensions and modifications to the existing TM mechanisms required to support atomic blocks in presence of nested parallelism.
3. We evaluate performance of our system by comparing transactions-based concurrency control mechanism for nested parallel computations with a mechanism already provided by OpenMP based on mutual exclusion, and demonstrate that the performance of the former is in many cases superior to the latter. (Section 6).

2 Background

Before diving into details of our programming model and describing NePaLTM's design and implementation, we would like to provide some relevant background information on both TM-style concurrency control and OpenMP-style fork-join programming model.

2.1 C/C++ Software Transactional Memory

Intel's Software Transactional Memory (STM) system, extending C/C++ with a set of TM constructs, forms our base TM system [19]. The `__tm_atomic` statement is used to define an *atomic block* which executes as a transaction; its operations execute atomically (i.e. all or nothing) and in isolation with respect to operations executed inside other transactions. The `__tm_abort` statement (user abort) allows a programmer to explicitly abort an atomic block. This statement can only appear in the lexical scope of an atomic block. When a user abort is triggered, the TM system rolls back all side effects of the atomic block and transfers control to the statement immediately following the block.

The TM system provides SLA (Single Lock Atomicity) [14] semantics; atomic blocks behave as if they were protected by a single global lock. This guarantees that programs that are race free under a single global lock will execute correctly when executed transactionally. Providing no guarantees for programs containing data races[1] is consistent with the emerging C/C++ memory model specification [7]. We next give an overview of the base system's structure [19].

The base system performs updates in-place with strict two-phase locking for writes, and supports both optimistic and pessimistic concurrency control for reads. The system keeps a *descriptor* structure per transaction which encapsulates the transaction's *context* (i.e. meta-data such as transactional logs). The

[1] A data race occurs when multiple threads access the same piece of memory, and at least one of those accesses is a write.

system also keeps a table of *transaction records* called the *ownership table*. Every memory address is hashed to a unique transaction record in this table but multiple addresses may be hashed to the same record. A transaction record contains information used by the concurrency control algorithm to control access to memory addresses mapped to this record. When a transaction record is write-locked, it contains information about the single lock owner. When a transaction record is read-locked, it contains information about all transactions holding read-locks for a given location. Additionally, when a transaction record is not write-locked, it contains a version timestamp used by optimistic readers as explained below.

Transactional memory accesses are performed through three types of transactional barriers: *write*, *optimistic read* and *pessimistic read* barriers. On a transactional store, the write barrier tries to exclusively write-lock the transaction record. If the record is locked by another transaction, the runtime resolves the conflict before continuing, which may abort the current transaction. If it is unlocked, the barrier write-locks the record, records the old value and the address in its *undo log*, adds the record to its *write log* (which keeps the transaction's *write set*), and then updates the memory location. On a transactional load, the optimistic read barrier checks if the transaction record is locked, but does not try to lock it. In contrast, the pessimistic read barrier tries to read-lock it. In both cases, if the record is write-locked by another transaction, the conflict is handled. If it is unlocked or read-locked, both optimistic and pessimistic read barriers return the value of the location and add the record to the *read log* (which keeps the *read set*). The optimistic read barrier also records the transaction record's timestamp, used to keep track of when the memory location is being updated.

On commit, an optimistic transaction uses the recorded timestamps to validate that no transaction record in its read set has been updated after the transaction read them. If validation succeeds, the transaction unlocks all transaction records in its write set; otherwise it aborts. A pessimistic transaction does not need to validate its read set but does need to unlock all transaction records in both its read and write set. On abort, in addition to all locks being released, the old values recorded in the undo log are written back to the corresponding addresses. On both commit and abort, the runtime modifies the timestamps of the updated locations – subsequent timestamp values are obtained by incrementing a global counter.

To provide SLA semantics correctly, the runtime guarantees several important safety properties, namely granular safety, privatization safety and observable consistency [15]. For granular safety, the runtime records transactional data accesses into the undo log at an appropriate granularity level – when accessing N (= 1, 2, 4 or 8) bytes of data, the runtime must be careful to record and restore only these N bytes without affecting memory adjacent to the location where the data is stored. Privatization safety and observable consistency are an issue only with optimistic transactions. Privatization [21] is a common programming idiom where a thread privatizes a shared object inside a critical section, then continues accessing the object outside the critical section. Privatization, if not supported correctly, can cause incorrect behavior in the following way: a committing priva-

tizer may implicitly abort a conflicting optimistic transaction due to an update resulting from its privatization action, and subsequent non-transactional code may read locations that were speculatively modified by the conflicting transaction, which has yet to abort and roll back. The system provides privatization safety through a quiescence algorithm [26]. Under this algorithm a committing transaction waits until all other optimistic transactions verify that their read set does not overlap with the committing transaction's write set. Observable consistency guarantees that a transaction observes side effects only if it is based upon a consistent view of memory. In other words, a transactional operation (a read or a write) is valid in the sense of observable consistency if it is executed under some consistent memory snapshot, even if that operation needs to be subsequently undone. The runtime provides this by having each transaction validate its read set before accessing any location written by a transaction that has committed since the previous validation of the read set.

2.2 OpenMP API

The OpenMP API is a collection of compiler directives, runtime library routines, and environment variables that can be used to explicitly control shared-memory parallelism in C/C++ [22]. We next give an overview of the features of version 2.5 of the OpenMP specification [5] which are relevant to this work. At the point of writing this paper we had no access to an implementation supporting the new features available in version 3.0 [6] such as OpenMP tasks so we defer exploration of these new features to future work.

Parallel regions OpenMP's fundamental construct for specifying parallel computation is the `parallel` pragma. OpenMP uses the *fork-join* model of parallel execution. An OpenMP program begins as a single thread of execution, called the *initial thread*. When a thread encounters the `parallel` pragma, it creates a *thread team* that consists of itself and zero or more additional threads, and becomes the *master* of the new team. Then each thread of the team executes the parallel region specified by this pragma. Upon exiting the parallel construct, all the threads in the team join the master at an implicit barrier, after which only the master thread continues execution. The `parallel` pragma supports two types of variables within the parallel region: shared and private. Variables default to *shared* which means shared among all threads in a parallel region. A *private* variable has a separate copy for every thread.

Work-sharing All of a team's threads replicate the execution of the same code unless a work-sharing directive is specified within the parallel region. The specification defines constructs for both iterative (`for`) and non-iterative (`sections`, `single`) code patterns. The `for` pragma may be used to distribute iterations of a for loop among a team's threads. The `sections` pragma specifies a work-sharing construct that contains a set of structured blocks defined using the `section` pragma that are to be divided among and executed by the threads in a team.

Each structured block is executed once by one of the threads in the team. The `single` pragma specifies that the associated code block is executed by only one thread in the team. The rest of the threads in the team do not execute the block but wait at an implicit barrier at the end of the single construct unless a `no wait` clause is specified.

