Supporting Nested Transactional Memory in LogTM

Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen,
Mark D. Hill, Ben Liblit, Michael M. Swift and David A. Wood
Department of Computer Sciences, University of Wisconsin–Madison
{moravan, bobba, kmoore, lyen, markhill, liblit, swift, david}@cs.wisc.edu
http://www.cs.wisc.edu/multifacet

Abstract

Nested transactional memory (TM) facilitates software composition by letting one module invoke another without either knowing whether the other uses transactions. **Closed nested transactions** extend isolation of an inner transaction until the top-level transaction commits. Implementations may flatten nested transactions into the top-level one, resulting in a complete abort on conflict, or allow partial abort of inner transactions. **Open nested transactions** allow a committing inner transaction to immediately release isolation, which increases parallelism and expressiveness at the cost of both software and hardware complexity.

This paper extends the recently-proposed flat *Log-based Transactional Memory (LogTM)* with nested transactions. Flat LogTM saves pre-transaction values in a log, detects conflicts with read (R) and write (W) bits per cache block, and, on abort, invokes a software handler to unroll the log. Nested LogTM supports nesting by segmenting the log into a *stack of activation records* and modestly replicating R/W bits. To facilitate composition with non-transactional code, such as language runtime and operating system services, we propose **escape actions** that allow trusted code to run outside the confines of the transactional memory system.

Categories and Subject Descriptors C.1.4 [**Processor Architectures**] Parallel Architectures

General Terms Design Languages

Keywords Transactional Memory, Nesting, LogTM

1. Introduction

Emerging chip multiprocessors (a.k.a. multi-core chips) are energizing interest in making multithreaded programming less painful. One promising approach is *transactional memory (TM)* [13]. A TM system lets a programmer invoke a transaction (e.g., `transaction_begin(); <some work>; transaction_end();`) and rely on the system to make its execution appear *atomic* with intermediate data *isolated*. A successful transaction *commits*, while an unsuccessful one that *conflicts* with a concurrent transaction *aborts* and may transparently or explicitly retry. Programmers may invoke transactions directly (e.g., by calling `transaction_begin()` from C) or indirectly (e.g., with statically scoped atomic blocks [8,

9, 11] or Java synchronized methods [4]). This paper focuses on systems implemented with hardware support *(HTMs)* [1, 7, 13, 21, 28]; others examine *software-only TM (STM)* systems [10, 12, 29].

1.1 Challenges

We see three major challenges to composing software modules in a transactional memory system: supporting software composition for transactions, providing high concurrency for long-running transactions that require contended resources, and invoking non-transactional language and operating system (OS) services.

Challenge 1: Facilitating Software Composition. Ideal *software composition* allows a module A to invoke a module B, which can invoke C, etc., using module interfaces rather than deep knowledge of module internals [27]. The currently prevalent method of *locking* fails to provide composition, because to ensure safety and avoid deadlock, programmers often need to know what locks are (or may be) held by caller and callee implementations. Open Solaris even warns relatively sophisticated operating system programmers that "*Blocking on a mutex is a surprisingly delicate dance*" [19].

To aid software composition, TM systems must support *transaction nesting* [22] wherein a transaction may begin and end within surrounding transactions. Straightforward nesting—called *closed nesting*—ensures atomicity and isolation until the top-level (i.e., outermost) transaction commits. Most HTMs implement closed nesting by *flattening* inner nested transactions into the top-level transaction [1, 7, 21]. `Transaction_begin()` simply increments a counter, `transaction_end()` decrements it, and commit occurs only when the count returns to zero.

While functionally correct, flat closed nesting may degrade performance. In particular, a conflict on an inner transaction may cause a *complete abort* to the beginning of the top-level transaction. A *partial abort* could improve performance, by only aborting the inner transaction and avoiding an abort of the possibly much longer top-level transaction [6].

Challenge 2: Enhancing Concurrency. Closed nesting does not eliminate all problems posed by modular software. In particular, closed transaction semantics limit concurrency by maintaining isolation until the top-level transaction commits. Consider, for example, a long-running, low-contention top-level transaction L that frequently invokes a short-running nested transaction S (e.g., a shared resource allocator). Closed nesting must maintain isolation on the resource allocator (e.g., because S updates the free list pointer) until L commits, severely restricting parallelism. Ideally, S should release the free list pointer, etc., so that other transactions can access the allocator without conflicting with transaction L.

Open nested transactions address some of the above concerns by relaxing the atomicity guarantee and managing isolation at a higher

level of abstraction [32]. When an open nested transaction S commits within an enclosing transaction L, (a) the TM system releases data read or written by S, so that other transactions can access them without generating conflicts and (b) S may register *commit* and *compensating actions* to be run when transaction L commits or aborts, respectively [9, 32]. These actions allow programmers to raise the level of abstraction by providing higher-level isolation and undo semantics [25]. For example, one can compensate for a `malloc()` by executing a `free()` (or by simply relying on garbage collection). Simply restoring the values of memory locations modified by S (e.g., the free list pointer) is insufficient, since subsequent open transactions may have modified them.

