A Lazy Approach for Supporting Nested Transactions

Lee Baugh and Craig Zilles
Dept. of Computer Science, University of Illinois at Urbana-Champaign
E-mail: {leebaugh, zilles}@cs.uiuc.edu

Abstract

Transactional memory (TM) is a compelling alternative to traditional synchronization, and implementing TM primitives directly in hardware offers a potential performance advantage over software-based methods. In this paper, we demonstrate that many of the actions associated with transaction abort and commit may be performed lazily *– that is, incrementally, and on demand. This technique is ideal for hardware, since it requires little space or work; in addition, it can improve performance by sparing accesses to committing or aborting locations from having to stall until the commit or abort completes.*

We further show that our lazy abort and commit technique supports open nesting and `orElse`*, two language-level proposals which rely on transactional nesting. We also provide design notes for supporting lazy abort and commit on our own hardware TM system, based on VTM.*

1. Introduction

The industry-wide shift to multicore processors has renewed research interest in synchronization methods. One leading technique, *transactional memory* (TM), exposes a much simpler programming interface than the traditional lock-based method, and potentially improves concurrency by speculatively overlapping critical sections. A transaction consists of a region of code, encompassed by a `begin_transaction` and an `end_transaction` (or some similar denotation). A TM system must guarantee that upon executing its `end_transaction`, a transaction appears to have committed atomically.

One of the attractive features of TM is a transaction's ability to compose. For example, under lock-based synchronization, if a library function called from inside a locked region itself acquires a lock, then care must be taken to avoid deadlock. With transactional synchronization, the programmer need know nothing about the atomic behavior of the functions it calls. However, to guarantee this, hardware TM systems must support nested transactions. *Collapsed nesting*, a trivial technique for handling transaction nesting employed by most hardware TM (HTM) systems, including LogTM [6], VTM [8] and UTM [1], involves subsuming all nested transactions into their top-level ancestor. In this scheme, the nested `begin_transaction`s and `end_transaction`s are ignored, except to update a nest depth counter; only when this counter is 0 after an `end_transaction` does a transaction commit.

The alternative to a policy of collapsed nesting is *closed nesting*[1], in which only conflicting transactions and their descendants – not their entire ancestry – are prone to being rolled back. Closed nesting offers a potential performance advantage in, for example, the case of a long transaction seeking to acquire a highly contended lock for some resource: enclosing the lock acquisition in a nested transaction of its own limits the cost of conflicting on the contended lock to only a few rolled-back instructions. Other performance justifications for supporting closed nesting may be conjectured; however, it must be noted that after a nested transaction commits, its entire read and write set are coalesced into its parent, exposing the parent to conflict for the rest of its life.[2]

In addition to the potential performance benefits, nesting is used by two proposed programming constructs: open nesting [7] and `orElse` [3]. Open nesting provides a means of committing a nested transaction directly to architected state, freeing its ancestry from risking conflict on its accesses after it has retired. `orElse` may be used to indicate a set of transactions, of which exactly one

1 The name *closed nesting* is introduced in [7], but in our work the term refers to a special case referred to in that paper as *linear nesting*.

2 For this reason, it would seem to be a good design choice to give longer transactions, in which more work has been invested, priority when resolving conflicts.

will commit.

There have been numerous proposals for TM systems, both hardware-based (such as [6], [8], [1], [2], [5], and [7]) and software-based (such as [3], [4], and [9]). While it has always been a primary goal of HTM systems to offer good performance, HTM proposals have also been becoming more feature-rich. Many now support arbitrarily large transaction footprints, and many support transactions living past context switches or migrating to different processors. Moss and Hosking's recent work introduces hardware support for closed nesting [7]. As the flexibility of HTM proposals increases, so may the viability of transactional memory as a whole.

In this paper, we introduce a new, lazy, method of handling the overflowed transaction state of aborting and committing transactions. Lazy abort and commit permit values belonging to aborted or committed transactions to be used by other memory accesses as soon as their transactions *logically* abort or commit – no access must wait for a physical abort or commit to complete. This technique also relies much less heavily on mechanisms which walk the overflowed transaction log, and requires very little computation or state, so it is quite suitable for HTMs seeking to keep their performance high.