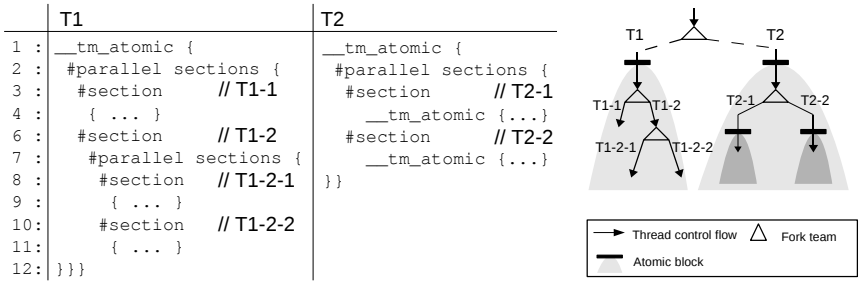
Synchronization Synchronization constructs control how the execution of each thread proceeds relative to other team threads. The `atomic` pragma is used to guarantee that a specific memory location is updated atomically. A more general synchronization mechanism is provided by the `critical` pragma used to specify a block of code called a *critical region*. A critical region may be associated with a name, and all anonymous critical regions are assumed to have the same unspecified name. Only one thread at a time is allowed to execute any of the critical regions with the same name. In addition to the implicit barriers required by the OpenMP specification at certain points (such as the end of a parallel region), OpenMP provides the `barrier` pragma which can be used to introduce explicit barriers at the point the pragma appears; team threads cannot proceed beyond the barrier until all of the team's members arrive at the barrier. The specification does not allow nesting of `barrier` pragma inside a critical region.

3 Programming model

The programming model we present in this section allows atomic blocks to benefit from OpenMP and vice versa. Transactions can use OpenMP's parallel regions to reduce their completion time and OpenMP can use transactions to synchronize access to shared data. We chose OpenMP because of its industry-wide acceptance as a method for programming shared memory, as well as because of the simplicity and expressive power of the fork-join execution model that OpenMP is based on. However, nothing prevents transactional nested parallelism to be supported in an alternative setting, exemplified by systems using explicit threading models.

3.1 Constructs

Our programming model adds TM's atomic block construct to the existing, more traditional, synchronization constructs specified by OpenMP (e.g. critical regions). Simultaneous use of these constructs is legal as long as they are used to synchronize accesses to disjoint sets of data. Previous work, exploring composition of the traditional synchronization constructs with atomic blocks, has shown that such composition is non-trivial [25, 27, 29], and, as such, is beyond the scope of this paper. Our programming model also supports OpenMP's `barrier` pragma for declaring synchronization barriers, but like the original OpenMP specification which does not allow the use of this pragma inside critical regions, we do not allow its use inside atomic blocks.



(a) Code (b) Control flow tree

Fig. 1. Shallow (T1) and deep (T2) nesting

An atomic block is orthogonal to an OpenMP's parallel region. Thus an atomic block may be nested inside a parallel region and vice versa. When an atomic block is nested inside a parallel region, each dynamic instance of the atomic block is executed by a single thread of that region. In the opposite case when a parallel region is nested inside an atomic block, a team of threads is created and all the threads execute under the same atomic block on behalf of the same transaction. We refer to the transitive closure of the threads created under the same atomic block as *atomic thread team* (or *atomic team* for short). When a user abort is triggered by a member of an atomic team using the `__tm_abort` construct, all the team's threads abort their computation and the entire transaction aborts.

An atomic block is also orthogonal to OpenMP work sharing constructs, with only one exception. While an atomic block can be nested inside a `single` construct, the opposite is not true. Recall from Section 2.2 that all team threads but the one executing the `single` region wait at an implicit barrier at the end of the region. If a `single` pragma was allowed to be nested inside an atomic block then it would be possible for the threads waiting at the barrier to transactionally hold resources needed by the thread executing the `single` region resulting in a deadlock. To prevent such a case, we disallow `single` from being nested inside of an atomic block. Note that this is not different from the original OpenMP specification which prevents nesting of a `single` pragma inside a critical region.

Before moving on with the description of the execution model, we need to introduce some more terminology. We call a thread that begins an outermost atomic block a *root thread*. We reason about the hierarchy between threads in terms of a parent-child relation; a thread spawning some threads becomes the *parent* of these threads (and the *ancestor* of these and all other transitively created threads), the spawned threads become its *children* and one another's *siblings*. Conceptually, execution of the parent thread is suspended at the spawn point and resumed when all children complete their execution. The *transactional parent* of a child thread is its ancestor thread that created an atomic block immediately enclosing the point of the child thread's creation. Atomic blocks form a nesting hierarchy. We refer to the atomic block of a root thread as a *root*

atomic block, and to an atomic block created by a nested thread as a *parallel-nested atomic block*. When the threads spawned under a root atomic block do not use any additional atomic blocks, we have *shallow nesting*. If however these threads do use additional atomic blocks then we have *deep nesting*. For example, Figure 1 [2] illustrates the control flow tree for a given code block. Threads `T1` and `T2` are root threads. Thread `T1` is the parent of thread `T1-2` and `T1-2` is the parent of `T1-2-2`. `T1` is both the ancestor and the transactional parent of `T1-2` and `T1-2-2`. The atomic blocks created by threads `T1` and `T2` are root atomic blocks while the atomic blocks created by threads `T2-1` and `T2-2` are parallel-nested atomic blocks. Threads `T1-1`, `T1-2`, `T1-2-1`, `T1-2-2` are part of the same atomic team. Finally, the tree with root `T1` represents a case of shallow nesting and the tree with root `T2` represents a case of deep nesting.

3.2 Execution Model

Recall from section 2.1 that our base TM model provides the SLA (Single Lock Atomicity) semantics for race free programs; atomic blocks behave as if they were protected by a single global abstract [3] lock. However in the presence of nested parallelism this model is insufficient. To see why, consider again Figure 1; if a single abstract lock was used by all atomic blocks, then threads `T2-1` and `T2-2` would block-wait for their parent, thread `T2`, to release the abstract lock protecting its atomic block resulting in a deadlock.

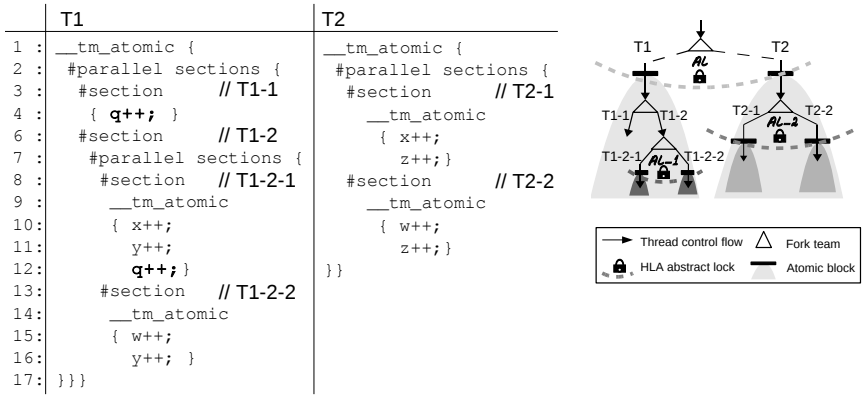
Our programming model logically extends the SLA execution model into the *HLA (Hierarchical Lock Atomicity)* model to account for nested parallelism. Like SLA, HLA defines semantics for race free programs. HLA is similar to the model used by Moore and Grossman in their formal definition of small-step operational semantics for transactions [17]. In HLA, abstract locks protecting atomic blocks form a hierarchy; a "fresh" abstract lock is used whenever a child thread starts a new atomic block, and it is used for synchronizing data accesses between this thread and threads that have the same transactional parent. Note how in the case of shallow nesting HLA degrades to SLA; only a single abstract lock is required to maintain concurrency control between all atomic blocks.