Challenge 3: Escaping to Non-Transactional Systems. Many TM systems will run on top of non-transactional base systems that may include run-time libraries, language virtual machines (e.g., JVMs), operating systems (Windows, Linux, Solaris), and even system virtual machines (VMware). TM applications may need to *escape* to such systems explicitly (e.g., system calls) or implicitly (e.g., TLB traps or interrupts). Like `asm()` statements in C, these escapes allow access to lower-level capabilities, but they should *not* be directly used by most application programmers.

STMs handle such escapes easily, because they virtualize the TM mechanisms in user-level software. HTMs, on the other hand, often use hardware conflict detection bits and/or speculative value buffers. An escape to a non-transactional system must disable these mechanisms, because such software may not operate correctly in the presence of transaction isolation and aborts [3].

1.2   Nested LogTM

This paper explores the above challenges in the context of the recently-proposed *Log-based Transactional Memory* system [21], which we call *Flat LogTM*. Flat LogTM provides hardware mechanisms for version management and conflict detection, which compilers and run-time library software use to implement language-specific transactional memory policies. Flat LogTM performs *version management* by storing new values (for commit) "in place" in cacheable virtual memory and old values (for abort) in a per-thread log, also in cacheable virtual memory. Hardware support makes commits fast, because no data must be moved, while a (library) software handler unrolls the log on aborts. Like many HTMs, Flat LogTM performs *conflict detection* using *read (R)* and *write (W)* bits in caches, but also adds mechanisms to allow cache evictions of transactional data. Flat LogTM uses a counter to flatten closed nested transactions into the top-level transaction.

Nested LogTM makes three contributions by extending Flat LogTM's mechanisms to support nesting and escape actions.

Contribution 1: Closed Nesting with Partial Aborts. Nested LogTM allows inner transactions to abort separately from the top-level transaction by segmenting the undo log and replicating the R/W bits. Each segment records old values for a single nesting level. This makes the log resemble a standard *stack of activation records*, where each frame holds the records for a given level of nesting. Nested LogTM narrows conflict detection to a specific nesting level by extending cache R/W bits to have k copies (e.g., 4) and flattening transactions nested deeper than k. Nested LogTM provides partial abort by unrolling a subset of log frames and clearing a subset of R/W bits.

Contribution 2: Open Nesting. Nested LogTM supports access to highly contended resources within a transaction with open nested transactions, which may commit and release isolation prior to the top-level transaction's commit. When an open nested transaction commits, the inner log segment is removed and commit and compensating action records are added to the parent's log segment. In addition, the inner transaction's R/W bits are cleared in the cache.

Contribution 3: Escape Actions. Nested LogTM supports calls to a lower-level non-transactional system, including the OS, from within a transaction with *escape actions*, which bypass transaction version management and conflict detection. Escape actions can be invoked explicitly via new instructions or implicitly as part of a trap. An escape action may register commit and compensating actions like an open nested transaction. Nested LogTM implements escape actions with a per-thread flag, which disables logging and conflict detection when set. We view escape actions as an enabling mechanism for supporting non-transactional system activity in HTMs. For example, trap handlers and many system calls can execute within escape actions, which both increases the variety of code that can execute within transactions and reduces the need to abort a transaction due to OS activity. In addition, escape actions are a key building block for implementing I/O within transactions and for supporting debuggers and garbage collectors.

Other recent HTM proposals have also explored support for closed nesting with partial aborts and open nesting [18, 24, 26]. We extend this work several ways, including proposing escape actions, implementing nesting in LogTM, and defining a condition for mitigating subtle open nesting semantic issues in TM systems.

We evaluate Nested LogTM versus Flat LogTM running (mostly) TM microbenchmarks on Solaris 9 on a 32-way multiprocessor simulated with an extension of Wisconsin GEMS [17]. The *Sorted List* and *B-Tree* microbenchmarks show little benefit from closed nesting and partial abort, but show significant improvement from the increased concurrency provided by open nesting. We also see little performance benefit from nesting running a subset of the SPLASH-2 benchmarks, in part due to the lack of conflicts between transactions in those workloads. The *strided array* microbenchmark demonstrates that escape actions improve performance in the presence of TLB miss traps by allowing transactions to continue after a trap.

2. A Transactional Memory Model

This section informally describes an implementation-independent model of closed nested transactional memory implemented by flattening; the remainder of this paper evolves this model to support partial aborts, open nesting, and escape actions and describes how the model maps to the Flat and Nested LogTM implementations.

