Leveraging Parallel Nesting in Transactional Memory *

João Barreto
INESC-ID/Technical University Lisbon, Portugal
joao.barreto@inesc-id.pt

Aleksandar Dragojević
Swiss Federal Institute of Technology, Lausanne, Switzerland
aleksandar.dragojevic@epfl.ch

Paulo Ferreira
INESC-ID/Technical University Lisbon, Portugal
paulo.ferreira@inesc-id.pt

Rachid Guerraoui
Swiss Federal Institute of Technology, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Michał Kapałka
Swiss Federal Institute of Technology, Lausanne, Switzerland
michal.kapalka@epfl.ch

Abstract

Exploiting the emerging reality of affordable multi-core architectures goes through providing programmers with simple abstractions that would enable them to easily turn their sequential programs into concurrent ones that expose as much parallelism as possible. While transactional memory promises to make concurrent programming easy to a wide programmer community, current implementations either disallow nested transactions to run in parallel or do not scale to arbitrary parallel nesting depths. This is an important obstacle to the central goal of transactional memory, as programmers can only start parallel threads in restricted parts of their code.

This paper addresses the intrinsic difficulty behind the support for parallel nesting in transactional memory, and proposes a novel solution that, to the best of our knowledge, is the first practical solution to meet the lowest theoretical upper bound known for the problem.

Using a synthetic workload configured to test parallel transactions on a multi-core machine, a practical implementation of our algorithm yields substantial speed-ups (up to 22x with 33 threads) relatively to serial nesting, and shows that the time to start and commit transactions, as well as to detect conflicts, is independent of nesting depth.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming - Parallel programming

General Terms Algorithms.

Keywords Nested parallel programs, fork-join, work-stealing, transactional memory.

* This work is funded by the Velox FP7 European project, by the Swiss National Science Foundation grant 200021-116745/1 and by the Portuguese National Science Fundation (FCT) Mercury project (PTDC/EIA/66589/2006).

PPoPP'10, January 9–14, 2010, Bangalore, India.


```
1 atomic { /* transaction t0 */
2      /* transfers a given amount from account A to B */
3      parallel {
4          atomic { /* transaction t1, child of t0 */
5              n = read(A.balance);
6              write(A.balance, n-amount);
7          }
8       ||
9          atomic { /* transaction t2, child of t0 */
10             n = read(B.balance);
11             write(B.balance, n+amount);
12         }
13     }
14     print("New balance of B is " + read(B.balance));
15 }
```

Figure 1. Example of parallel nested transactional program.

1. Introduction

Multicore architectures are on their way to becoming the norm for computing devices in a near future. Yet, the power of multicore requires concurrent programs. However, such programs are significantly more difficult to code than sequential ones. By encapsulating the difficult issue of concurrency control, transactional memory (TM) [9] is a prominent abstraction for simplifying this task.

Using TM, a programmer just needs to (i) create threads, and (ii) delimit which regions of her program must run atomically. A concurrent program is then as simple as depicted in Figure 1. As program execution enters each atomic region, a new transaction begins; nested transactions are created when an atomic region is entered inside an outer atomic region.

Conceptually, the execution of a concurrent program yields a dynamic tree of active transactions, inter-connected by child-parent relations, as Figure 1 illustrates. At any moment, some of the transactions will be running in some processors, while others will be waiting (for instance, for some processor to become available, or waiting for their children to commit). The TM runtime must ensure that, whenever a given transaction wishes to read or write to some shared memory location, it does not violate correctness.

Given two distinct active transactions, when both access the same object, if at least one transaction tries to write the object, and neither one is an ancestor of the other [1], TM detects a conflict. As an example, consider an execution of the previous example where both accounts happen to be the same ($A = B$). Assume that t_1 accesses $A.balance$ (line 5) and, before t_1 commits, t_2 tries to read/write to the same object (lines 10,11). The TM will verify that t_1, which is still active, has already accessed the object and is

not an ancestor of t_2; hence, conflict exists and either t_1 or t_2 will rollback and abort. If otherwise, t_1 had already committed, then the object would have been inherited by t_0's write-set. In this case, the TM will conclude that, although a currently active transaction (t_0) holds the object, t_0 is an ancestor of t_2, thus no conflict exists.

Answering the above ancestor query in an efficient manner is crucial for the TM's overall performance [1]. Most TM systems that support nesting simplify this ancestor test by disallowing child transactions to execute in parallel [4, 8, 12]; i.e. they exclusively support *serial nesting*. In this case, if some parent transaction creates child transactions, then the children will run in the same thread that runs the parent transaction, one after another. Enforcing serial nesting means that the ancestor test is reduced to comparing thread identifiers: only if the thread identifier of the transaction requesting access is the same as the thread identifier of some active transaction that has accessed the desired object, then necessarily the latter is an ancestor of the former.

While serial nesting achieves acceptable performance levels [5, 13], it 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). Recalling the program in Figure 1, the debit and credit transactions (t_1 and t_2, respectively) would have to run sequentially, as they run inside transaction t_0. More generally, any finer grained parallelism that may exist inside the transaction tree is simply neglected. The full power of novel and ambitious paradigms that exploit fine-grained intra-transaction parallelism, such as Free Objects [7], Automatic Mutual Exclusion [10] and dynamic languages as XCilk [1], is greatly hindered by the current TM state-of-the-art.

The main technical challenge of supporting nested parallelism in TM is the intrinsic difficulty of efficiently answering the ancestor test in such a context. This is especially important for composability: deep nesting can create long chains of ancestors.

As some authors emphasize [1, 16], effective support for parallel nesting must gracefully scale to such arbitrary depths. Otherwise, the programmer will be discouraged to expose parallelism that may exist in parts of her programs at reasonable nesting depths. Experience with real transactional programs suggests that nesting is frequent and, often, relatively deep (e.g. [17]). Furthermore, even programs with relatively shallow transactional trees can be called from inside other programs, and hence run at deeper nesting levels.

Agrawal et al. have proposed an algorithm that supports parallel nested transactions such that no-conflict executions of a program[1] with work T_1 and a critical-path length T_∞ (which clearly grows as nesting depth grows) complete in $O(T_1/P + PT_\infty)$ time, where P is the number of threads available [1]. Despite being the best known upper bound up to date, it does not lead directly to a practical implementation. Agrawal et al.'s complex algorithm potentially queries a large number of data structures on each object access, expectedly yielding unacceptable performance penalties [1], and so has not been implemented in practice. More recently, new proposed solutions have moved closer towards the goal of practical parallel nesting. However, they either support significantly limited forms of parallel nesting [14, 16]; or fail to achieve the above upper bound because per-transaction and per-access overhead grow linearly with depth [2].

In this paper we propose the first practical solution for parallel nested TM to achieve depth-independent times to create and commit transactions, as well as to detect conflicts, as implicit in Agrawal et al.'s theoretical upper bound. In our solution, depth-independence is verified as long as the number of threads is bounded by CPU word size (e.g. up to 64 threads in 64-bit computers), in contrast to Agrawal et al.'s unbounded result.