HLA semantics differs from the semantics of OpenMP's critical regions in that critical regions with the same name are not re-entrant. This implies that if we hierarchically nest critical regions in the same fashion as atomic blocks, we end up with a non-recoverable deadlock.

To better understand HLA consider the example given in Figure 2 which extends the example in Figure 1. In contrast to the previous example, threads `T1-2-1` and `T1-2-2` create new atomic blocks which are nested under the atomic block of thread *T1*. Let's first consider how abstract locks are assigned to the atomic blocks according to HLA. The root atomic blocks of threads `T1` and `T2`

[2] For readability we abbreviate OpenMP pragmas in all figures by omitting the initial `pragma omp`.

[3] We call this and other locks abstract because locks do not have to be used to enforce a semantics, even if this semantics is expressed in terms of locks.



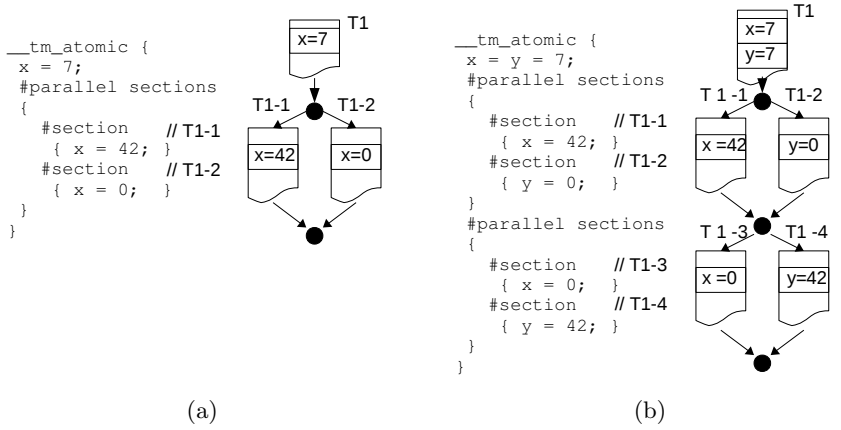
(a) Code (b) Control flow tree

Fig. 2. HLA Example

are assigned abstract lock *AL*, atomic blocks of threads `T1-2-1` and `T1-2-2` are assigned lock *AL-1*, and atomic blocks of threads `T2-1` and `T2-2` are assigned lock *AL-2*. Too see how these abstract locks are used to synchronize data accesses consider the accesses of threads `T2-1` and `T2-2`. Accesses to `x` and `w` by `T2-1` and `T2-2` respectively are isolated from the accesses of `T1-2-1` and `T1-2-2` using lock *AL*. Accesses to `z` by `T2-1` and `T2-2` are isolated from each other using lock *AL-2*. Similarly, accesses to `y` by `T1-2-1` and `T1-2-2` are isolated from each other using lock *AL-1*. Finally consider the accesses to `q` by threads `T1-1` and `T1-2-1`. Since the two threads do not synchronize their accesses to `q` through the same lock, these accesses are not correctly synchronized and therefore they are racy.

4 Design

The HLA semantics can be supported through two different types of concurrency mechanisms: transactions and mutual exclusion locks. Our design choice is to use transactions for concurrency control between root atomic blocks and mutual exclusion locks for parallel-nested atomic blocks. This choice is motivated by the following three observations. First, it has been demonstrated that transactions scale competitively to locks or better [1, 11, 12, 19]. Thus our system offers root atomic blocks that can execute efficiently as transactions and which can use transactional nested parallelism to further accelerate their execution in case shallow nesting is used. Second, by supporting deep nesting through locks, our system provides composability, which is a very important property for the adoption of the model. Third, while we considered supporting parallel-nested atomic blocks using transactions, previous work by Agrawal et al. [2] has shown that such a design is complex and its efficient implementation appears to be questionable. As we discuss at the end of this section, our personal experience on the subject is very similar.



(a) (b)

Fig. 3. Examples of transactional logs in the presence of shallow nesting. Arrows depict ordering constraints.

4.1 Shallow Nesting

In the case of shallow nesting no transactional synchronization is enforced between the members of an atomic team. Nevertheless, because operations of all the team members are executed on behalf of a single transaction, they must appear to be executed as a single atomic unit and in isolation from other concurrent root transactions. To achieve this, atomic team members inherit the transactional context of their transactional parent and perform all transactional operations using that context. Having multiple threads working on behalf of a single transaction has several important implications on how the runtime manages transactional logs and how it guarantees transactional safety properties. Below we describe these implications in detail:

Logging Recall that there are three types of logs associated with a transactional context: read, write, and undo log. The read and write logs track transaction's read and write sets, respectively, and the undo log keeps the old values of the locations written by the transaction. Conceptually members of an atomic team and their parent share the same log, so a simple solution would be to have threads use a single log instance and synchronize access to that log. However this would require excessive synchronization making this solution impractical. Instead of using a single log and paying the cost of synchronizing log accesses, we leverage several properties described below so as to allow each atomic team member to maintain its own private instances of transactional logs.

Write log A transaction's write log is used to release write locks when the transaction completes. Because locks can be released in any order, they can be freely distributed between multiple logs. A potential problem of using multiple logs is double-releasing a write lock if the same lock appears in more than one log.

However, since children acquire locks on behalf of their parent, at most one child will find the lock not held and bring it into its log.

Read log In the pessimistic case, a transaction's read log is used to release read locks, and therefore the correctness reasoning is the same as in the write log above. In the optimistic case, the read log is used for validation, but the ordering with which validation is done is not important either. Moreover, since no locks are released, the read log can tolerate multiple entries per location.