We further demonstrate how our lazy abort and commit technique may be extended to support the abort and commit (or *coalesce*) of closed nested transactions, and may be further extended to support open nesting and `orElse`. Finally, we will give design notes concerning implementation of lazy abort and commit in our own hardware TM system.

2. Transactional Memory

In this section, we review what is involved with creating and committing transactions, as well as what must be done when a transaction reads data, writes data, and aborts. We then review the `retry` primitive and discuss what supporting closed nesting involves. We then introduce lazy abort, commit, and coalesce.

2.1. A Transactional Primer

Transactions, like traditional lock-based critical sections, are regions of code which, upon exiting, are guaranteed to have appeared atomic to all processes. However, transactions, unlike locks, may be executed concurrently. Should there be a conflict, then at least one running transaction must be squashed or stalled.

Transactions are started with a `begin_transaction` command. When a TM system sees such a command, it records in local memory that a transaction has begun. In this paper, we will refer to this record as the local transaction state (LTS). In the LTS, the TM system may store transaction-specific data such as the transaction's current state, a register checkpoint taken just before the `begin_transaction`, or information it can use to resolve conflicts between transactions, such as a timestamp.

When a memory read is performed inside a transaction, the TM system must add the accessed block to the transaction log. If there is already a read entry for this address and the current transaction in the transaction log, then nothing new needs to be added to the log. This log, which is part of a structure we call the global transaction state (GTS), is logically shared among all transactions. The precise organization of this log varies between TM proposals: in some software TM systems the log is a software table ([3], [4], [9], etc.). In the original HTM proposal by Herlihy and Moss, the log resides entirely in caches [5]. More recent hardware proposals permit the transaction log to overflow from the cache into user-level memory ([8], [1], etc.). The following discussion supposes an unbounded hardware system similar to VTM [8], but this is merely a convenience – the same techniques could be applied to other types of TM system.

The entries in the transaction log – each representing a read or write access to a piece of memory by some transaction – contain some reference to the transaction to which they belong (*e.g.* a pointer to the transaction's LTS), the address of the accessed data, and, in the case of writes, a space to store data – for a write in a running transaction introduces two copies of data for the same address: a transactional copy and a nontransactional copy.

Memory writes inside a transaction are somewhat more complicated. As with transactional reads, the write must be added to the footprint of the transaction. Since transactional writes, as mentioned, introduce two values for the same address into the system – one transactional and one nontransactional – both values must be stored, one in the transaction log and one in architected state. Here there are two design choices. In *eager versioning*, the original, nontransactional value is stored in the log, and memory is optimistically updated with speculative state. In *lazy versioning*, the speculative value is stored in the log, and the memory remains untouched. The former policy, first proposed for HTMs in LogTM [6], streamlines the commit case while requiring some extra work for aborts; the latter, used in UTM [1], VTM [8],

and others, streamlines aborts while requring some extra work for commits. As our own TM system logs nontransactional state, the following discussion will assume eager versioning – however, either policy is suitable for all techniques we will describe. Thus, in our scheme, when a transaction writes an address it has already written before, then the old entry must be left alone, as only the old entry has the correct nontransactional value.

Transactions are committed when their control flow reaches an `end_transaction` instruction. When the TM system sees such a command, then it must atomically commit the entire transaction. There are several ways to do this, but the method we describe (following VTM) consists of two parts: the logical commit, wherein the state of the transaction is changed to Committing, and the physical commit, wherein all entries for the committing transaction are removed from the log. Once the entire physical commit has completed, the committing transaction may be shifted to state nontransactional (NonT), whereupon it senesces. As this transaction's write accesses have speculatively been written through to architected state, nothing more than this is required.

Finally, transactions may conflict. While some conflicts can be resolved by momentarily stalling one of the conflicting processes, sometimes it is necessary to abort one of the transactions. When this happens, one or more losers must be determined. If a nontransactional access conflicts with a set of transactions[3] then all transactions must lose. If two (or more) transactions conflict, then some kind of priority, such as timestamp, can be used to determine which transactions win and which lose.

Although optimizations exist wherein losing transactions are simply paused until the winners have retired, we will focus here on the case where the losing transactions must abort. When a transaction is aborted, it is first logically aborted: its state is changed to Aborting and its registers, including the PC, are restored to the initial checkpoint. Then, it is physically aborted: all entries for the aborting transaction are removed from the log. Read-only entries are, as with commit, simply removed, whereas write entries will need to be rolled back to memory – that is, their data field must be written back to architectural state, replacing the speculative value there – before they are removed from the transaction log. Once the physical abort has completed, the aborting transaction may be shifted to state NonT.