[1] Assuming all accesses are writes.

parallel(thread T_i, block $parBlocks[]$)
1: $contBlock$ = copy of T_i.*block*;
2: $contBlock.program$ = the remainder of $T_i.block.program$
3: **for** each block b in $parBlocks$ **do**
4: $b.baseTx = T_i.tx$;
5: $b.minEp = T_i.ep$;
6: $b.succBlock = contBlock$;
7: $contBlock.precBlocks++$;
8: $enterMonitor(queue)$;
9: **for** each block b in $parBlocks$ **do**
10: $queue.enqueue(b)$;
11: $leaveMonitor(queue)$;

stealBlock(thread T_i)
1: $enterMonitor(queue)$;
2: $b = freeBitnumQueue.reserveFreeBit()$;
3: **while** $isEmpty(queue)$ **do**
4: $wait(queue)$
5: $T_i.block = queue.dequeue()$;
6: **if** $T_i.block.succBlock$ is not null **then**
7: $lock(T_i.block.succBlock)$;
8: $T_i.block.succBlock.precBitnums += b$;
9: **if** $T_i.block.succBlock.precBlocks == 1$ **then**
10: $discardBitnum(T_i.block.T_i.block, T_i.ep)$;
11: $unlock(T_i.block.succBlock)$;
12: $leaveMonitor(queue)$;
13: $T_i.ep = max\{T_i.block.minEp; T_i.block.bn.minEp\}$;
14: $T_i.tx = T_i.block.baseTx$;
15: $T_i.block.bn = b$;

finishBlock(thread T_i)
1: $discardBitnum(T_i.block.bn, T_i.ep)$;
2: **if** $b = T_i.block.succBlock$ is not null **then**
3: $lock(b)$;
4: $comDesc -= (discarding + comMask[b.minEp])$;
5: $comDesc += T_i.block.bn$;
6: $b.precBlocks--$;
7: $b.precBitnums -= T_i.block.bn$;
8: $b.minEp = max\{T_i.ep; b.minEp\}$;
9: **if** $b.precBlocks == 1$ and $b.precBitnums \neq 0$ **then**
10: $discardBitnum(b.precBitnums, T_i.curEp)$;
11: **if** $b.precBlocks == 0$; **then**
12: $T_i.block = b; T_i.ep = b.minEp; T_i.tx = b.baseTx$;
13: run b;
14: $unlock(b)$;

discardBitnum(bitnum, lastEpoch)
1: $T_i.lastComEp[bitnum] = lastEpoch$;
2: $T_i.discardBitnum[bitnum] = TRUE$;

Figure 2. Work-stealing.

As proof of concept, we have completely implemented and evaluated our algorithm as a Java library. The experimental results confirm competitive speed-ups relatively to serial nesting (up to 22x with 33 threads). Furthermore, results support our theoretical claim that the time taken by our algorithm to start and commit transactions, as well as detecting conflicts on each object access is independent of nesting depth, for the evaluated number of threads.

Overview

At the heart of our algorithm, we rely on a very lightweight, constant-time ancestor query algorithm that only supports a bounded number of transactions. However, as we show later in the paper, such a bounded query is sufficient to support any execution with an unbounded number of transactions, at an unbounded nesting depth.

The key data structures are *bit vectors* of fixed size, N, which we use in diverse parts of our algorithm to identify the ancestor sets of our transactions. The fixed dimension, N, is given by $2P$, where

beginTx(thread T_i)

1: $T_i.tx.bitnum = T_i.block.bn$;
2: $T_i.tx.beginEp = T_i.ep$;
3: $T_i.tx.anc+ = T_i.tx.bitnum$

commitTx(thread T_i)

1: $T_i.lastComEp[T_i.tx.bitnum] = T_i.ep$;
2: $T_i.ep + +$;
3: $T_i.tx = T_i.block.baseTx$;

write(object x, value v, thread T_i)

1: $lock(x)$;
2: **if** $x.stack.isEmpty()$ **then**
3: $x.stack.push(T_i.tx.anc, x.value)$;
4: $x.value = v$;
5: **else if** $x.stack.top().anc == T_i.tx.anc$ and $T_i.tx.beginEp \leq x.stack.top().ep \leq T_i.ep$ **then**
6: $x.value = v$;
7: **else if** $noConflict(x.stack.top(), T_i)$ **then**
8: $x.stack.push(T_i.anc, x.value)$;
9: $x.value = v$;
10: **else**
11: Handle conflict.
12: $unlock(x)$;

noConflict(stackEntry e, thread T_i)

1: $xanc = activeAncestors(e, T_i)$;
2: **return** $(\overline{xanc} \wedge ((xanc \oplus T_i.anc)) == 0)$;

Figure 3. Basic transactional support.

P is the number of threads on which the underlying STM runs a given program. We identify transactions by a *bitnum*, a unique index (ranging from 0 to $N-1$) of all bit vectors that our system maintains. Hence, given any bit vector, the set of transactions it represents is given by the transactions whose bitnum is set to 1.

In order to check whether some transaction i, whose ancestor set is anc_i, is an ancestor of another transaction j, with ancestor set anc_j, it suffices to determine if anc_i is a subset of anc_j. With our bit vector representation of anc_i and anc_j, we can answer such a query with a couple of bitwise operations, by checking whether $(\overline{anc_i} \wedge (anc_i \oplus anc_j)) == 0$.

Our TM maintains ancestor bit vectors along each object, in order to denote the ancestors of the current active transactions that have accessed that object. We further enrich our solution with techniques such as bitnum re-use, lazy bit reclaim, immediate commitment propagation and single-child transaction optimization, which we describe next.

The core of our algorithm is presented in detail in Figures 2 and 3. In the remainder of the paper we revisit these figures, describing and discussing each line in detail.

The following sections are organized as follows. Section 2 starts by introducing the notations and assumptions of the paper. Section 3 describes the work stealing system, the framework underlying our TM. Section 4 then introduces the basic algorithms for creating and committing transactions, and detecting conflicts. We then address advanced aspects of our algorithm: Section 5 describes lazy bitnum reclaiming, while Section 6 explains how our bounded bit vector structure can, in fact, support unbounded transaction trees. Section 7 evaluates the algorithm. Section 8 surveys related work. Finally, we draw conclusions in Section 9.

2. Basic Notation and Assumptions

Hereafter, when considering two bit vectors, x and y, we use $x+y$ and $x-y$ to denote $x \vee y$ and $x \wedge \overline{y}$, respectively, where $\overline{y}$ is the bit inversion of y. When no ambiguity exists, when referring to a given bitnum, b, we either mean the integer representing the position of b in any bit vector (from 0 to $N-1$), or the bit vector where the only 1-bit is the one corresponding to b. Moreover, we write $x+b$ or $x-b$ (where x is a bit vector and b a bitnum) to denote the bit vector resulting from setting/clearing (respectively) b's bit in x.

We assume that individual memory writes are atomic when no write contention to the same variable exists (i.e. any concurrent read to the same single variable will always read a consistent value). This is supported by most modern multi-core machines and runtime platforms (e.g. [6]).

3. Epoch-based Work Stealing

We depart from an XCilk-like [1] programming language and work stealing system [3]. For simplicity of presentation, we consider an elementary work stealing system, based on a global queue; this is without loss of generality, as it is straightforward to extend our solution to more efficient multi-queue work stealing systems [3].

Since the work stealing system forms the framework on which our TM will operate, we start by describing the work stealing system next, enriching it with additional features required by our TM. We assume a multi-core computer running P worker threads, $T_0, ..T_P$.[2] A central assumption in the remainder of the text is that P is bounded by word size, so that comparing bit vectors of size $N = 2P$ becomes possible in one or two hardware instructions.

When no ambiguity exists, we simply refer to worker threads as threads. Each thread, T_i, can either be idle, or running some *block* that it has stolen from the global queue [3]. We assume that, at each moment, T_i is running in the context of some transaction.