Let a TM system be represented by a memory M that maps addresses a to values v and a set of threads. Each thread i has a transaction level $level_i$, a read set R_i, a write set W_i, and a value map V_i that maps addresses a to updated values v. Let $block(a)$ be a function that maps an address a to a possibly-larger aligned granularity (e.g., a memory block). Initially, M maps initial values, all $level_i$ are zero, and all R_i, W_i, and V_i are null. Thread i begins a transaction by incrementing $level_i$. Subsequent reads to address a add $block(a)$ to R_i. Writing a new value v to address a adds $block(a)$ to W_i and the mapping $<a, v>$ to V_i (replacing any

previous mapping $<a, v'>$ in V_i.[1] A read of address a generates a conflict if $block(a)$ is in W_k of some thread $k \neq i$. A write of address a generates a conflict if $block(a)$ is in R_k or W_k of some thread $k \neq i$. Transaction commit decrements $level_i$ and, if $level_i$ is now 0, updates M with the new mappings in V_i and makes R_i, W_i, and V_i null. A transaction abort discards R_i, W_i, and V_i, sets $level_i$ to 0, and reverts execution to just before the top-level transaction's begin.

3. Flat LogTM Background

The *Log-based Transactional Memory* system [21], referred to as *Flat LogTM* and depicted in Figure 1, serves as the framework for our nested transaction implementations.

Flat LogTM performs *version management* by storing new values (for commit) "in place" in cacheable, virtual memory and old values (for abort) in a per-thread log, also in cacheable, virtual memory. The log consists of a fixed-size header containing thread state (e.g., registers) and a variable sized body comprising undo records. Figure 2 shows an in-depth example of a transaction's execution. After step ❷, for example, the log header contains thread i's register state (including the program counter for step ❶) and the log body contains an undo record $<old\text{-}block\text{-}value$, $block(a)>$, where $old\text{-}block\text{-}value$ contains the value of $block(a)$ prior to the `transaction_begin()` in step ❶ (including the old value 2 for a) and $block(a)$ is the block's aligned virtual address. The processor maintains pointers to the log header and the log end. On the first write to a block (i.e., W bit not set—see below), Flat LogTM writes an undo record to the end of the log. On commit, Flat LogTM discards the log, resetting the log end pointer. On abort, Flat LogTM invokes a runtime software handler to unroll the log—processing the undo records in last-in-first-out order (i.e., starting with the log end pointer) and restoring thread state from the header. Flat LogTM implements closed transactional nesting by using a counter to flatten all nested transactions into the top-level transaction.

Like several other HTMs, Flat LogTM performs *conflict detection* using read (R) and write (W) bits in caches. Both commit and abort *flash clear* these R and W bits. Flat LogTM handles cache evictions of transactional data with an extended directory protocol (i.e., the *sticky-S* and *sticky-M* states [21]).

Flat LogTM implements Section 2's transactional memory model not as a forward-value map, but instead as a backward-value map. It replaces memory M with update-to-date memory M^+ and each "forward" value map V_i with a "backward" value map V_i^+ that has old mappings, e.g., $<a, v'>$. Because Flat LogTM updates memory in place, M^+ always equals $M \oplus Union_i\{V_i\}$, where $A \oplus B$ means that mappings $<a,v>$ in A are replaced with corresponding mappings $<a,v'>$ in B. Conflict detection ensures that the V_i are disjoint sets. Flat LogTM can always revert memory M^+ back to memory M, since $M = M^+ \oplus Union_i\{V_i^+\}$. For example, to abort thread i's transaction, Flat LogTM uses the backward map V_i^+ to undo only thread i's changes ($M^+\oplus= V_i^+$). We omit a proof here, since the equivalence of backward and forward logs is well understood in the database literature [20]. Finally, Flat LogTM's R and W bits directly maintain the model's read and write sets, R_i and W_i, for blocks in the cache, while the *sticky-S* and *sticky-M*
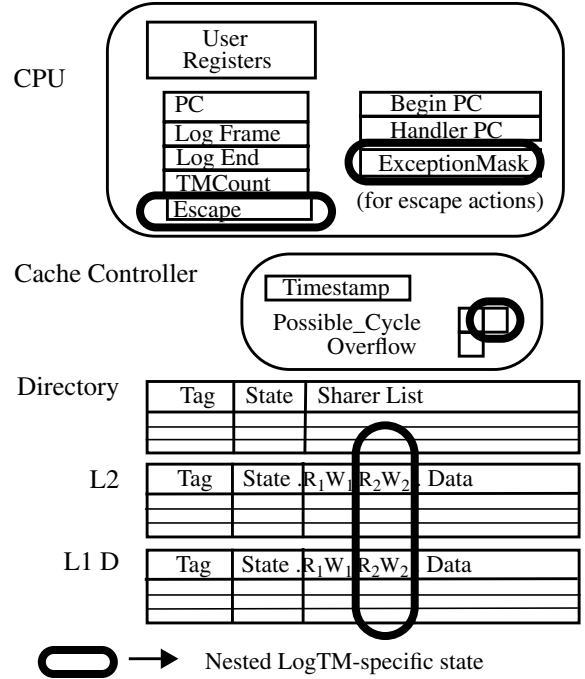
Figure 1: LogTM Node

Circles denote the architectural state additions to Flat LogTM [21] required to support Nested LogTM (with k=2).

states conservatively overestimate set membership for replaced blocks.