We summarize the actions that must be taken on read and write log entries for aborting and committing transactions, under eager versioning, in Figure 1.

	Abort	Commit
Write	Roll back & Remove this entry	Remove this entry
Read	Remove this entry	Remove this entry

Figure 1. Actions required for log entries of aborting and committing top-level transactions, under eager versioning

2.2. `retry`

A language-level transactional primitive called `retry` is introduced in Harris' *et al.* work on composing transactions [3]. If a programmer can, by examining within a transaction the current architected state, determine that the current instance of that transaction cannot possibly commit, then a `retry` may be inserted into the transaction. When control flow in a transaction reaches a `retry`, the transaction is descheduled (but left in state Running) and all its writes are rolled back. A `retry`ing transaction loses all conflicts, so that only when something in its readset is written will it be reawakened, aborted, and reexecuted. The `retry`ing transaction therefore wastes no CPU cycles spinning.

With respect to the transaction log, a `retry`ing transaction is simply one which loses all conflicts. Thus, as pertains to the log, handling `retry` devolves to handling aborts.

2.3. Nested Transactions

We now describe the behavior of closed nested transactions. This discussion is based on the work of Moss and Hosking [7], which also introduced the term 'closed nesting'.

When a running transaction executes a `begin_transaction`, then a nested transaction has begun: a new LTS is allocated, a parent field in it is set to the current running LTS, and a child field in the current running LTS is set to the new LTS.

Memory accesses, both writes and reads, are exactly the same inside a closed-nested transaction as their top-level counterparts: each access gets its own log entry just as

3 There could be more than one conflicting transaction, as in the case of a nontransactional write conflicting with a set of transactional reads.

is described above. If a nested transaction accesses an address which one of its ancestors accessed, there is of course no conflict; instead the nested transaction creates a new log entry. Aborts and commits, however, are different.

When a closed nested transaction commits, it should be *coalesced* into its parent transaction. By this we mean that its entire readset and writeset should be merged into its parent's. When the TM system sees an `end_transaction`, and the current running LTS C has a (non-`NULL`) parent P, then a nested transaction needs to be committed. As before, the state of the running transaction is set to Committing. This thread's current running LTS is set to P, as the parent is now the current running transaction for this thread. Then, the transaction log for C must be coalesced into the transaction log for P. Writes in C's transaction log are not copied into P's transaction log unless P has no writes to that address. Any reads in C's transaction log are copied into P's transaction log, unless P already has a write access to that address.

When a closed nested transaction loses a conflict and aborts, then its entire effect must be undone. As before, its state is changed to Aborting and the register checkpoint is restored. It is possible that a nested (or indeed top-level) transaction with descendants of its own loses a conflict. So, when a transaction is aborted, all its descendants must recursively be aborted. To abort a nested write, we must roll it back and remove its entry from the log. It is possible that several descendants have all written the same address; in this case, all these writes must be rolled back in a LIFO manner, with the deepest children rolling back their writes first. However, if we force all transactional accesses to read or write at some block granularity, and ensure that every accessed block gets its own transaction log entry (such that a very large single memory access might produce several transaction log entries, and a very small memory access might induce false sharing) then it is sufficient simply to roll back the shallowest aborting write to a given address, and then remove all aborting writes for that address from the transaction log. All reads of nested, aborting transactions are also simply removed from the transaction log.

We summarize, as before, the actions that must be taken on read and write log entries for aborting and committing nested transactions in Figure 2. Differences from top-level aborts and commits are in italics.

	Abort	**Commit**
Write	*If shallowest aborting entry for this address, roll back; In any case remove this entry*	*If DRA has no write entry for this address, coalesce up; In any case remove this entry*
Read	Remove this entry	*If DRA has no entry for this address, coalesce up; In any case remove this entry*

Figure 2. Actions required for log entries of aborting and committing closed nested transactions, under eager versioning. *DRA stands for 'deepest running ancestor'*

2.4. Lazy Abort, Commit, and Coalesce

In the TM model given above, actions fall generally into two types: transaction state actions and transaction log actions. Transaction state actions *logically* change the transaction state. For example, in VTM, changing a transaction's state to Committing logically commits the transaction, though the actual *physical* commit is not complete until their log walker (a mechanism which iterates through the transaction log physically aborting and committing transactional accesses) has physically committed every log entry belonging to the committing transaction – at which point the transaction's state is shifted from Committing to NonT.