As later sections explain, our TM needs to reason about the happens-before relation [11] between events concerning different transactions. Namely, such events include the beginning and commitment of any transaction, as well as each memory access any transaction makes. Threads support that by maintaining logical clocks [11], which we call *epochs*.

At each moment, each running thread is executing a given block, and is at a particular epoch. During execution of a block, a thread can evolve to greater epochs than the one in which it started running the block, in situations which we shall explain in the next section. The current epoch of a thread increases monotonically only, and it can differ from other threads' current epochs.

Summing up, each thread, T_i, maintains the following local state:

- T_i.*ep*, the current epoch number;
- T_i.*tx*, the current transaction identifier (addressed in the next section);
- T_i.*block*, the block being executed;

Additionally, T_i maintains the following attributes, whose meaning we clarify shortly:

- T_i.*lastComEp*
- T_i.*discardBitnum*

Blocks encapsulate program fragments, which threads can run. A block can be in one of three states: (i) *waiting*, meaning that the block cannot run yet, as it is dependent on one or more blocks to complete; (ii) *enqueued*, meaning that any thread that becomes idle can steal and run it; (iii) *running*, at some thread T_i. Every block starts in the waiting state, and eventually it moves to the enqueued and running states, in this order.
In either state, a block has the following attributes:

- $b.program$, the block's program;
- $b.baseTx$, the transaction in which the block is when its execution starts;

[2] Ideally, P is less than the number of available cores, but there can also exist more (swapping) threads than cores.

- $b.bn$, the bitnum reserved for this block, which is unassigned in the waiting and enqueued states;
- $b.minEp$, the minimum epoch at which the thread that steals the block has to be to start running it;
- $b.precBlocks$, the number of blocks that need to complete before this block can be enqueued; when enqueued or running, $b.precBlocks$ is 0.
- $b.precBitnums$, a bit vector denoting the set of reserved bits of every block which b is waiting for; when enqueued or running, $b.precBitnums$ is empty.
- $b.blockWaitingForMe$, the block (if any) that is waiting for this block (and possibly others) to complete.

The next sections explain the need of each attribute above.

3.1 Forking into parallel blocks

Initially, only one block exists, called the *root block* which contains the main program. As a running block (e.g. the root block) finds a `parallel` statement, it ceases its execution and decomposes itself into multiple smaller blocks. The resulting blocks comprise: the inner parallel blocks, which are enqueued, and a continuation block, which contains the remainder of the program after the `parallel` statement, and whose state is set to waiting, until the inner blocks finish. It is straightforward to extend the work stealing system to support other models. For instance, when the root block continues to execute, instead of waiting for the spawned blocks to finish.

Hereafter, we say that the inner parallel blocks in a `parallel` statement are *sibling blocks* (as any transactions they create will be sibling transactions). Moreover, we say that the inner parallel blocks are the *preceding blocks* of the continuation block, which in turn is the *succeeding block* of the inner parallel blocks.

Procedure $parallel$ from Figure 2 details how we initialize the blocks resulting from a `parallel` statement found by some thread, T_i.

Each block resulting from the `parallel` statement is created in the context of T_i's current transaction when T_i reached the statement (lines 2 and 4). We designate such a transaction the blocks' *base transaction*.

Other threads may later steal the blocks resulting from the `parallel` statement. Therefore, such threads' epochs need to reflect the fact that the events that the original block has seen happen-before the new blocks. We ensure that by associating a minimum epoch with each new block, which we set to the last epoch at which the original block ran (lines 2 and 5).

3.2 Running and stealing blocks

During execution, a block may initiate new child transactions of its base transaction, as the block's execution enters new `atomic` regions. We impose that, in the program of a single block, at most one nesting level of `atomic` regions can exist.[3]

As the next section describes, for a thread to initiate a transaction, it needs to hold a reserved bitnum, which will serve to identify that transaction. We obtain such a bitnum from a global *free bitnum queue*. Each entry in the free bitnum queue contains a bitnum and the minimum epoch at which the thread wishing to use the free bitnum must be. The minimum epoch attribute is necessary because bitnums can be re-used for different transactions at different epochs. As we describe shortly, we set the minimum epoch of each free bitnum to a value that is greater than the epochs at which the previous transactions that have used the bitnum have committed. This way, we ensure that epochs correctly reflect the happens-before relation between transactions that have shared a bitnum.

Since different threads can try to concurrently reserve a bitnum, we ensure mutual exclusion when accessing the free bitnum queue by locking. Acquiring a lock each time a new transaction started would be prohibitively expensive. Instead, we minimize synchronization overhead by reserving, on steal-time of a given block, one bitnum. We then use (and re-use) that bitnum for any transaction that that block may initiate.

We can finally define all the steps a thread, T_i, needs to take before being able to run a block. Such steps occur when T_i steals the block, as Procedure $StealBlock$ in Figure 2 presents.

Essentially, T_i starts by reserving a free bit for the block it will steal (line 2). Then, T_i dequeues one block from the queue (line 5) and saves the bit reserved for the stolen block in the $precBitnums$ attribute of its succeeding block (if any) (line 8). The need for line 8 is not evident now; we return to it in Section 6. The previous two steps require mutual exclusion, which we can safely achieve with only one lock associated with the queue. Thread T_i then advances its current epoch in order to satisfy the minimum epochs of the block and the reserved bit (line 12), and sets T_i's current transaction as the block's base transaction. Finally, T_i can start running the block's program.

3.3 Finishing a block

Finally, when a block's program finishes, the thread running it proceeds as in Procedure $finishBlock$ of Figure 2.

Essentially, $finishBlock$ has two goals. Firstly, a call to *discardBit* notifies the system about the fact that thread T_i has finished using bitnum T_i.*block*.bn, at epoch T_i.*ep*. We publish such information in the T_i.*lastComEp* and T_i.*discardBitnum* vectors of T_i's state, in the position corresponding to the bitnum to discard. No locking is required since, while holding the reserved bitnum, only this thread can write to these vectors.

After setting such values, the thread immediately returns. Eventually, after the call to *discardBit*, the system will asynchronously place the discarded bitnum in the free bitnums queue again, with a minimum epoch that is greater than the epoch at which T_i discarded it. For the moment, we are not concerned about how such an asynchronous task takes place. Section 5 addresses that.

Secondly, we update the finishing block's succeeding block, b (if any). This means removing any reference to the finishing block in b's attributes (lines 6 and 7). Furthermore, imposing that, once b starts running, it will not be at an earlier epoch than the epoch at which b's preceding blocks have finished (line 8). Otherwise, we would be inconsistent with the happens-before relation between preceding blocks and their successor. If the finishing block is the last one b is waiting for, this means that b is finally ready and T_i can immediately run it (line 13). After running $finishBlock$, T_i become idle again, thus it then tries to steal a new block.

4. Basic Transactional Memory Algorithm

In the previous section, we have laid the ground on which our TM will work. This section finally introduces the base algorithms that support transactions in our programs. We start by explaining how we identify transactions and represent their position within a tree of other transactions. Then we proceed to describe how we ensure the safety of memory accesses.