Undo log In contrast to read and write logs, ordering of undo log entries matters because side effects must be rolled back in the opposite order to that in which they happened. While undo entries do not need to be totally ordered, undo entries for the same memory location must be partially ordered. There are two cases of writes to the same memory location that we need to consider. First, simultaneous writes by multiple threads of the same atomic team may generate multiple undo entries. Since these writes constitute a race, and according to our programming model racy programs have undefined behavior, their undo entries need not be ordered. Figure 3(a) shows an example where two threads of the same atomic team both write `x`; ordering of the undo entries is not important because the writes are racy. In the second case, writes of a memory location by threads in different atomic teams may also generate multiple undo entries. However, if the writes are performed in different parallel regions executed one after the other then they are ordered and therefore they are not racy. For example in Figure 3(b) writes to `x` and `y` by threads `T1-1, T1-3` and `T1-2, T1-4` respectively are partially ordered. In order for undo entries distributed between multiple logs to be recorded correctly, atomic team members merge their logs with that of their parent at join points, and the parent records the ordering.

Safety Properties NePaLTM must guarantee the three safety properties discussed in Section 2, namely granular safety, privatization safety, and observable consistency. For granular safety no additional support is necessary because granular safety depends only on undo logging granularity which is not affected by having multiple threads working on behalf of a single transaction. However, having multiple threads does have implications on the mechanisms used to provide observable consistency and privatization safety. For observable consistency, children must not only validate their private read sets but they must also validate the read set of their (suspended) parent. This is because their computations depend on their parent's computation and therefore they must have a view of memory which is consistent with that of their parent. For privatization safety, a committing transaction waits separately for the atomic team members to validate and become stable rather than waiting for validation of the root thread's transaction as a whole which can only happen after execution of all the team members is completed and could thus be sub-optimal.

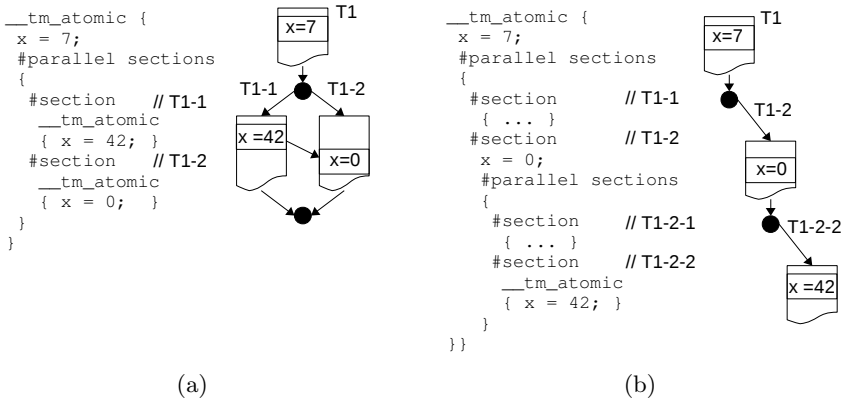


Fig. 4. Examples transactional logs in the presence of deep nesting. Arrows depict ordering constraints.

4.2 Deep Nesting

NePaLTM supports deeply nested atomic blocks using mutual exclusion locks. As defined by the HLA semantics presented in Section 3.2, a fresh mutual exclusion lock is allocated per atomic team and used for concurrency control between atomic blocks created by the atomic team's threads.

Despite using mutual exclusion for concurrency control in case of deep nesting, all the code executed by deeply nested atomic blocks must be *transactionalized*, that is instrumented to support transactional execution. First, transactional instrumentation of memory accesses is necessary to be able to roll back side effects in case the atomic block needs to abort. Second, transactional concurrency control must still be used for synchronizing memory accesses performed by threads inside a root atomic block with memory accesses done by threads executing inside other root atomic blocks. We now present a discussion of how deep nesting impacts logging and safety properties.

Logging In Section 4.1 we reasoned about the correctness of our design decision to let children use private instances of transactional logs. Deep nesting, however, adds the additional ordering constraint that logs must respect the order enforced by parallel-nested atomic blocks. This is particularly important in the case of undo logs since undo operations must be executed in order opposite to that of transaction commits. A child committing an atomic block, in order to correctly capture the commit order of this atomic block with respect to other atomic blocks executed under the same parent, must merge its current private log with the log of its transactional parent before releasing the mutual exclusion lock guarding its atomic block. If "intermediate" threads have been forked between a child's transactional parent and the child itself then log merging must respect the order implied by the fork operations in case these intermediate threads performed data

accesses on their own. To accomplish this a child must transitively merge logs up the parent/child hierarchy until it reaches its transactional parent.

Figure 4 shows two examples of deeply nested transactional logs. In the example presented in Figure 4(a), thread `T1-2` commits its atomic block after `T1-1` does so. Log merging must respect the commit ordering of the two atomic blocks as shown by the arrow connecting the two logs. In the example presented in Figure 4(b), thread `T1-2-2` commits a deeply nested atomic block with an intermediate thread `T1-2` forked between the time when thread `T1-2-2` has been created and the time when thread `T1-2-2`'s transactional parent `T1` has started its transaction. To capture the fork order shown by the arrows, `T1-2-2`'s log must be first merged with `T1-2`'s log and then the resulted log must be merged together with `T1`'s log.

Safety Properties Since parallel-nested atomic blocks use mutual-exclusion locks to handle synchronization, no additional support is necessary to guarantee the safety properties for these atomic blocks.

4.3 Discussion

We initially considered an alternative design where transactional concurrency control is used at all nesting levels. However we eventually abandoned it in favor of the one we described above for the reasons we discuss here. When using transactions at all nesting levels, as described by Agrawal et al. [2], the parent/child relation plays a very important role in ensuring correctness of data access operations. Maintenance and querying of the structure representing this relation is likely to significantly complicate the implementation and decrease its efficiency. Moreover, supporting optimistic transactions further complicates the algorithms used for guaranteeing privatization safety and observable consistency between atomic blocks at all nesting levels.

5 Implementation

Our prototype implementation follows our design guidelines and supports HLA via transactions for root atomic blocks and via mutual-exclusion locks for deeper nesting levels. As a base for our implementation we used Intel's STM runtime library and Intel's TM C/C++ compiler [19], as well as Intel's implementation of the OpenMP library supporting version 2.5 of the OpenMP specification. Despite a somewhat simplified design for deep nesting, significant extensions and modifications to the algorithms of the base TM runtime were required. Additionally we needed to modify the compiler and the OpenMP runtime library to correctly compile and execute OpenMP constructs that are nested inside atomic blocks.

In the remainder of this section we describe the data structures used in NePaLTM as well as the mechanisms that implement concurrency control in the presence of nested parallelism, with emphasis on the implementation of the

abort procedure and the transactional logs. We finally discuss the required modifications to the TM compiler to support nested parallelism.

5.1 Concurrency Control

Execution Modes and Descriptor We introduce two new execution modes to the base TM system, namely *omp_optimistic* and *omp_pessimistic* which are used by atomic team members working on behalf of an optimistic or pessimistic transaction respectively. The base TM system supports multiple execution modes through a layer of indirection similar to a vtable. Each execution mode defines its own dispatch table of pointers to functions that implement the transactional read/write barriers and transaction begin/commit/abort routines specific to that mode. At runtime, the dispatch table of the current execution mode is used to indirectly call that mode's routines. This mechanism allows us to incrementally pay the overheads associated with nested parallelism. A transaction always starts at a base execution mode (e.g. optimistic) where it uses the original barrier implementations. Then, when it spawns an atomic team, its children transition into one of the two new execution modes where they use the nested-parallelism-aware barriers and begin/commit/abort routines.