4. Closed Nesting With Partial Abort

This section presents requirements for closed nesting implemented with partial abort and an example.

4.1 Requirements

To extend the model with partial aborts, each thread maintains a separate read set, write set, and value map for each nesting level. For nesting level j, thread i maintains a read set $R_i(j)$, write set $W_i(j)$, and value map $V_i(j)$. Transaction begins, reads, and writes are similar to the base model in Section 2. Thread i begins a transaction by incrementing $level_i$. Reading address a adds $block(a)$ to $R_i(level_i)$. Writing a new value v to address a adds $block(a)$ to $W_i(level_i)$ and the mapping $<a, v>$ to V_i ($level_i$). Reading address a generates a conflict if $block(a)$ is in $W_k(j)$ of some thread $k \neq i$ and some level j. Writing address a generates a conflict if $block(a)$ is in $R_k(j)$ or $W_k(j)$ of some thread $k \neq i$ and some level j.

Transaction commit changes more significantly with the addition of partial aborts. A *top-level* commit ($level_i = 1$) remains the same: decrement $level_i$ to zero, update M with the new mappings in $V_i(1)$, and clear $R_i(1)$, $W_i(1)$, and $V_i(1)$. *A closed nested commit* (i.e., a commit when $level_i > 1$) instead promotes the committed transaction's state to the parent transaction's nesting level. Specifically, a closed nested commit at $level_i$ maintains isolation at $level_i$-1 by taking the union of the read and write sets: $R_i(level_i$-1) $\cup= R_i(level_i)$ and $W_i(level_i$-1) $\cup= W_i(level_i)$, where 'A $\cup=$ B' means that set A is assigned the set union of A and B. Similarly, the commit merges the value maps of $level_i$ and $level_i$-1: $V_i(level_i$-1) $\oplus= V_i(level_i)$, where 'A $\oplus=$ B' is the same as 'A $\cup=$ B' except that a mapping $<a,v>$ in B replaces a mapping $<a, v'>$ in A.

---

1. The model assumes the same granularity for read and write conflicts, but could be generalized for systems with different granularities, like TCC [7].
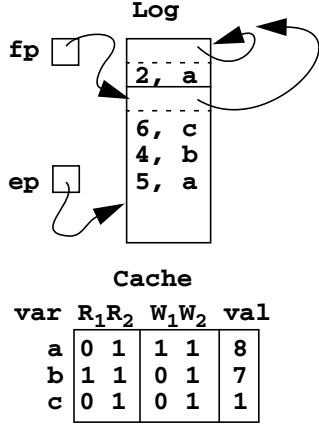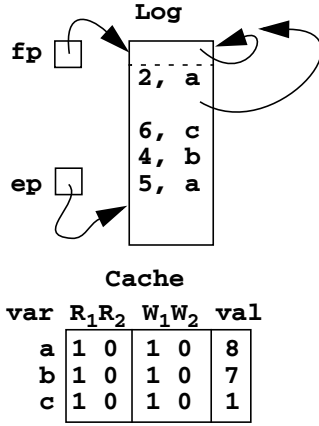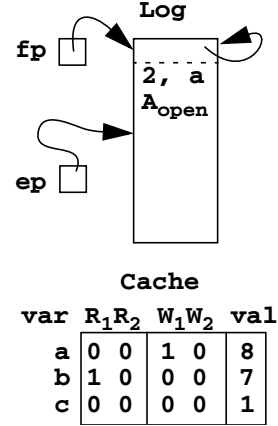
(a) After Step ❼

var	R₁	R₂	W₁	W₂	val
a	0	1	1	1	8
b	1	1	0	1	7
c	0	1	0	1	1

(b) After Step ❽ Closed Level 2

var	R₁	R₂	W₁	W₂	val
a	1	0	1	0	8
b	1	0	1	0	7
c	1	0	1	0	1

(c) After Step ❽ Open Level 2

var	R₁	R₂	W₁	W₂	val
a	0	0	1	0	8
b	1	0	0	0	7
c	0	0	0	0	1

Figure 2: Nested LogTM Operation

This figure shows three snapshots of the log and conflict detection bits. The lower half shows the cache state. The *var* columns correspond to the variable names from Figure 3, and the *val* columns show the current values in the processor local cache. The upper half shows a logical view of the log. Frames are separated by a solid black line. For instance, part (a) has two frames, while parts (b) and (c) each have only one. A dotted line separates the header from the undo records in each frame. Only the previous frame pointer part of the header records is shown. An undo record is abbreviated <old value>, <variable name>. The logs pictured also assume that $block(a) \neq block(b) \neq block(c)$.