4.1 Transactions

We identify each transaction, t, by a bitnum. However, the limited size of the bitnum space (N) requires us to re-use bitnums to identify new transactions, as soon as the previous transaction that has used the bitnum has committed. Therefore, t's identifier is only valid when we consider an epoch during which t was active; out-

[3] Note that this internal restriction is not visible to the programmer. The programmer can still transparently code programs with multiple nesting levels inside a block, such as `atomic{atomic{...}}`. A pre-compiler can then seamlessly translates such a program to an equivalent one with single-level blocks: `atomic{parallel{atomic{...}}}`.

side that period, that bitnum can be identifying another transaction. Consequently, we can only univocally identify t by the pair comprising its bitnum and some epoch at which t was active.

Besides identifying, we need to be able to determine position of t within the transactional tree. For that, we maintain t's ancestor set as a bit vector whose bitnums corresponding to t's ancestors (t included) are set to 1. Once again, t's ancestor set is only valid with respect to an epoch at which t was active (thus, all its ancestors too).

So far, we know that each block, b, has a base transaction. The corresponding attribute, $b.baseTx$, has three fields: $bitnum$, $beginEp$ and anc, which respectively denote the transaction's bitnum, first active epoch and ancestor set.

Another transaction relevant to block b is its current transaction when b is running at some thread, T_i. T_i maintains, at $T_i.tx$, the same fields as above, $bitnum$ and anc, concerning its current transaction. Upon stealing b, $T_i.tx$ is $b.baseTx$. However, when execution enters an `atomic` region, a new child of the base transaction starts. Starting such a new transaction is a very lightweight operation, which essentially involves devising the bitnum, $beginEp$ and anc attributes for the new transaction, as Procedure $beginTx$ in Figure 3 shows. As we know, the bitnum of the new transaction is the one that has been reserved a priori for the block. The initial epoch is the current one of T_i. Finally, we easily update anc by setting the new transaction's bitnum to 1, since all the other ancestors were already present in anc.

Eventually, the child transaction will commit.

Committing the current transaction is, again, a very simple operation, as Procedure $commitTx$ in Figure 3 shows. We start by updating $lastEpoch$ to the last epoch where the thread used its reserved bitnum with some transaction (line 1). Although line 1 may seem intuitive we have not motivated the need for it; we return to this line in next section.

Before returning to the base transaction, we advance T_i's current epoch (line 2). This ensures that, should T_i initiate subsequent transactions in the same block, they use the same bitnum (i.e. the bitnum reserved for that block), it will be at a distinct epoch than the epochs at which the committing transaction was active.

4.2 Basic Conflict Detection

We maintain a stack attached to each object, where we push the ancestor set of each transaction that accesses the object; i.e. when transaction t accesses an object, we push $t.anc$ and $t.ep$ into the object's stack. As it will become evident next, the topmost entry of each object's stack will always denote a transaction that is a descendant of any other entry in the object's stack.

During a transaction's lifetime, a thread can perform read and write accesses to objects. Of course, conflicts can occur with concurrent transactions, which must be avoided in order to ensure correctness [1]. Our approach to conflict detection is eager-validation.[4] Hence, before accessing some object, a transaction must test for conflicts. For space restrictions, in this paper we consider all accesses as writes. Prior to any access, we run Procedure $write$ from Figure 3.

Each time some thread, T_i, wishes to access a given object with a non-empty stack, we simply peek the topmost ancestor vector in the object's stack and check whether a conflict exists.

Firstly, we look for the easy cases: where the stack is empty (line 2) and where the transaction trying to access the object was the latest one to do it (line 5). Otherwise, we need to answer the hard question of whether the transaction, t_x, on top of the object's stack, is an ancestor of the transaction requesting access to the object. Another way of making the same question is whether the set of *active* ancestors of t_x is a *subset* of the ancestor set of transaction the requesting access to the object. By taking advantage of our representation of ancestor sets as bit vectors, we answer such a query with the couple of bitwise operations in line 2 of Procedure $noConflict$ in Figure 3.

If the above answer is yes, then the transaction trying to access the object is necessarily a descendant of the all the active transactions that have accessed the object, and thus can access it too. Otherwise, a conflict exists and at least one of the contending transactions must abort.

The main challenge in the conflict test above is in determining which is the set of currently active ancestors of t_x (function *activeAncestors* in Procedure $noConflict$ in Figure 3). One naive solution is to, when some transaction, t, is about to commit, go through the stack of every object in that t's write-set (i.e. the objects accessed by t and by every t's descendant), and clear t's bitnum in the corresponding stack entries (hence, automatically propagating the object to the write-set of t's parent).[5] Let us call such a step *bitnum reclaiming*.

Obviously, on-commit bitnum reclaiming would incur a considerable overhead. Namely, it requires extensive locking, the work performed during bitnum reclaiming of some transaction is repeated at each of its ancestors (i.e. it is multiplied by the nesting depth of the transaction), and it implies maintaining explicit write-sets along with each transaction. In the next section we describe our alternative, which eliminates all the above shortcomings, keeping $commitTx$ as simple as defined in Section 4.

5. Lazy Bitnum Reclaiming

Our algorithm employs lazy bitnum reclaiming, instead of on-commit bitnum reclaiming. Intuitively, we accomplish that by postponing bitnum reclaiming of committed transactions to a much later period, where we reclaim the bitnums from transactions that have committed recently all together. As we show next, the period between batch bitnum reclaiming sessions is proportional to the maximum number of epochs, E, and can be substantially long. Thus, the set of transactions whose bitnum reclaiming is batched can be considerably large. Consequently, for such a large set of transactions, we avoid repetitive (thus redundant) bitnum reclaiming of common stack entries (the pathological effect discussed in the previous section).

The key challenge is, thus, to ensure that accesses to objects that have been accessed by already committed, but not yet bitnum-reclaimed transactions are correctly handled. *Committed masks* (as well as the notion of epoch) help us solve such a problem, as we explain next. We maintain a global array of bit vectors, each called a committed mask, one per epoch ($0..E$). Intuitively, each committed mask centralizes information about which transactions that were active at the corresponding epoch have already committed.

We denote the committed mask of epoch e as *comMask*$[e]$. Naturally, each committed mask starts with all bits set to 0. Eventually, as each transaction t commits, the corresponding bitnum in the committed mask of every epoch during which the transaction was active will be set to 1. By then, we say that t is *published*.

We address the publication process shortly. For now, let us simply assume that a transaction is instantaneously published once it commits.

activeAncestors(stackEntry e**, thread** T_i**)**
1: **return** $e.anc - comMask[e.epoch]$

Above, we show how we resort to the information centralized in committed masks to implement the $activeAncestors$ function of our conflict test algorithm (recall $noConflict$ from Figure 3).

[4] Adapting a late-validation solution to the case of parallel nesting seems straightforward. However, as discussed in [1], late-validation necessarily implies doing work that is proportional to transaction depth at commit.

[5] In this case, function *activeAncestors* would do nothing.

```
1: loop
2:     for each worker thread T_i do
3:         for each bitnum 0 ≤ bn ≤ N − 1 do
4:             if T_i.lastComEp[bn] > lastEpochInMask[bn] then
5:                 for e = lastEpochInMask[bn] + 1 to T_i.lastComEp[bn]
                   do
6:                     comMask[e] += bn;
7:                     lastEpochInMask[bn] ++;
8:             if T_i.discardBitnum[bn] = TRUE then
9:                 discarding[bn] = TRUE;
10:                maxCurEp = max_{0≤k<P}(T_k.ep);
11:                for e = lastEpochInMask[bn] + 1 to maxCurEp do
12:                    comMask[e] += bn;
13:                lastEpochInMask[bn] = maxCurEp;
14:                discarding[bn] = FALSE;
15:                T_i.discardBitnum[bn] = FALSE;
16:                enterMonitor(queue);
17:                freeBitnumQueue.addFreeBit(b, maxCurEp + 1);
18:                leaveMonitor(queue);
```

Figure 4. Publisher thread.