In NePaLTM, similarly to the base system, every thread in the system (both root and child threads) is associated with a *descriptor* which is stored in the thread's local storage and encapsulates the thread's transactional context. We have extended the descriptor structure to keep some additional state related to nested parallelism. In particular: information to reconstruct the parent/child hierarchy such as pointers to the descriptor of the parent and the transactional parent, state used by the recursive abort mechanism described later, and a mutual exclusion lock for use by parallel-nested atomic blocks under this transaction.

To correctly switch execution mode and construct the parent/child hierarchy we extended the OpenMP runtime library functions responsible for spawning and collecting team threads with callbacks to the TM runtime.

Transactional Barriers We have implemented new transactional read and write barriers for instrumenting memory accesses performed by team members that run at one of the two new execution modes (*omp_optimistic* and *omp_pessimistic*). Our new barriers are based on the original ones which we have extended to support nested-parallelism.

On a transactional store, the write barrier executed by transactional thread tries to exclusively lock the transaction record associated with the memory location. If it is already locked by this thread's root transaction then the thread proceeds without acquiring the lock. However the barrier must still record the old value and the address in its own private undo log[4]. If the record is locked by another root transaction, the runtime resolves the conflict before continuing, which may abort the current transaction. If the transaction record is unlocked

[4] This is necessary because the ownership table is implemented as a hash table where multiple memory locations may be hashed on the same transaction record.

then the barrier locks it using the current thread's root transaction descriptor, adds the record to its own private write log, records the old value and address in its private undo log, and then updates the memory location.

On an optimistic transactional load, the optimistic read barrier checks if the transaction record is already locked but does not try to lock it. If the record is write-locked by another root transaction, the conflict is handled. If it is unlocked, locked by the current thread's root transaction or read-locked by another root transaction, the barrier returns the value of the location, adds the record into its read log and records the transaction record's timestamp.

On a pessimistic transactional load, the pessimistic read barrier tries to read-lock the transaction record. If the record is write-locked by another root transaction, the conflict is handled. If it is already read-locked or write-locked by the current thread's root transaction then the barrier returns the value of the location without acquiring the lock. If it is unlocked or read-locked by another root transaction then the barrier read-locks the transaction record using the current thread's root transaction descriptor, adds the lock into its own private read log, and returns the value of the location

Observable Consistency and Privatization Safety For observable consistency, a thread executing inside of an optimistic transaction validates the read sets of all of its ancestors, up to and including the root transaction, in addition to its own read set. As an optimization, when a thread validates the read set of an ancestor, it updates the last validation timestamp of that ancestor so that other threads with the same ancestor do not need to re-validate that read set. For privatization safety, we modified the base quiescence algorithm so that a parent thread temporarily removes itself from the list of in-flight transactions until its child threads complete execution. By removing itself from that list, other committing transactions do not have to wait for that thread. This is safe because those other committing transactions will wait for that thread's children instead.

Transaction Commit and Abort Transaction begin, commit and abort procedures executed by atomic team members are also different than those executed by root threads. Since deep nesting is implemented using locks, the routines do not implement a full-fledged begin and commit. Instead they simply acquire and release the mutual exclusion lock of the transactional parent at entry and exit to a parallel-nested atomic block. Additionally, since parents must access transactional logs of its children, as described in Section 4, children must pass their logs to their parents at commit. The implementation details of this procedure are described in Section 5.2.

NePaLTM supports a *recursive abort* mechanism, which can be used by any active thread in the parent/child hierarchy to trigger an abort of the whole computation executing under a transaction. Our extension to the OpenMP library implementing this mechanism keeps some additional state: a per atomic team *abort flag* stored in the transactional descriptor of every parent, and *internal*

checkpoint of the stack/register context for each atomic team's thread, taken before the thread starts its computation. Please note that in the presence of transactional nested parallelism it is no longer sufficient to record a checkpoint only at transaction begin – restoration of a child thread's state to the checkpoint taken by the parent would be incorrect.

An atomic team member triggers an abort of the entire transaction by setting its parent's abort flag and then aborting its current computation. An atomic team member detects abort requests by recursively checking the abort flag of all of its ancestors up to the root thread. All these checks happen either in the slow path of the transactional barriers or when a thread waits behind the lock protecting a parallel-nested atomic block. When a team member detects an abort request, it aborts its current computation by marking its state as completed and then restoring its internal checkpoint. After restoring its checkpoint, the thread appears to OpenMP as completed, and OpenMP can safely shutdown and recycle that thread. When execution of all the child threads is completed, execution of their parent thread is resumed. When the parent thread resumes, it checks for a pending abort request; if the thread itself is a child then the abort is propagated up the parent/child hierarchy.

We support aborts requested both implicitly through a TM conflict and explicitly through the `__tm_abort` statement (user-level abort). The abort flag carries the reason for the abort giving priority to conflict aborts. The `__tm_abort` statement in our system is required to be lexically scoped within an atomic block and, when executed, it aborts this atomic block only. As a result, abort propagation stops upon reaching the atomic block lexically enclosing the `__tm_abort` statement.

Admittedly, there exist alternative implementations of the abort mechanism, but we believe that our implementation achieves a good balance between efficiency and required modifications to an already complex and carefully tuned OpenMP runtime library.

5.2 Transactional Logs

As explained in the design section, each child thread uses private transactional logs and whenever ordering information needs to be captured, a child merges its private log together with the log of its parent. We have implemented log merging using a hybrid algorithm that combines two methods: *concatenation* and *copy-back*. Whenever we need to merge two logs together, our hybrid algorithm calculates the log size of the source log to be merged. If the log size is above what we call a *concatenation threshold*, the log is concatenated; otherwise, the log is copied-back. We next describe *concatenation* and *copy-back* separately.

Concatenation In this method, a transactional log is composed of several log fragments linked together to form a list. Similarly to logs in the base TM system, log fragments keep pointers to potentially multiple log buffers. A source log is merged with a target log by connecting the log fragment list of the former at

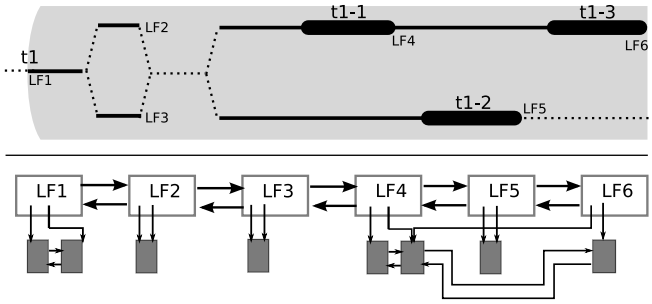


Fig. 5. Representation of the log as a list of log fragments.

the tail of the list of the latter. For example, in Figure 5, log fragment `LF1` (log fragments are represented by empty grey rectangles) associated with the root thread executing transaction `t1` has all the log fragments that belong to its children linked to it.