Part (a) shows a snapshot of the log and the conflict detection bits after Figure 3's step ❼. There are several key points here. First, we see multiple logging: there are two logged "old values" of *a*, one in level 1's frame, and one in level 2's frame. Second, level 2's header includes a pointer to the beginning of level 1's frame for restoration on commit or abort. The header also contains other data, including a register checkpoint (not shown).

Finally, there is also "redundant" conflict detection: both levels have marked *b* as part of their read set. Clearly, a conflict occurs if another thread tries to write *b*. But a conflict also happens if another thread tries to read *b*. The important difference is how these bits affect abort. For instance, suppose another transaction makes a read request for *b*. "Redundant" conflict detection shows that it would be sufficient to abort and roll back only the level 2 transaction to successfully resolve this conflict. Conversely, a write conflict would abort both transaction levels 1 and 2.

Part (b) shows the state after level 2 completes execution and commits (step ❽ from Figure 3). The key take-away here is how the merge operations work. The log contains the same data before and after the merge; the only difference is that what used to be level 2's frame is now encompassed by level 1's frame. Note the two sources of waste: level 2's header and level 2's previous value of *a*. These are useless because a subsequent abort would have to go back through level 1, so level 1's header would be restored, and (since the log is traversed in LIFO order) level 1's old value of *a* would be the final value restored before restart.

The conflict detection bits also show the result of the merge operation. A level 1 bit is still set if it had been set before the level 2 commit, but it is additionally set if the corresponding level 2 bit was set. The level 2 bits have all been flash cleared. A write conflict to *b* would now be attributed to level 1 (the sole writer).

Part (c) shows the post-commit state if level 2 had been open instead of closed (Section 5). Unlike part (b), level 2's versions have been discarded and replaced with a compensation record (abbreviated "A_open" in the figure). The conflict bits show that the level 2 bits have been flash cleared, but the level 1 bits remain the same as before the transaction. At this point, another transaction could read *b* and read or write *c*. When traversing the log, an abort handler first executes the compensating action left by level 2, and only then restores the old value of *a*. It distinguishes the record types by examining their tags (not shown).

Finally, a commit clears $R_i(level_i)$, $W_i(level_i)$, and $V_i(level_i)$ and decrements $level_i$.

Like a top-level commit, a top-level abort with closed nesting remains unchanged: set $level_i$ to 0, clear $R_i(j)$, $W_i(j)$, and $V_i(j)$ for all levels *j*, and revert execution to just before the transaction begins. Aborting a closed nested transaction at level *j*, *j* > 1, results in a *partial abort* that leaves the enclosing transactions' states unchanged. Specifically, an abort at level *j* clears $R_i(n)$, $W_i(n)$, and $V_i(n)$ for $n \geq j$, sets $level_i$ to *j*-1, and reverts execution to just before the begin transaction at level *j*-1.

Figure 3 presents an extended example of closed nesting, including top-level and nested transactions, reads and writes, a partial abort, and eventual commit.

4.2 Closed Nesting in LogTM

We adapt Flat LogTM to support closed nesting with partial aborts by extending version management and conflict detection. Figure 1 depicts the architectural state added to a Nested LogTM node. Buffers used to hide the latency of log writes are not shown.

Version management. Closed nesting extends Flat LogTM's per-thread log (header plus a series of <*old-block-value, block-address*> undo records) into a stack of log frames, resembling a stack of activation records (similar to Harris et al. [11]). Each log frame consists of a fixed-size header and variable-size body containing undo records and garbage headers, described below. More specifically, Nested LogTM operates as follows. A transaction begin allocates a new log frame and initializes it with the thread's current registers (e.g., program counter), saves its

```
//thread i at level 0 (Non-transactional)
a = 2; b = 4; c = 6; // initialize
transaction_begin(); // top-level (level 1) ❶
    a = b + 1; // a gets 5 ❷
    transaction_begin(); // level 2 ❸
        c = b - 3; // c gets 1 ❹
        //PARTIAL ABORT: first write to b ❺
        b = a + 2; // b gets 7 ❻
        a = c + 7; // a gets 8 ❼
    transaction_commit(); // level 2 ❽
transaction_commit(); // level 1 ❾
```

Figure 3: Closed Nesting Example

Thread i begins execution at non-transactional level 0 and initializes variables a and b.

It then begins the top-level (level 1) transaction at step ❶. At step ❷, the read of variable b first adds *block(b)* to $R_i(1)$ for conflict detection. Next, the write of variable a with value 5 adds *block(a)* to $W_i(1)$ for conflict detection and $<a, 5>$ to $V_i(1)$ for version management.

Thread i begins a nested (level 2) transaction at step ❸. At step ❹, the read of b adds *block(b)* to $R_i(2)$, while the write of c with 1 adds *block(c)* to $W_i(2)$ and $<c, 1>$ to $V_i(2)$. At step ❺, the first try of step ❻'s write of variable b encounters a read conflict (i.e., *block(b)* is in $R_k(j)$ for some $k \neq i$). The system resolves this with a partial abort, which clears $R_i(2)$, $W_i(2)$ and $V_i(2)$ and restarts execution at step ❸.