Essentially, we filter out the bitnums of the transactions that, being active at the epoch at which the access took place, have committed in the meantime. Those are the ones published in that epoch's committed mask (line 3).

$activeAncestors$ works correctly no matter the nesting depth of the transaction, t, that originally accessed the object. When t commits and is published, the ancestor set stored at the object's stack entry, when consulted by $activeAncestors$, will retain every bitnum except t's; thus, it will become t's parent's ancestor set. Implicitly, this is equivalent to merging t's write-set with t's parent write-set at the moment when we published t. The same thing happens when t's parent commits, and so forth.

5.1 Publishing Committed Transactions

The assumption, made in the previous section, of instantaneous publication is not a realistic one. Publishing a transaction entails updating $O(e)$ committed masks, where e is the number of epochs of the transaction's lifetime. Pushing that task into the $commitTx$ function would increase its time complexity to $O(e)$ (rather than $O(1)$) and would require expensive locking in order to synchronize access to the shared committed masks.

We eliminate both shortcomings by delegating such a task to a single specialized thread, called the *publisher* . The publisher is the only thread that writes to the committed masks, therefore avoiding the need for any locking. When a worker threads runs the simple, constant-time commit procedure described in Procedure $commitTx$ in Figure 3, it can immediately continue its execution, without having to wait until the committed transaction is finally published.

The publisher works in background, guided by the information it collects from the T_i.*lastComEp* vectors that each worker thread maintains. Recall, from $commitTx$, that each time a thread, T_i, commits some transaction, t, it sets the corresponding entry in T_i.*lastComEp* to its current epoch (before advancing to the next epoch). Moreover, when thread T_i completes executing a block, it discards its reserved bitnum by setting the bitnum's entry in T_i.*discardBitnum* to true.

The publisher thread continuously loops in the algorithm in Figure 4. The algorithm periodically reads the T_i.*lastComEp* and $T_i.discard$ vectors of each thread T_i, checking for modifications in either one.

On the one hand, each time the publisher finds a thread whose T_i.*lastComEp*$[b]$, for some bitnum bn, is greater than the epoch at which the publisher last set bn as committed (line 4), it means that, in the meantime, one or more transactions have committed using that bitnum. Hence, the publisher sets bn in the committed masks of every epoch $e \leq T_i$.*lastComEp*$[b]$.

While this necessarily implies that masks of the epochs at which the committed transaction was active will show that transaction as committed, it can also set the transaction as committed in epochs during which the transaction did not exist. However, this does not affect the safety of our conflict detection algorithm, as it is easy to show that, in the latter epochs, no other (non-aborted) transaction was or will be active. Hence, no object stack may have an ancestor set timestamped with one of those epochs and having bitnum bn set.

On the other hand, the above algorithm is also responsible for freeing bitnums that have been discarded because the block that held them finished. In such a case, the publisher adds such a bitnum to the free bitnum queue (lines 15 to 18). The minimum epoch attribute of such a new free bitnum will be greater than the epoch after the last epoch at which a transaction with such a bitnum committed (line 17). Before actually freeing a bitnum, the publisher may need to set some additional epochs as committed for the bitnum being discarded (line 12); only after those epochs can the bitnum be re-used again. In the next sections we turn our attention to lines 9 to 14, explaining the need for such additional committed epochs.

If enough cores are available, one can easily parallelize the publisher into multiple parallel threads. We would simply need to partition the bitnum space and have each thread responsible for publishing the bitnums in one particular partition.

However, no matter how parallel the publisher is, having a background publisher has an important impact on our conflict detection test, as we illustrate now. Consider some transaction, t_a, that has committed, and some active transaction, t_b, which may wish to access objects that t_a previously accessed. We say that t_a*'s commitment has propagated to* t_b once t_b becomes aware of the fact that t_a has committed.

So far, we have seen that, after t_a commits, the committed masks of the epochs at which t_a was active will be temporarily stale, until t_a is finally published in background. Consequently, if t_b calls $activeAncestors$ (see previous section) during such a period, t_b may detect false conflicts with objects that t_a has accessed. False conflicts do not harm safety, as aborting t_b (and retrying some time after) is always safe. Nevertheless, unnecessary aborts degrade performance. We address such a problem in the next section, showing that, for the most relevant situations, we can actually prevent false conflicts, regardless of how long the publication latency is.

5.2 Preventing Pathological False Conflicts

So far, we have seen how commitment of one transaction, t_a, propagates to the remaining active transactions by publishing t_a. Since publication may take a significant time, our algorithm may detect false conflicts if, in the meantime, any active transaction, t_b, requests access to any object that the committed transaction had accessed.

However, the gravity of such false conflicts varies substantially, depending on the relation between t_a and t_b. More precisely, we can distinguish the three following cases:

1. t_a and t_b have the same parent transaction, and t_b runs sequentially after t_b (at the same block);
2. t_b is t_a's parent, which blocked until the block running t_a completed, before resuming execution (in the same block of t_a or not);
3. t_b and t_a are concurrent transactions;

Clearly, the programmer's expectations concerning the possibility of conflicts between t_a and t_b in each case vary radically. Whereas, in case 3, the programmer already expects conflicts to occur oc-

casionally (after all, the program has two concurrent transactions accessing a common object), his expectation in cases 1 and 2 is that conflicts between t_a and t_b will never occur. Consequently, the programmer will try to minimize accesses to objects that are shared with concurrent transactions (case 3).

However, he will not be reluctant to (and will often) have programs such as the one in Figure 1 from Section 1. The program illustrates a very common case of case 2 (one can easily think of a similar program with case 1), where t_a is t_2 and t_b is t_0. Naturally, the programmer expects that line 14, which accesses a shared object (B) that, in turn, a previous transaction (t_a, i.e., t_2) has accessed, is conflict-free. In fact, the programmer knows that, at that point, t_a will necessarily have committed.

Therefore, false conflicts due to the non-immediate commitment propagation from t_a to t_b are particularly problematic in cases 1 and 2, in contrast to case 3. Case 2 is especially delicate since a false conflict when t_b tries to access an object accessed by its committed descendant, t_a, will imply aborting t_b and, consequently, t_a and all other descendants. For these reasons, we call cases 1 and 2 *pathological* false positives.

Fortunately, we can prevent pathological false positives with no or few lightweight modifications to the algorithm described so far. Most importantly, such modifications neither increase the time complexity of our algorithm, nor require additional locking. Intuitively, our approach is to find alternative means to propagate t_a's commitment to t_b in cases 1 and 2, avoiding t_b's need to wait for the general-case propagation through the publisher.

Perhaps surprisingly, the conflict detection test in Procedure $write$ in Figure 3 already prevents false conflicts in case 1. Recalling the algorithm, one can see that, if the contending transactions have the same ancestor set, the algorithm never yields a conflict (independently of whether t_a is already published or not). This is clearly what happens in case 1, where t_a and t_b have the same ancestors, and have the same identifier bitnum (the one reserved for the block containing both). In this case, $write$ knows that t_a (whose ancestor set is in the object's stack) has necessarily committed already. Otherwise, its bitnum could not have been subsequently reused by t_b.