Such a physical commit cannot be performed instantaneously. In VTM, which uses lazy versioning, when a memory access conflicts with a transaction in state Committing, then the access is stalled until the physical commit is complete. The VTM proposal suggests that the access must be stalled, but actually, it is possible to avoid this stall.

In VTM, upon any kind of memory access at all, if there is projected to be[4] a conflicting entry in the transaction log, the log must be searched to find the conflicting access so that the proper action may be determined.

Therefore, any access which conflicts with a committing transaction has already had to do a log lookup to determine that its conflictee is logically committing. It could simply read the logged value and get on with its execution. This action could trigger a background physical commit for this entry, coupled with the entry's removal from the transaction log.

The same technique may be applied to the eager-versioning TM which we have been describing. How-

4 The projection is due to VTM's transaction filter (XF) and overflow counter, and is beyond the scope of this paper.

ever, as it supports closed nesting, there may be several aborted writes to the same location. Care must be taken to roll them back in the proper order: again, if accesses are forced into a block granularity, it is sufficient to roll back only the oldest (shallowest) aborted write to any location.

Log entries of committing transactions may be lazily handled by simply being removed from the log whenever they are encountered in a log lookup.

In Figure 1 and Section 2.2, we summarized the actions that must happen *to transaction log entries* on aborts, commits, and `retry`. We presented these actions at the log entry level in anticipation of this kind of *lazy abort and commit*. Indeed, the lazy technique may be applied to nested transactions as well; once a nested transaction is *logically* committed, it may be *physically* coalesced lazily, by following the actions in Figure 2.

When a nested transaction commits lazily, its state is immediately shifted into state Committing. When a memory access conflicts with one of its transaction log entries, the LTS pointer in that entry is changed to the committing transaction's parent. If the parent already has an entry for the given address, it is overwritten only if the parent's entry is a read and the committed access is a write. The committing transaction's state may ultimately be shifted into NonT (releasing that LTS for reuse) when a low-priority background walker has moved through every transaction log entry, lazily aborting and committing entries, at least once since the transaction began committing.

When a nested transaction aborts lazily, its state is immediately shifted into state Aborting. Since an aborted transaction with descendants aborts all its descendants as well, it is possible that a memory access will conflict with several aborting entries for the same address. In this case, the oldest aborting write must be rolled back. Then, it and all other aborting matches are dropped from the transaction log.

To a reader, the correctness of lazy abort and commit is perhaps not immediately obvious. We provide a set of invariants which ensure that lazy abort and commit works properly:

- A correct value is unambiguously available between memory and the transaction log.
- If there are N transactional writes to an address logged, there must be $N + 1$ versions of the data at that address between memory and the transaction log, so that any write can be rolled back.
- Every access to an address which exists in the transaction log must logically perform a log lookup.[5]
- Whenever a log lookup finds aborting or committing entries, they must all be lazily aborted or committed then.
- If there is a set of aborted writes for an address in the log, the correct value will be in the log entry for the oldest aborted write.
- The only way multiple aborted writes for an address may appear in the transaction log is if they are ancestrally related in a direct lineage. That is, given the set S of transactions to which the multiple aborted write entries for the same address belong, every transaction in S is either an ancestor or a descendant of every other transaction in S.
- Then, the oldest log entry will necessarily be the entry belonging to the most ancestral transaction – that is, the transaction which has no ancestors in S.

It is also possible to lazily handle `retry`. When a transaction lazily `retry`s, its state is shifted to state Retry_Running, and the transaction is descheduled. When a memory access conflicts with one of the `retry`ing transaction's accesses, the `retry`ing transaction is immediately shifted to state Aborting, restarted, and the conflicting log entry is dealt with as an aborting entry.