Assume that the retry of step ❻ succeeds, adding *block(a)* to $R_i(2)$, *block(b)* to $W_i(2)$ and $<b, 7>$ to $V_i(2)$. Next, step ❼ adds *block(c)* to $R_i(2)$, *block(a)* to $W_i(2)$ and $<a, 8>$ to $V_i(2)$. Note that an address may exist in the structures of more than one transaction level. After step ❼, for example, *block(b)* is in both $R_i(1)$ and $R_i(2)$, *block(a)* is in both $W_i(1)$ and $W_i(2)$, and $<a, 5>$ is in $V_i(1)$ and $<a, 8>$ is in $V_i(2)$.

At step ❽, the nested (level 2) transaction commits by merging $R_i(2)$, $W_i(2)$ and $V_i(2)$ into $R_i(1)$, $W_i(1)$ and $V_i(1)$, respectively. Note that the merge causes the mapping $<a, 8>$ in $V_i(2)$ to replace the mapping $<a, 5>$ in $V_i(1)$. Finally, at step ❾, the top-level transaction commits, updating memory M with the mappings in $V_i(1)$ before clearing $R_i(1)$ and $W_i(1)$.

parent's transaction information (e.g., base of the parent's frame), and sets the log frame pointer to the new frame. Like Flat LogTM, Nested LogTM reuses W bits (described below) to detect the first write to each unique block, causing it to add an undo record to the end of the log's current frame body. Nested LogTM adds a two-bit tag to each log record, indicating whether it is an undo record, frame header, etc. Since block addresses are aligned to 64-byte blocks, the common undo record's tag value `00` is encoded for free in the address's least-significant address bits.

A closed nested commit merges the current log frame with its parent's frame. Specifically, Nested LogTM sets the log frame pointer back to the parent's frame (using the value saved at transaction begin in the committing transaction's frame). The committed transaction's frame header remains in the body of the parent as a *garbage header*. Garbage headers occupy space in the parent's frame, but have no semantic value.

An abort of the current transaction at level j traps to a (library) software handler that walks the body of j's log frame backwards to process undo records and skip garbage headers, finally restoring the register state saved in the header. A transaction abort through a level m ancestor of the current transaction level j has the software

handler undo j-m+1 log frames. This is easily implemented in a software handler, an advantage of LogTM's approach.

Conflict detection. Closed nesting changes each cache line's R and W bits into an array of bits, R[1..k] and W[1..k], where k is small (e.g., 4—Chung et al. [5] find two levels of explicit support sufficient). Reads and writes at transaction level j set R[j] and W[j] if $j \leq k$, and R[k] and W[k] if $j > k$ (i.e., flattened at level k). Flattening provides the correct behavior for deeper nests but removes the performance benefit of partial abort. An incoming read of *block(a)* generates a conflict at level j for the minimum j for which the corresponding W[j] is set (i.e., the outermost conflicting transaction). Similarly, an incoming write examines the corresponding R[j] and W[j] to find the minimum j with a conflict. When there is a possible cycle, indicating potential deadlock, Nested LogTM uses a bit per transaction level to inform the software handler how far to rollback.

A top-level transaction commit clears R[1] and W[1] with a flash clear. A nested transaction commit (level $j > 1$) merges R[j] and W[j] into R[j-1] and W[j-1], respectively, and flash clears R[j] and W[j]. One implementation of the merge is a *flash-OR* circuit that calculates R[j-1] $\vee$= R[j] and W[j-1] $\vee$= W[j] for each cache line in parallel (where A $\vee$= B assigns the logical-or of A and B to A). A flash-OR circuit adds two transistors and one "bit" line to the seven transistors and one "word" and three "bit" lines of an SRAM cell with flash clear. A transaction abort from level j back through level m first undoes log frames (see above) and then flash clears R[i] and W[i] for $m \leq i \leq j$.

Figure 2a and Figure 2b illustrate version management and conflict detection with closed nested transactions. Figure 2a shows Nested LogTM state when a nested transaction is ready to commit within a top-level transaction (after step ❼ in Figure 3). Note that the level-1 frame records the write from step ❷, while the level-2 frame records the writes from steps ❹, ❻ and ❼. Figure 2b shows Nested LogTM state after the inner transaction performs a closed commit, leaving only the top-level transaction active.

Correctness argument. Nested LogTM with closed nesting implements Section 4.1's closed nesting model in two steps. First, its stack of log frames provides backward maps $V_i^+(j)$ that complement the forward maps $V_i(j)$ in a manner similar to the way in which Flat LogTM's V_i^+'s complemented its V_i's. For example, a partial abort of thread i from level j back through level m must undo the changes made by thread i in transaction levels m through j: $M^+ \oplus = V_i^+(j) \oplus ... \oplus V_i^+(m+1)$. Second, cache bits R[1..k] and W[1..k] directly implement multi-level read- and write-sets, $R_i(j)$ and $W_i(j)$, for $j < k$, while, due to flattening, R[k] and W[k] represent the union of all $R_i(j)$ and $W_i(j)$, for $j \leq k$.