False conflicts are harder to prevent in case 2. The additional difficulty results from the fact that t_a's bitnum now has already been discarded (since the containing block has already completed, in contrast to case 1) and, thus, can already be in use by some concurrent thread. Still, we are able to prevent false conflicts in case 2 with few lightweight modifications to the algorithm constructed so far and, most importantly, with no need for additional locking. For that purpose, we need maintaining one additional attribute, called *comDesc*, along with each block. Intuitively, *comDesc* includes the bitnums of transactions that the block knows to have committed, but might not have been published yet. We start by tackling case 2 where t_b has only one descendant transaction, t_a, before we proceed to the more generic case of multiple descendants with distinct bitnums.

In the single-descendant case, when t_a's block finishes (calling function $finishBlock$, defined in Section 3), we add t_a's bitnum to the *comDesc* attribute of its succeeding block (which will run in the context of t_b).

The *comDesc* attribute can be seen as a note that the thread that has completed the block containing t_a leaves to whichever thread steals its succeeding block, which runs in the context of t_b. Such a note allows the latter thread to immediately learn about the fact that t_a has committed, probably before t_a is published. We ensure this by adding two relatively lightweight and $O(1)$-time changes to the $activeAncestors$ function, as Figure 5 shows.

In line 3, t_b's thread takes advantage of the information in *comDesc* and ignores t_a's bitnum in any ancestor set of stack entries of objects t_b tries to access. Hence, no matter how long

activeAncestors(stackEntry e**, thread** T_i**)**
1: **if** T_i.*comDesc* $\neq 0$ **then**
2: $\quad T_i$.*comDesc* $-=$ *comMask*$[T_i.block.minEp]$;
3: **return** $e.anc -$ *comMask*$[e.epoch] - T_i$.*comDesc*;

Figure 5. New version of *activeAncestors*.

t_a's publication takes, t_b will never run into false conflicts when accessing objects that t_a had accessed.

Evidently, t_b's thread can only do that as long as t_a's bitnum has not yet been re-used for new transactions. If t_b kept ignoring t_a's bitnum in that situation, conflicting accesses (to objects accessed by concurrent transactions that re-use t_a's bitnum) could be incorrectly granted to t_b.

Fortunately, t_a's bitnum is only freed (i.e., new transactions can re-use it) after t_a has been published. Hence, we first check whether t_a has already been published (line 2). If so, we cease ignoring t_a's bitnum (by clearing it from *comDesc*), in line 2. We know whether t_a is already published by reading from the committed mask of t_a's commit epoch (which, in the single-descendant case, is the epoch set in the $minEp$ attribute of the block running t_b).

6. Supporting Unbounded Transaction Trees

The algorithm described so far does not support more than N transactions to be simultaneously active (where N is the bit vector size). This is not a problem with parallel programs with no nesting, as long as $N \geq P$ (the number of worker threads). In fact, a flat program with $m > N$ parallel (flat) transactions will attain as much parallelism as it would if an unbounded number of transactions was supported, since only P transactions can be running at a given moment. Hence, only $N = P$ simultaneously active transactions need to be supported in this case: as soon as some thread commits one transaction, the corresponding bitnum will be freed and then become available for that thread to start the next transaction.

6.1 Limiting Parent Transactions

A problem, however, arises when we consider nesting. A worst-case program can have a transactional tree of depth P (with intermediate transactions $t_0, .., t_P$, where t_i is t_{i+1}'s parent), where each t_i decomposes into two blocks: one starting the next-level transaction and another *work block*, w_i, with some flat program that does some useful work. In this case, we have P active transactions, $t_0, .., t_P$, each consuming one bitnum.

If $N = P$, then no free bitnums would be available at this point. This means that the worker blocks (w_i) that, at each nesting level, had been queued up for other threads to steal, would not be stealable, as no bitnums would remain to run them. In contrast to the flat program example, there are now some transactions that have blocked (i.e., each t_i, awaiting for their descendants to commit), consuming bitnums that otherwise could allow parallel blocks to run. Consequently, $P - 1$ threads are forced to remain idle, even when there are enqueued parallel work blocks.

A solution to this situation is to impose a limit, L, on the number of bitnums that can be used for parent transactions. This way, there will always remain (at least) P bitnums left for leaf blocks (i.e. blocks with no sub-blocks) to run by each thread.

We ensure the limit L by maintaining, along with the free bitnum queue, a counter of how many bitnums have been reserved so far for transactions that, subsequently have become parent transactions. When a block that has started an (initially childless) transaction reaches a `parallel` statement and is about to enqueue the inner parallel blocks it has found, we conservatively increment such a counter (as each such a block can potentially start child transactions of the outer transaction). (If the limit is reached, an error is returned and queuing of the inner blocks is disallowed, a situation to which we return shortly.)

A good choice for L is $L = P - 1$, thus we set N to $2P - 1$. Such a value of L is sufficient if each parent transaction has at least two parallel child blocks (as in the example above): it means that $P - 1$ parent transactions will queue enough leaf blocks (at least P), which the P threads can steal and run. A lower value of L could restrict the number of blocks that would be made available for other thread to run; and, as we show next, a higher value is not necessary.

6.2 Single-Child Transactions

Limiting the number of bitnum reservations for parent transactions is sufficient to attain maximum parallelism in the particular example above. Still, it is not generic enough for arbitrary transaction trees, namely: (*case (i)*) when some parent transactions have only one child block; and/or (*case (ii)*) when nesting depth is higher than P.

The key insight to solving both cases is that single-child transactions do not need a distinguishing bitnum, and can simply borrow their parent's bitnum. Consider some transaction, t_p, that has a single child transaction, t_c. Clearly, every ancestor of t_p is also an ancestor of t_c, and every concurrent transaction (i.e. neither ancestor nor descendant) of t_p is also concurrent with t_c. This means that, being a single child, t_c can borrow its parent identifiers without affecting the safety of our conflict tests.

This observation directly solves case (i). In the case of a `parallel` statement with a single inner block, any transaction in that block will not consume a free bitnum, as it inherits its parent's bitnum.

Case (ii) is more intricate but we can solve it with few simple modifications to our algorithm. Let us recall transaction T_P at depth P from the example in Section 6.1. Assume that the block in which T_P runs has a `parallel{b1,b2,..,bn}` statement. Assume that thread T_i runs such a block. Being at depth P, we know that we have reached limit L, and thus T_i will not be allowed to enqueue the new parallel inner blocks.

Instead of halting T_i, we resort to a better alternative: T_i can simply treat `parallel{b1,b2,..,bn}` as block `b1` (sequentially) followed by `parallel{b2,..,bn}`. This alternative means that any transaction starting at `b0` will now be a single child transaction, and hence it can borrow its base transaction's bitnum as its own. As `b1` completes, the same process is repeated: the free bitnum queue is checked for an available bitnum; if such a bitnum exists (we explain next how), the remaining blocks are enqueued and can finally run in parallel while, if not, the serial execution of the next block (`b2`, `b3`, ...) proceeds until the next iteration.

Of course, this option discards the parallelism among the parallel blocks in the `parallel` statement, which may seem to contradict our main goal. However, it is fundamental to observe that the reason why the system reached the L limit is that, for each non-single-child parent transaction started, there exist sufficient queued parallel blocks to allow all the remaining $P - 1$ threads to steal and run in parallel.

However, some time after such an initial moment, the remaining threads will inevitably complete the enqueued parallel blocks (and any parallel blocks arising from them). In a worst case scenario, T_i can eventually become the only thread running. Unfortunately, in such a situation, T_i will be running a set of parallel blocks (and any sub-blocks) serially while, paradoxically, the remaining threads are idle because no more blocks are enqueued.