A log is split into log fragments only when necessary. Thus if nested parallelism is not used inside atomic blocks, a log is represented by a single log fragment. However, when nested parallelism is used, we only utilize log fragments at points where we need to establish correct log ordering as previously described. There are two cases in which this may happen. First, in the case when a child thread reaches its join point, its parent connects the log fragment currently used by the child to its own list of log fragments. Then the child is assigned a fresh log with a new buffer. A fresh log is necessary because the child thread may be recycled by OpenMP and used by another parent. In Figure 5, `LF2` and `LF3` represent log fragments passed from children to their parent at the join point.

Second, in the case when a child commits a nested atomic block, it splits its log and then connects the produced log fragment to the parent's log fragment list. If ancestor threads exist between the child and the root thread, the child splits the log fragments of all ancestor threads and connects them recursively until reaching the root thread. Since only one child can acquire an atomic team's lock at any given time, this child can safely access and modify logs of all its ancestors. Returning to our example in Figure 5 (atomic blocks created by children are represented by narrow rounded black rectangles), the child thread at the top of the figure splits its log when committing its first atomic block (`t1-1`) to create log fragment `LF4`. Then the child thread at the bottom of the figure splits when committing its own atomic block (`t1-2`) to create log fragment `LF5`. Finally the last log fragment, `LF6`, is created when the child thread at the top of the figure commits its second atomic block (`t1-3`). In contrast to thread join where we assign new buffers to children because they complete their execution, the committing child is still active so it keeps using its assigned buffers continuing from the last buffer entries used by its log fragments. As a result, a buffer can be shared between multiple log fragments as in the case of log fragments `LF4` and

LF6 in Figure 5 (log buffers are represented as solid grey rectangles). Naturally, a transaction using a given log fragment may fill up a buffer initially associated with this log fragment and attach a new buffer (e.g. log fragments LF1 and LF4 in Figure 5).

Copy-back In this method two transactional logs are merged together by simply copying every single entry of the source log to the target log at appropriate points, the same in fact as the points when the logs are concatenated as described above. It is the parent that copies entries from the child's log into its own when the child has already reached the join point. It is the child that copies entries from its own log to that of the parent whenever that child commits a nested atomic block.

5.3 TM compiler

The most important compiler-related issue was to ensure that the code generated for OpenMP's parallel region executed inside transactions is appropriately transactionalized, as described in Section 4.2. The TM compiler automatically transactionalizes function calls (direct and indirect) that are annotated with a special `tm_callable` attribute. However, OpenMP's parallel region is specified as a code block at the source level and packaged into a form of a function internally by the compiler. As a result it could not be transactionalized automatically and the compiler needed to be modified to do it explicitly. Also, certain OpenMP functions are called as part of setting up the execution environment of a parallel region. The TM compiler had to be modified to treat these functions as transactional intrinsics (similarly to calls to the TM runtime library) otherwise they would be treated as non-transactional functions of unknown origin. For safety reasons, a transaction encountering a call to a non-transactional function transitions to the so called *serial mode* which allows only a single transaction to execute in the system. Clearly this would defeat the purpose of introducing nested parallelism in the first place.

6 Performance Evaluation

Our performance evaluation focuses on evaluating our design decisions of using transactions at the outermost level and locks at the deep nested levels. Because benchmarks exercising nested parallelism are not easily available, we evaluate performance of our prototype implementation using an in-house implementation of the multi-threaded OO7 benchmark [8]. This benchmark is highly configurable so it allows us to study the behavior of our system under different scenarios.

We seek to answer the following two questions:

1. Can transactions retain their performance advantage over locks in presence of nested parallelism?
2. How does performance of parallel-nested atomic blocks compare with performance of the atomic blocks executing the same workloads sequentially?

6.1 Benchmark Description

OO7 is a highly configurable benchmark that has been previously used in several TM-related studies [1, 27, 29]. The benchmark is also easy to port and modify, which was an important factor since the previous TM-enabled version of this benchmark was written in Java, was not OpenMP-enabled and did not support nested parallelism.

The OO7 benchmark operates on a synthetic design database consisting of a set of composite parts. Each composite part consists of a graph of atomic parts. Composite parts are arranged in a multi-level assembly hierarchy, called a module. Assemblies are divided into two classes: base assemblies (containing composite parts) and complex assemblies (containing other assemblies).

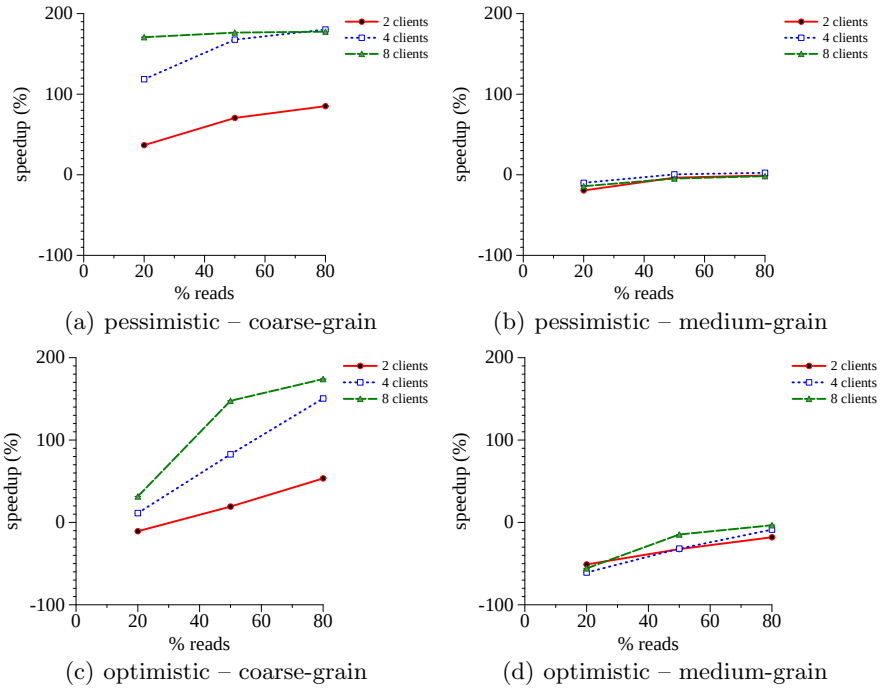
The multi-threaded workload consists of multiple *client threads* running a set of parameterized traversals composed of primitive operations. A traversal chooses a single path through the assembly hierarchy and at the base assembly level randomly chooses a fixed number of composite parts to visit. When the traversal reaches the composite part, it has two choices: (a) it may access atomic parts in the read-only mode; or, (b) it may swap certain scalar fields in each atomic part visited. To foster some degree of interesting interleaving and contention, the benchmark defines a parameter that allows additional work to be added to read operations to increase the time spent performing traversals.