5. Open Nested Transactions

Nested LogTM also supports open nested transactions [24, 32], which provide greater concurrency and richer semantics. In this section, some "generational" terminology is helpful. Let a transaction T be invoked by thread i at level j. T's *parent* is thread i's level j-1 transaction, which began T. T's *ancestors* are its parent and its parent's ancestors. T's *siblings* are the level j transactions of thread i begun by T's parent (not including T).

5.1 Requirements

An open nested transaction releases isolation on commit and optionally registers *commit* and *compensating action* handlers [9, 14]. To ensure consistency, open nested transactions must raise the

level of abstraction of both isolation and rollback [25]. For example, Figure 4 illustrates an outer `insert_set()` transaction that calls `insert()` to add multiple entries to a B-tree data structure. Making each `insert()` an open nested transaction increases concurrency by releasing memory-level isolation on internal data structures. But to preserve consistency, the data structure must maintain isolation on these inserted entries—e.g., using a lock flag as in Figure 4—until the outer transaction commits. Note that the outer transaction must release isolation on the inserted entries by performing the specified commit actions, e.g., by calling `unlock(key)`. Similarly, the compensating actions must undo the forward action at this higher level of abstraction, e.g., by calling `delete(key)` for each B-tree entry.

An open nested transaction T_{open} commits by propagating its value map $V_i(j)$ to the nearest enclosing value map or memory, clearing $R_i(j)$, $W_i(j)$, and $V_i(j)$, (optionally) registering commit action C_{open} and/or compensating (or abort) action A_{open}, and decrementing $level_i$. For each mapping $<a, v>$ in $V_i(j)$, if there exists a mapping $<a, v'>$ in $V_i(k)$, $max\ k<j$, then the value propagates by $V_i(k) \oplus= <a, v>$. Otherwise the value propagates to memory, $M \oplus= V_i(j)$.

If thread i invokes only one open nested transaction, then when T_{open}'s top-level ancestor commits, it performs the commit action C_{open} as an open nested transaction (at a level no greater than T_{open} used before it committed). If one of T_{open}'s ancestors aborts, the system executes A_{open} as an open nested transaction. Moss argues that a compensating action should execute "in the state that held when its forward action committed" [23], that is A_{open} executes with the same $V_i(1) \oplus V_i(2) \oplus ... \oplus V_i(j-1)$ as when T_{open} committed. The top-level ancestor transaction releases isolation only after processing all commit or compensating actions. Note that programmers are responsible for ensuring that these actions do not generate additional conflicts beyond the original transaction; failure to do so may result in deadlock [31].

If thread i executes more than one open nested transaction before a top-level transaction commits, we must specify how handlers interact. For clarity, we first explain the semantics for the case where thread i executes *only* open nested transactions. As sibling open transactions $T_{open-1} ... T_{open-n}$ commit, they register commit and compensating actions $C_{open-1} ... C_{open-n}$ and $A_{open-1} ... A_{open-n}$ with their parent transaction $T_{open-parent}$. If $T_{open-parent}$ commits, the commit actions get executed in *first-in-first-out (FIFO)* order, $C_{open-1} ... C_{open-n}$, the compensating actions $A_{open-1} ... A_{open-n}$ are discarded, and $T_{open-parent}$ registers higher-level commit and compensating actions $C_{open-parent}$ and $A_{open-parent}$. Conversely, if $T_{open-parent}$ aborts, the compensating actions get executed in *last-in-first-out (LIFO)* order: $A_{open-n} ... A_{open-1}$. The action A_{open-k} executes with Moss's simple semantics (i.e., the same value maps $V_i(1) \oplus V_i(2) \oplus ... \oplus V_i(j-1)$ as when T_{open-k} committed) as long as none of the open nested transactions or compensating actions have written to the same address as an ancestor. Section 5.2 addresses the potential problems that arise in this case. More generally, when thread i executes both closed and open nested transactions, these actions are taken when the closest open ancestor transaction commits.

To illustrate compensating actions for multiple open transactions, suppose that $T_{open-parent}$ is the `insert_set()` routine that calls `insert()` to add multiple key/value pairs to a B-tree. If $T_{open-parent}$ aborts, the compensating actions A_{open-i} call `delete()` to remove the inserted keys. But once $T_{open-parent}$ commits, it registers a

```
insert(int key, int value) {
   open_begin();
      leaf = find_leaf(key);
      entry = insert_into_leaf(key, value);
      // lock entry to isolate node
      entry->lock = 1;
   open_commit(abort_action(delete(key)),
            commit_action(unlock(key)));
}
insert_set(set S) {
   open_begin();
      while ((key,value) = next(S))
         insert(key, value);
   open_commit(abort_action(delete_set(S)));
}
```

Figure 4: Open Nested B-Tree Example

higher-level `delete_set()` routine as $A_{open-parent}$, replacing the individual calls to `delete()`.