Recall that such a pathological situation arises from the fact that all L bitnums for parent transactions have been consumed. To solve such a situation, it is crucial to observe that, once other threads complete the available parallel leaf blocks that had been queued in the context of a given parent transaction, t, only one sub-block will remain: the one containing the next parent transaction, t_{i+1}.

Therefore, whereas, originally, t_{i+1} was not guaranteed to be a single child (as there existed parallel blocks which could, in turn, initiate siblings of t_{i+1}), now t_{i+1} has become a single child. Hence, from that point on, we can make t_{i+1} discard its own bitnum and use t_i's bitnum, thus making one bitnum available for new parent transactions. Upon regaining such a bitnum, T_i will finally be able to queue the parallel blocks it currently runs in a serial manner, therefore delivering parallel work to the remaining threads.

The challenge is that, in the moment that t_{i+1}'s last sibling completes and t becomes a single child, different threads may be running in the context of t_{i+1} or any of its descendants. Therefore, such threads, each running at possibly distinct epochs, may be accessing objects and leaving at each object's stack an ancestor set containing t_{i+1}'s original bitnum. Therefore, discarding t's bitnum implies: (i) reclaiming t_{i+1}'s bitnum from all such ancestor sets, and (ii) forcing all the threads running in the context of t_{i+1} or any descendant of t_{i+1} to remove t_{i+1}'s bitnum from their ancestor sets.

Perhaps surprisingly, we achieve step (i) very easily: when the thread, T_k, of t_{i+1}'s last sibling block completes, it unilaterally discards t_{i+1}'s bitnum, as if T_k was the thread running t_{i+1}. Independently of T_k's current epoch, we know (by Algorithm 4) that, eventually, t_{i+1}'s bitnum will be set in all committed masks up to an epoch, m, that is greater or equal than the current epoch of any thread running in the context of t_{i+1} or any descendant of t_{i+1}. In other words, we achieve goal (i), as all accesses such threads made up to epoch m will have t_{i+1}'s bitnum automatically filtered out by the committed masks.

More precisely, this implies lines 9 and 10 of Procedure *finishBlock* from Figure 2. This is, however, insufficient since the threads running in the context of t_{i+1} or any descendant of t_{i+1} can evolve to later epochs than m and continue accessing objects using t_{i+1}'s bitnum in their ancestor sets. Hence, we need to force each such thread to erase t_{i+1}'s bitnum before advancing to such epochs. We ensure that by having each thread, before advancing its current epoch while executing some block, verify whether any bitnum of its current transaction's ancestor set has been published as committed in the meantime. Hence, we must add the following line before every time we change a thread's current epoch:

$$T_i.tx.anc- = (discarding + comMask[T_i.ep]);$$

This happens in Procedures $stealBlock$ and $commitTx$ (Figures 2 and 3, respectively).

It is easy to show that, if the next $T_i.ep$ is greater than m, then t_{i+1}'s bitnum is necessarily set in the $discarding$ vector or the committed mask of the epoch at which $commitTx$ is being called. Therefore, we ensure that any thread running in the context of t_{i+1} or any descendant of t_{i+1} will erase t_{i+1}'s bitnum from the thread's current ancestor sets before the thread reaches epoch $m + 1$.

To sum up, with a few lock-free modifications to the algorithm, we ensure that, as the L limit is reached and, subsequently, the parallel blocks that have been queued for other threads complete, the system regains the ability to make more parallel blocks available. This cycle shows that supporting a fixed number of transaction identifiers can actually be sufficient to handle unbounded trees of parallel nested transactions.

7. Evaluation

In our evaluation we try to answer two simple questions: *What is the benefit of parallel over serial nesting?*, and *Is transaction handling (begin, commit and conflict detection) performance independent of nesting depth?*

To this end we devise a simple benchmark in which a single transaction T is executed. T consists of leaf transactions T_{l_i}. Every T_{l_i} first sleeps for a random period of time (up to 2s) and then writes to 2000 shared objects. The first half of objects accessed by T_{l_i} is also accessed by $T_{l_{i-1}}$ and the second half is accessed by $T_{l_{i+1}}$. In order to execute T atomically, an STM that only supports serial nesting has to execute all T_{l_i} in a single thread, one

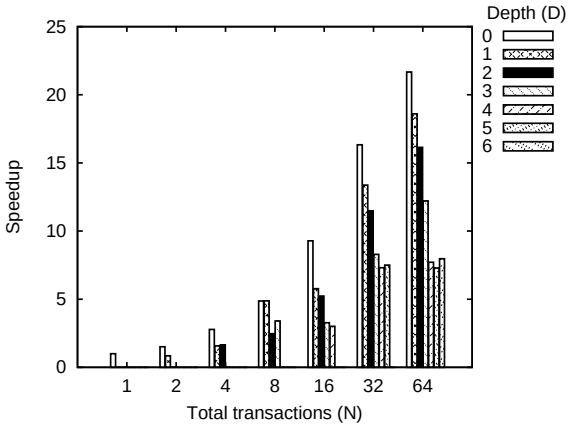

Figure 6. Speedup of our parallel nesting vs. serial nesting using our algorithm.

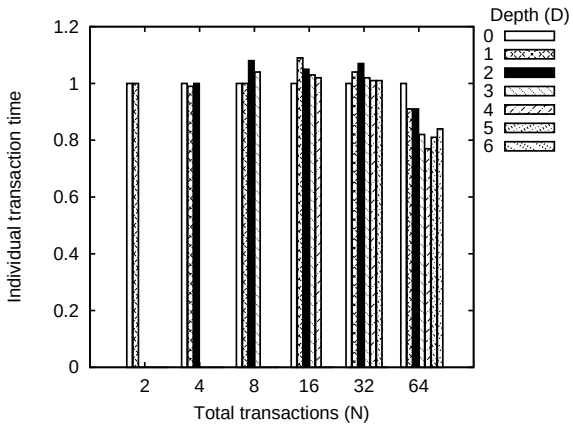

Figure 7. Time to begin/access objects/commit on average over all transactions, considering different levels of nesting (normalized relatively to depth 0).

by one. On the other hand, an STM that supports parallel nesting can execute different T_{l_i} concurrently in different threads and thus speed up T, while still executing it atomically. Our experimental setup ensures: (1) transactions that are sufficiently long to measure their duration with enough precision (2000 objects), (2) overlapping write sets to avoid trivial conflict detection (where the object stack is empty and the ancestor query is skipped) and (3) not too many conflicts (randomized wait of 1s on average).

We use different numbers of leaf transactions N and organize them in trees of parallel transactions of different depths in our experiments. With serial nesting all leaf transactions are executed serially inside a single transaction (this means there is no parallelism). In this case work stealing is disabled (one thread runs all transactions serially without any dequeuing or locking after each transaction completes) and conflict detection always implies reading the top object stack and verifying that it is empty (which is always the case in the benchmark).

With nesting depth of D, transactions are organized in a binary tree that is D levels deep. Each leaf of this binary tree executes $N/2^D$ transactions in parallel. This means that when D is 0, all T_{l_i} are children of the root transaction and they are executed in parallel, when it is 1 there are two transactions that are children of the root transaction, each of which executes $N/2$ leaf T_{l_i} transactions as its children in parallel, etc.