In other HTM proposals such as VTM and UTM the efficiency of the log walker mechanism is of cardinal importance, since conflicting accesses may be forced to wait on it. With lazy abort, commit, and coalesce, the log walker may no longer strictly even be necessary. As mentioned, in our own HTM implementation we still employ a transaction log walker, but its purpose is only to permit us to recycle LTSs and to ensure that the log stays efficient. We shall discuss this in more detail in Section 4.

2.5. Lazy Abort & Commit – An Example

We clarify lazy abort and commit by means of an example. Suppose the following transactional program:

```
1  a = 10; b = 10;
2  begin_transaction();   // trans P
3    a = 1;
4    begin_transaction(); // trans A
5      if (f()) {
6        b = a;
```

5 Some proposals cache part of the transaction log; if the cache can guarantee these invariants about the log then it is not a problem.

```
7         abort;
8       } else {
9         a = a + 1;
10      }
11    end_transaction();   // trans A
12    begin_transaction(); // trans B
13      a = 2;
14    end_transaction();   // trans B
15  end_transaction();     // trans P
16  c = a * b;
```

The first log entry is created at line 3. It contains an indication of to which transaction it belongs (P), with what address it is concerned (`a`), and the original value at that address (10):

P (Running)	a	10

After `f()` evaluates to `true` on line 5, the second log entry will be created at line 6, and belongs to transaction A. Before it can create this entry, though, transaction A needs to find the proper value of `a`. Since there is a log entry for `a`, transaction A first looks it up. It observes that the transaction owning that entry is an ancestor, so it does not conflict, and that it is running. This means that it can find the correct value for `a` in memory, where transaction P put it[6]. The transaction log is now:

P (Running)	a	10
A (Running)	b	10

Now transaction A loses a conflict on line 7[7] and must restart as A' (as it is given a new LTS). Upon its second execution, `f()` evaluates to `false`, and transaction A' adds a new entry to the transaction log:

P (Running)	a	10
A (Aborting)	b	10
A' (Running)	a	1

Note that the entry for `a` under transaction A' has as its original value the (speculative) value left there by transaction P. Now transaction A' logically commits without complication and transaction B begins. Upon its attempt to write `a` on line 13, it finds the committing entry of transaction A' in the transaction log, which is therefore lazily committed. Since the deepest running ancestor of A', P, already has a write to `a`, no coalesce is needed; the committing entry is simply removed from the log. The log now:

P (Committing)	a	10
A (Aborting)	b	10
B (Committing)	a	2

Transaction B and P now logically commit, and the nontransactional code at line 16 is executed. It must access the transaction log for both `a` and `b`. For `a` it finds two committed writes, which are then lazily committed by being removed from the transaction log; as they are committed, the correct value is found in memory. For `b` it finds an aborted write. The correct value for `b` is forwarded from the aborted entry to the nontransactional access, and the entry is lazily aborted by writing back its original value to memory and then being removed from the transaction log. After line 16, the transaction log is once again empty.

3. Open Nesting and `orElse`

We now describe two language-level proposals which improve transactional performance and capability: open nesting and `orElse`. We show that these techniques can, like abort, commit, and coalesce, be handled lazily.

3.1. Open Nesting

In Section 2.3, we discussed how a committing nested transaction must coalesce its readset and writeset up into its parent. This exposes the parent to conflicts on addresses the committed, nested child accessed. Moss and Hosking have proposed a technique, *open nesting*, which provides a way of avoiding these conflicts [7].

An open nested transaction is exactly like a closed nesting transaction on `begin_transaction`, transactional reads and writes, and aborts. However, when an open nested transaction commits (which event we shall call *open committing*,) its writes are allowed to commit to architected state, and all addresses which it wrote are removed from its ancestors' transaction logs. An open committing transaction may also register with its ancestry *compensation code* which is to be run should an ancestor be aborted. Open nesting can compose, so if multiple compensation codes are registered, they must be executed in LIFO order.

This technique can improve concurrency by reducing the length of time a transaction is exposed to conflict on some memory locations, but it constitutes a violation of atomicity and care must be taken by the programmer that no harm will come from other transactions or nontransactional code accessing open committed data.

6 From transaction A's point of view, the value transaction P placed in memory is the correct one, since if transaction P is aborted, so too will be transaction A.