Unlike the previous implementations of the OO7 benchmark used for TM-related studies, ours has been written in C++ and uses OpenMP pragmas for thread creation and coordination. Similarly to these previous implementations, while the structure of the design database used by the benchmark conforms to the standard OO7 database specification, the database traversals differ from the original OO7 traversals. In our implementation we allow placement of synchronization constructs (either atomic blocks or mutual exclusion locks provided by OpenMP) at various levels of the database hierarchy, and also allow multiple composite parts to be visited during a single traversal rather than just one as in the original specification. We introduce nested parallelism by allowing the work of visiting multiple composite parts to be split among multiple *worker threads*. Naturally, in order to remain compliant with the original workload executed during benchmark traversals performing database updates, the worker threads have to be synchronized using appropriate synchronization constructs (atomic blocks or locks, depending on a specific configuration).

6.2 Setup

We performed our experiments on a 4 x six-core (24 CPUs total) Intel Xeon 7400 (Dunnington) machine and running Redhat Enterprise Edition 4 at 2.66GHz.

In our experiments with the OO7 benchmark, we compare configurations where mutual exclusion locks (provided by OpenMP implementation) are used for synchronization with configurations where transactional atomic blocks are used for synchronization. We use the standard medium-size OO7 design database. Each client thread performs 1000 traversals and visits 240 (so that it is divisible



(a) pessimistic – coarse-grain (b) pessimistic – medium-grain

(c) optimistic – coarse-grain (d) optimistic – medium-grain

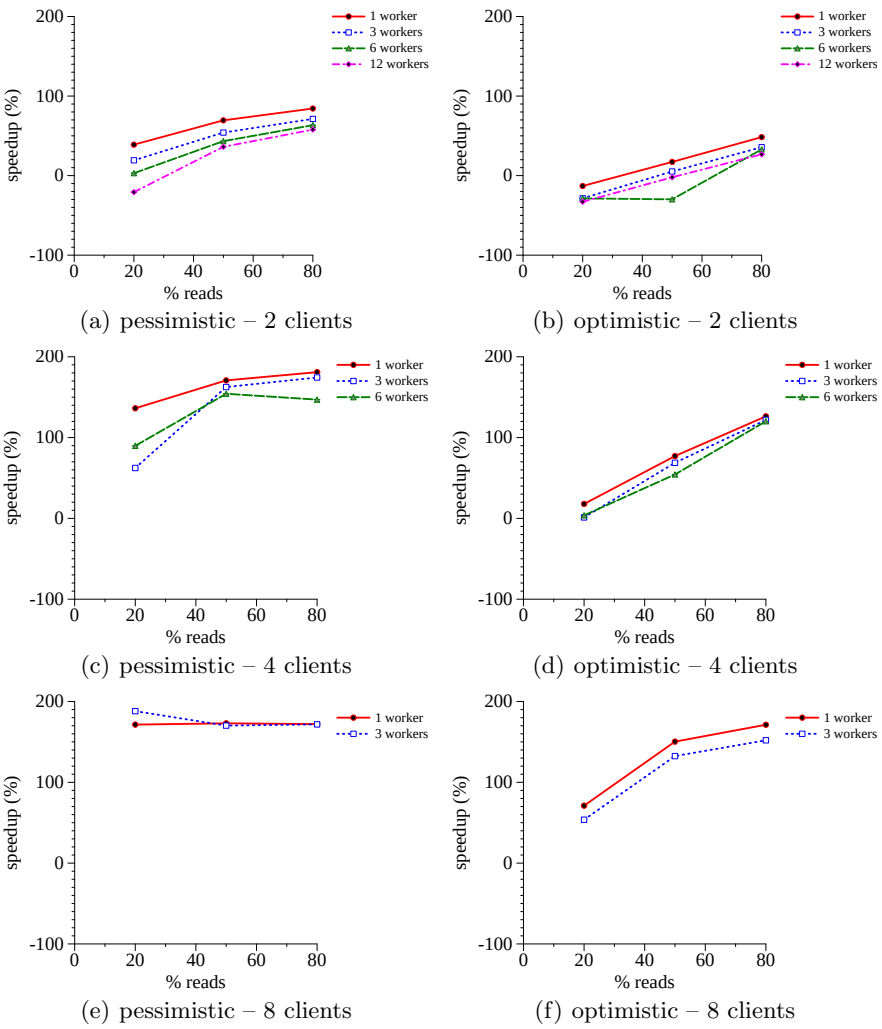
Fig. 6. Performance of transactions vs. locks – no nested parallelism

by 1, 3, 6, 12 and 24 threads) random composite parts at the base assembly level. Since worker threads are created at the base assembly level, which represents level 8 of the database hierarchy (where level is the module), synchronization constructs must be placed at the same level to guard accesses to parts located at the lower levels of the hierarchy.

We vary the following parameters of the OO7 benchmark to achieve good coverage of possible nested parallel workloads:

- number of clients: 2, 4 and 8
- number of workers: between 1 and 24 (to complement number of clients up to 24 total parallel threads)
- percentage of reads vs. writes: 80-20, 50-50 and 20-80
- synchronization level: 1 and 4 for the clients (to simulate coarse-grain and medium-grain synchronization strategy) and 8 for the workers (to guarantee correctness)

We have also experimentally established the value of the concatenation threshold to be 224 log entries, based on the results obtained from a simple single-threaded microbenchmark executing a sequence of memory accesses triggering a series of log operations.



(a) pessimistic – 2 clients (b) optimistic – 2 clients

(c) pessimistic – 4 clients (d) optimistic – 4 clients

(e) pessimistic – 8 clients (f) optimistic – 8 clients

Fig. 7. Performance of transactions vs. locks – coarse-grain + nested parallelism

6.3 Evaluation

When attempting to answer questions we have raised at the beginning of this section, we report performance numbers for both optimistic and pessimistic concurrency protocol as they exhibit different characteristics for a given workload, which may or may not change upon introduction of nested parallelism.

Can transactions retain their performance advantage over locks in presence of nested parallelism? Our hypothesis is that introduction of nested parallelism into a TM system should not change the relative performance

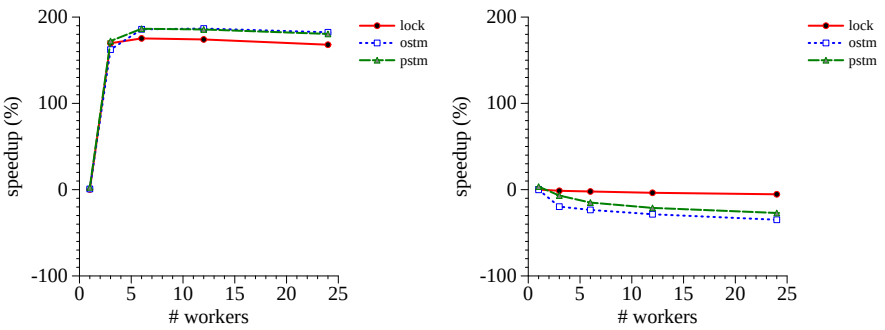


Fig. 8. Performance of a multi-threaded transaction vs. a single-threaded transaction

characteristics between transactions and locks, regardless of the type of concurrency protocol (optimistic or pessimistic) being used.