We ran our experiments on a SPARC Niagara 2 machine that supports 64 hardware threads. We use at most 32 worker threads and bit vector size of 64 bits because our prototype implementation does not support more than that.[6] We repeat each experiment 10 times.

Figure 6 shows the speedup of our algorithm with parallel over serial nesting for different values of D. The x-axis shows the total number of transactions executed and the y-axis shows the speedup over serial nesting. Each data point in the graph depicts the speedup with different D (it does not make sense to have $2^D > N$ and this is why not all data points appear for all transaction counts). The figure depicts that parallel nesting yields performance improvements already with two total transactions and $D = 0$. As the number of total transactions increases, the performance benefits are higher. On the other hand, a single level of nesting ($D = 0$) performs better than higher levels of nesting for all transaction counts. This is a consequence of two factors that are out of the scope of our contribution.

[6] An extension to this is straightforward.

First, additional depth implies more blocks and inter-block dependency, thus more work stealing synchronization overhead. In other words, the critical path (T_∞) is larger, hence the $O(PT_\infty)$ term from the upper bound mentioned in Section 1 inevitably becomes visible. This is a consequence of Agrawal et al.'s upper bound [1], and we do not avoid it.

A less evident secondary effect of depth is the increase in conflict ratio. In fact the experiments show that, for the same set of leaf transactions (accessing some fixed write-set), conflicts are more frequent if such transactions are distributed on a deeper tree. Intuitively, this is explained by the fact that, in shallower trees the deepest common ancestor of any pair of transactions will, on average, be at a smaller distance from both transactions than in a deeper tree. Hence, if both transactions share an object in their write-sets and one accesses it and commits, the object will propagate faster to the write-set of the ancestor of the second transaction; thus, the period before the second transaction can safely access the shared object is shorter in shallower trees, on average, which implies less conflicts, hence less aborts.

The main contribution of our paper, however, related to the second term of the asymptotical upper bound in Section 1, concerning the work, T_1, performed by a program (outside its critical path). In order to ensure that T_1 completes in $O(T_1/P)$, any transactions that T_1 may run may take the same time, no matter at which depth they run. Our results confirm the analysis of the algorithm in the previous sections. They show that the average times to begin, run and commit a successful transaction (i.e. one that does not abort) do not grow with nesting depth at which the leaf transactions run. Figure 7 shows such results for different amounts of T_1 (2, .., 64 leaf transactions) and for different depths. For the same amount of work, clearly T_1 is relatively stable and, most importantly, does not exhibit a tendency to grow with depth, thus confirming our claims.

8. Related Work

The closest algorithm to ours was proposed by Agrawal et al. for the CWSTM STM [1]. Their algorithm has shown that depth-independent conflict queries are possible but, to the best of our knowledge, has never been implemented nor evaluated in practice. CWSTM shares some general design principles with our solution, namely they employ eager validation and updates, lazy update of object stacks and the absence of explicit read/write-sets.

While following the above principles, our solution is crucially different than CWSTM. We employ a different algorithm for ancestor queries, based on bit-wise logic. Our algorithm supports a

bounded number of active transactional identifiers, whereas their conflict detection model relies on a practically unbounded transaction identifier space. Hence, most of our contribution is related to re-using transactional identifiers, which is not an issue in their algorithm. Most importantly, while asymptotically equivalent to our solution, each memory access CWSTM potentially queries a large number of data structures, expectedly yields unacceptable performance penalties [1].

More recently, new proposed solutions have moved closer towards the goal of practical parallel nesting, namely NePaLTM [16], NesTM [2] and SSTM [14]. NePalTM supports parallel threads to be forked inside a transaction. As long as the sub-transactional threads create no sub-transactions, they run in parallel, thus unveiling parallelism that serial nesting TMs prohibit. Nevertheless, should such thread create sub-transactions, NePalTM has the severe limitation of requiring such sibling transactions to run in mutual exclusion. In other words, NePalTM does not support fully-parallel nesting.

NesTM does support parallel nesting and has been implemented and evaluated. However, its transaction handling overheads (beginning, committing and detecting conflicts) grow linearly with nesting depth [2], which makes it an adequate solution for low nesting depths only.

SSTM [14] follows an alternative model of parallel nested transactions, called Xfork. Supporting the Xfork model with our algorithm is straightforward. Although fully supporting the Xfork model requires solving the same ancestor query test we address, the authors do not describe how nor whether SSTM solves it efficiently.

9. Conclusions

Support for parallel nesting is intrinsically difficult because unbounded depth is a crucial requirement for the novel and ambitious paradigms that parallel nesting promises to enable. Although recent solutions try to reach closer to the goal of parallel nesting, they fail to achieve the lowest theoretical upper bound known for the problem [1].

In this paper we propose a novel solution that, to the best of our knowledge, is the first solution that meets the upper bound while, in practice, imposing reasonably low overheads. Experimental results, obtained by running a complete implementation of our algorithm on a multi-core machine, show substantial speed-ups (up to 22x with 33 threads) relatively to serial nesting, and support the hardest requirement to meet the upper bound: that the time to start and commit transactions, as well as to detect conflicts, is independent of nesting depth.

While this paper focuses on the central problem of efficient conflict detection and handling of write-only transactions, leveraging parallel nesting in transactional memory exposes other new research directions that, to the best of our knowledge, remain unsolved. Firstly, efficient support for parallel nesting when read accesses can occur is a harder problem, since one wants to optimize such accesses by allowing multiple (possibly conflicting) transactions to simultaneously read from a common object. The main consequence is that the conflict detection test must be extended to answer ancestor queries between one transaction and *a set of multiple transactions*. Ensuring this efficiently is not trivial.

Secondly, as some authors already claim, the semantics of fork-join parallel nested programs are not always intuitive to the general programmer, thus simpler language constructs such as Free Objects [7] or Xfork primitives [14] are highly desirable.

Finally, traditional contention managers [15], oblivious of parallel nesting, are not adequate when one considers parallel nested programs. For instance, assume that some transaction, T_i, is running in parallel with one of its children transactions, T_k, and both wish to write to the same object. Contention managers that are oblivious of parallel nesting could decide to abort T_i. However, clearly such an option is not adequate, since aborting T_i would implicitly abort every children of T_i, including T_k. Novel contention managers that are aware of the ancestor-descendant relationships would avoid such a pathological decision.

References

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

[2] 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.

[3] R. Blumofe, C. Joerg, B. C. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. *SIGPLAN Notices (Proceedings of the 2006 PLDI Conference)*, 41(6):1–13, 2006.

[5] J. Chung, C. Cao Minh, B. Carlstrom, and C. Kozyrakis. Parallelizing specjbb2000 with transactional memory. In *Workshop on Transactional Memory Workloads*. 2006.

[6] B. Goetz. Java theory and practice: Managing volatility. IBM developerWorks, 2007.

[7] R. Guerraoui. A Smooth Concurrency Revolution with Free Objects. *Internet Computing*, 11(4):84.87, 2007.

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

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[10] M. Isard and A. Birrell. Automatic mutual exclusion. In *HOTOS'07: Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, pages 1–6, 2007.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[12] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting Nested Transactional Memory in LogTM. *SIGPLAN Notices (Proceedings of the 2006 ASPLOS Conference)*, 41(11):359–370, 2006.

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

[14] 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.

[15] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th annual ACM Symposium on Principles of Distributed Computing*, pages 240–248, 2005.

[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)*, pages 123–147, 2009.

[17] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–34, 2009.