7 Actually it explicitly aborts, but the effect is the same.

Because of this violation of atomicity, open nesting is not a technique automatically and transparently provided by a TM system. It is expected that the open nesting semantics will be fully visible at the language level, and that the programmer will not only provide suitable compensation code, but will further ensure that other transactional or even nontransactional code which may view open committed data that is later revoked will still function correctly. Nonetheless, the technique provides substantial opportunity for optimization in certain classes of problems where contention is strong but localized and compensation can be readily performed.

As we have said, until it commits, an open nested transaction is identical to any other transaction. Since the compensation code expects to compensate for a completed transaction, an open nested transaction must not write anything to architected state until it commits; yet when it commits, only those writes inside its region should be written to architected state.

In Figure 3, we show the behavior of log entries for an aborted or committed open-nested transaction, with differences from closed-nesting behavior shown in italics. We observe that this behavior can also be performed lazily. When an open nested transaction aborts, it may be lazily handled just as described in Section 2.4. When it commits, its state is immediately shifted into state Open Committing. When a memory access conflicts with one of its transaction log entries, that entry is lazily open committed. If it is a write, then every entry to the same address in the transaction log is checked. Those which belong to ancestors of the open committing transaction are removed from the log, as is the open committed entry. Open committed read entries are simply lazily removed from the log.

	Abort	**Commit**
Write	If shallowest aborting entry for this address, roll back; In any case remove this entry	*Write to architected state, remove all ancestral entries for this address, remove this entry*
Read	Remove this entry	*Remove this entry*

Figure 3. Actions required for log entries of aborting and committing open nested transactions, under nontransactional logging

3.2. `orElse`

In their work on composable transactional memory [3], Harris *et al.* introduce several novel extentions to language-level atomicity in the context of a software TM system for the Haskell functional programming language. Among these extensions is the primitive `orElse`.

In Section 2.2, we described the `retry` command. Harris *et al.* observed that `retry` can be used for more than optimizing an event loop. If, instead of simply descheduling on a `retry`, a process provided an alternate transaction to attempt, then several `retry`s could be composed – much like the Unix `select` syscall composes file-event loops. The code {`transaction_A() orElse transaction_B()`} results in either A or B being executed. A is attempted first; if it commits then the whole block is exited. If, however, A executes a `retry`, then B is attempted. If it should commit, then the whole block is exited (without committing the actions performed by A), but if B reaches a `retry` as well, then the whole block stalls as if on a `retry`, its readset being the union of the readsets of both A and B.

When a transaction under `orElse` lazily commits, then its log behavior is exactly like a closed nested transaction lazy commit, as described in Section 2.4. Likewise when it lazily aborts. However, should it execute a `retry`, this can be handled lazily as well.

In our implementation of `orElse`, when a transaction directly (not transitively) containing an `orElse` begins, it has an `orElse` bit set in its LTS. Transactions directly containing an `orElse` have a `retry` placed immediately before their `end_transaction`. It is the job of the HLL to ensure that should A or B commit, execution is continued from after their enveloping transaction's `end_transaction`.[8]

When a transaction lazily retries under orElse, it is immediately descheduled and its state is shifted to state OrElse_Running. When a memory access conflicts with one of the retrying transaction's accesses, then its ancestry is walked, from its parent up towards the top-level ancestor, until it finds a transaction T whose parent is either `NULL` or has an unset `orElse` bit. Then, T is aborted, which aborts all its descendants including the conflictee. Finally, the conflict is handled as a nested abort. Note that the ancestry traversal allows `orElse` to compose.

8 It is also the HLL's job to ensure that when a transaction retries under an `orElse`., the process is not descheduled but instead is continued after the `retry`ing transaction's `end_transaction`.

Whether the conflict spurred a wakeup or not, the OrElse_Running entry which conflicted must be rolled back (if it is a write) and in any case removed from the transaction log.

Care must be taken in the case that A reads address M, then B writes M. As `orElse` was introduced as an extension of the functional language Haskell, this case does not occur there. However, in languages which feature destructive update, a policy must be decided – should B's write trigger a wakeup of A, or not? Either case can be handled lazily, but a more detailed discussion of the two policies is beyond the scope of this paper.

4. Our HTM Implementation

We have implemented a HTM system, based on VTM [8], which performs lazy commit, abort, coalesce, open nesting, `retry`, and `orElse`. In this section, we will describe some of our implementation details.