Figure 6 depicts the relative performance of transactions over locks for pessimistic and optimistic concurrency. In this and the rest of the performance charts, unless noted otherwise, Y axis is speedup of transactions over locks (100% speedup indicates that a transactional configuration was 2x faster) and X axis represents the percentage of reads executed during OO7's database traversals. As we can see, this particular benchmark favors pessimistic protocols. Even though both optimistic and pessimistic transactions perform better than coarse-grain locks, only pessimistic transactions are competitive with medium-grain locks – optimistic transactions can perform up to 50% worse. It is important to note that this is a characteristic of a specific workload executed by OO7 benchmark. Several studies [12, 19, 20] report that optimistic protocols may in fact perform better than the pessimistic ones. It is therefore important to support both types of protocols in a TM system.

In Figure 7 we plot results for configurations utilizing nested parallelism, varying the number of worker threads and comparing the performance of transactions over coarse-grain locks. The number of worker threads gets lower as we increase the number of clients to sum up to 24 threads, which is equal to the number of CPUs available on the machine we use for running all the experiments.

By comparing Figure 7 to Figure 6 we observe that transactional memory with support for nested parallelism preserves the performance benefits that transactions provide over locks. While the parallel nested transactions do not maintain the exact same relative performance difference with respect to a lock-based solution, significant performance improvements can still be expected (up to approximately 200% speedup) especially in cases when contention between client threads is significant, as is the case even with just 4 or 8 client threads. The same performance trend holds for configurations using medium-grain synchronization style (we do not report numbers of medium-grain configurations due to space constraints).

How does performance of parallel-nested atomic blocks compare with performance of the atomic blocks executing the same workloads sequentially? Because of our decision to use locks for synchronization at deeper nesting levels, our expectation is that introduction of transactional nested parallelism should provide the largest performance advantage over sequential execution of code inside transactions when nested parallel threads do not need to be synchronized, that is in the case of shallow nesting. However, the only OO7 workload that can be safely executed without having worker threads synchronized within the same transaction is the read only workload. Nevertheless, we decided to present numbers for some selected configurations of this somewhat trivial workload as they serve as an indication of the performance improvement achievable by applications exercising shallow nesting. In Figure 8(a) we plot results for a single client executing a read-only workload, when varying the number of worker threads between 1 and 24. The worker threads execute unsynchronized but the client thread executes synchronization operation (at level 1), even though it is not necessary for correctness, to account for the cost incurred by transactional execution of the workload (i.e. transaction begin and commit and the cost of read barriers). We report numbers for both optimistic and pessimistic transactional modes, as well as for a configuration that uses an OpenMP's mutual exclusion lock to implement client's synchronization operation and that does not include any transactional overheads. Every data point in Figure 8(a) is normalized with respect to the equivalent configuration (optimistic, pessimistic or lock-based) that does not use nested parallelism and executes the entire workload sequentially. As we can observe, in case of shallow nesting, nested parallelism helps to improve performance of the workload over its sequential execution. However, while it improves performance quite dramatically when increasing the number of worker threads from 1 to 6, it remains constant or even degrades slightly when further increasing the number of worker threads. We attribute this effect to the cost of worker thread maintenance incurred by the OpenMP library that starts playing a more important role as the amount of work executed by a single worker thread gets smaller. This observation is indirectly confirmed by the fact that the configurations using OpenMP locks exhibit similar characteristics.

The remaining question is then how sequential execution of transactional code compares performance-wise with configurations exercising deep nesting. In Figure 8(b) we plot results for a single client executing a workload where the percentage of writes is equal to the percentage of reads [5] and where the worker threads require synchronization in addition to synchronization operations executed by the client. The numbers are normalized similarly to the numbers presented in Figure 8(a). As we can observe, in case of OO7 benchmark the performance of parallel-nested transactional configurations is actually worse than that of configurations executing transactional code sequentially. This result is not surprising, considering that the majority of useful work in the OO7 benchmark is performed by the worker threads. As a result, if the execution of these threads is serialized then, especially after adding inherent overhead incurred by the STM

[5] Configurations with other read-write percentages exhibit similar characteristics.

implementation, no performance improvement over the sequential configurations should be expected. However, please note that lock-based parallel-nested configurations provide no performance benefit over their sequential counterpart either.

To summarize, our performance evaluation indicates that with nested parallelism inside transactions enabled, performance of transaction-based concurrency control mechanisms can still be better than of those based on mutual exclusion. However the serialization imposed by the lock used to implement parallel-nested atomic blocks might detrimentally affect performance of transactions exercising deep nesting.

7 Conclusions

In this paper we have presented the design and implementation of the first STM system supporting nested parallelism inside of transactions, along with a programming model where OpenMP's constructs enabling nested parallelism can be nested inside of TM constructs used for concurrency control (atomic blocks). We expect our system to benefit more applications that use nested parallelism inside transactions with no or low synchronization between nested threads.

References

1. Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay S. Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.
2. Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *PPoPP 2007*.
3. G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS*, 1967.
4. Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM transactional application programming interface. In *PACT 2007*.
5. OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 2.5*.
6. OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 3.0*.
7. Hans J. Boehm and Sarita Adve. Foundations of the C++ concurrency memory model. In *PLDI 2008*.
8. Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *OOPSLA 1994*.
9. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC 2006*.
10. Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA 2004*.
11. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA 2003*.
12. Timothy Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI 2006*.

13. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993*.
14. Jim Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.
15. Vijay S. Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA 2008*.
16. Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S. Unsal, Adrian Cristal, Eduard Ayguadé, and Mateo Valero. Multithreaded software transactional memory and OpenMP. In *MEDEA 2007*.
17. Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL 2008*.
18. Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *HPCA 2006*.
19. Yang Ni, Adam Welc Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA 2008*.
20. Bratin Saha, Ali-Reza Adl-Tabatabai, Rick Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006*.
21. Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Brief announcement: Privatization techniques for software transactional memory. In *PODC 2007*.
22. The OpenMP API specification for parallel programming. OpenMP application programming interface.
23. Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*.
24. Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), September 2005.
25. Haris Volos, Neelam Goyal, and Michael Swift. Pathological interaction of locks with transactional memory. In *TRANSACT 2008*.
26. Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO 2007*.
27. Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *ECOOP 2006*.
28. Jeannette M. Wing, Manuel Fähndrich, J. Gregory Morrisett, and Scott M. Nettles. Extensions to standard ml to support transactions. Technical report, Carnegie Mellon University, 1992.
29. Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay S. Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for Java. In *ECOOP 2008*.