4.1. VTM

The original hardware TM proposal of Herlihy and Moss [5] constrained the transaction log to reside in the caches, which both limits the amount of state a transaction may touch and constrains transactions never to live past a context switch. VTM is an unbounded HTM proposal which permits transactions to overflow to a transaction log stored in virtual memory. This log, called the transaction address data table or XADT, is arranged as a hash, to facilitate lookups. VTM transactions are associated with a transaction status word (XSW), which is analogous to the state component of our LTS. VTM also supports transaction logging in the cache, only overflowing to the XADT when transactions are context-switched out or when their footprint no longer fits in the cache.

4.2. Alias Tables

In order to support nesting, our TM system must allow for multiple transaction log entries for the same data address. In general, it should be possible to achieve this by overloading whatever method a TM system uses to permit multiple readers. As VTM's method for supporting multiple readers is not defined, we have developed a solution which we call *alias tables*. These are arrays of XADT entries whose allocation (and growth) is managed by software, and which contain multiple log entries, all of which pertain to the same address. Alias tables are linked to the XADT using an XADT entry whose address field contains the address referred to by the alias table's entries, but which also contains a pointer to the alias table. One advantage of these alias tables is that there is strong temporal locality among aliasing entries – when one entry for an address is read, others for that address are likely soon to be read – and such a table will keep them close together. When several entries all alias to the same address in our XADT, they are guaranteed to be collocated on a common alias table. Almost all lazy operations may be accomplished with only one pass over the alias table.

For example, when an entry belonging to a closed nested transaction is lazily committed, a pass is made over its alias table to see if there is an entry for this address under the committing transaction's deepest running ancestor. If there isn't such an entry, then the entry's LTS is changed to point to that deepest running parent's LTS; if there is such an entry, then the committing transaction's entry is coalesced into it as described in Section 2.4.

4.3. Software Support

VTM makes use of software[9] in allocating virtual memory space for its various structures, primarily the XADT. Our system also requires software support for these structures but additionally leaves LTS management to software. As suggested in [8], our system maintains a pool of LTSs. As mentioned in Section 2.4, transaction state is only ever shifted to NonT by a software XADT walker. This walker is responsible for deciding when a committing or aborting transaction no longer has any XADT entries, and releasing that transaction's LTS to the free LTS pool.

5. Conclusion

Most unbounded hardware TM proposals have avoided attempting to support closed nesting, both because it was not perceived as critical to performance and because it was not straightforward to do so. In their paper on nested transactional memory [7] Moss and Hosking introduced an HTM system which could support closed nesting, and showed how it could be extended to support open nesting, a proposed method for reducing the time that critical transactional values remain vulnerable to conflict.

9 Whether OS or user-level is not specified. Either is possible, and arguably user-level software is preferable

Open nesting is indicative of a compelling justification for true – that is, closed nesting – support in any TM system, hardware or software: nesting tends to enable interesting and useful language-level transaction features. Nor is open nesting the only such feature. `orElse`, proposed by Harris *et al.*, is a method of composing alternative transactions, and it also requires closed nesting support. As it was first described in the context of the functional language Haskell, we have shown what is required to support `orElse` in languages with destructive update.

Hardware transactional memory support has, as its primary advantage over the otherwise more flexible software transactional memory, the potential for superior performance. To this end, we have introduced a technique for lazily aborting and committing transactions which eliminates the need for memory accesses to stall when they conflict with committing or aborting transactions. We have shown that transactions may also be lazily coalesced, permitting nested transactions also to abort and commit lazily. Further, we have shown how the aforementioned novel techniques of open nesting and `orElse` may, too, be accomodated through lazy methods. While lazy abort and commit are not necessarily hardware techniques, they are well-suited to hardware, as they can be accomplished incrementally, and in parallel with execution.

Finally, we have detailed some aspects of our own HTM implementation which permit it to perform efficiently lazy abort and commit.

References

[1] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, Feb. 2005.

[2] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, June 2004.

[3] T. Harris, S. Marlowe, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Principles and Practice of Parallel Programming (PPOPP)*, 2005.

[4] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Symposium on Principles of Distributed Computing (PODC)*, 2003.

[5] 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, May 1993.

[6] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2006.

[7] J. E. B. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proceedings of the workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[8] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[9] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.