5.2 Interactions Between Undo and Compensating Actions

Open nested transactions can lead to strange behavior when a transaction T_{open} and one or more of its ancestors write to the same memory location. Figure 5 gives one such example. The problems arise because of non-obvious interactions between the memory-level undo operations of the parent and the semantic undo of the child. This problem is exacerbated because different TM implementations handle these interactions differently [18, 24, 26].

To avoid making programmers reason about subtle, non-portable implementation details, we advocate that most programmers observe the following restriction with open nested transactions.

Condition O1 (No Writes to Data Written by Ancestors): Neither an open transaction, T_{open}, nor its commit and compensating actions, C_{open} and A_{open}, writes any data written by T_{open}'s ancestors ($W_i(k)$ for all $k < j$), where T_{open} is executed by thread i at level j.

We see three advantages to (usually) obeying condition O1:

- Obeying condition O1 allows many effective uses of open nested transactions, For example, the B-tree example and our uses of open nested transactions in Section 8 obey condition O1.

- Obeying condition O1 frees programmers from reasoning about many subtle issues surrounding the leakage of uncommitted transactional state [26]. For example, if T_{open} and its ancestors write datum D, does isolation on D end when T_{open} commits, but its ancestors are still active?

- Obeying condition O1 avoids subtle interactions between the recovery of the parent's old values and the semantic undo of the child's actions.

5.3 Open Nesting in LogTM

Open nesting in Nested LogTM requires changes to version management and conflict detection.

Version management. When an open nested transaction T_{open} at level j commits, Nested LogTM discards T_{open}'s frame from the log (making T_{open}'s parent at level j-1 current) and optionally appends commit and compensating action records C_{open} and A_{open} to the newly exposed end of T_{open}'s parent's frame. The handler records contain a function address, a variable-length argument list, and its length.

```
counter = 0; // initializes ❶
transaction_begin(); // top-level ❷
    counter++; // counter gets 1 ❸
    open_begin(); // level 2 ❹
        counter++; // counter gets 2 ❺
    // commit with compensating action
    open_commit(
        abort_action(decr(counter))); ❻
...
// Abort and run compensating action ❼
// Expect counter to be restored to 0
...
transaction_commit(); // not executed
```

Figure 5: An Example Violating O1

Consider this example where thread i initializes a counter (to 0), begins a transaction, increments the counter (to 1), begins an open nested transaction, increments the counter again (to 2), commits the open nested transaction with a compensating action to decrement the counter (step ❻), and later aborts the top-level transaction (step ❼). If no other thread updated the counter, one might expect the counter to be restored to 0.

Nested LogTM (Section 5.3) would normally raise an exception at step ❺ to signal a dangerous programming practice. If the *condition-O1* exception is masked, the Nested LogTM implementation gets the expected value for this example. This occurs because it processes the log back to the commit point of the open nested transaction (step ❻) before running the compensating action, then processes the rest of the log (restoring the initial counter value).

Conversely, the McDonald, et al. TCC implementation [18] will set the counter to +1, not 0 [Kozyrakis and Olukotun, personal communication]. This occurs because the open transaction commit (step ❻) writes 2 into memory, which is later the input to the compensating action's decrement. Furthermore, because other, more complex examples cause strange behavior in all TM implementations, we believe that most programmers should obey condition O1.

On transaction abort, the software abort handler uses the log to restore values and perform compensating actions. Processing the log in LIFO order naturally produces the correct interleaving of restored values and compensating actions specified by Moss' semantics. The handler undoes transactional updates to the point of each open commit, before performing a compensating action.

Nested LogTM normally logs only the first store to a cache block at a given transaction level. After an open nested transaction commits, however, our semantics require that the log contain all undo records needed to roll back to the state of memory when that transaction committed. The run-time system ensures the presence of all necessary log records by immediately beginning a new *implicit* closed transaction at the same nesting level as the just committed open transaction (i.e., level j). The runtime system commits this implicit transaction immediately before the next begin or commit operation, promoting all conflict information to the j-1 nesting level.

Commit and compensating actions execute as open nested transactions (at the same (or lower) level as the corresponding T_{open} that committed). These actions may conflict, abort, and retry like any other open transaction.

Conflict detection. Beginning an open nested transaction T_{open} at level j and performing reads and writes within it behaves the same

as a closed nested transaction. When T_{open} commits, however, Nested LogTM simply flash clears all R[j] and W[j] bits (instead of employing the flash-OR used by a closed nested commit).

Nested LogTM also supports two maskable exceptions. An *open-level* exception gets raised if T_{open} begins at level $k+1$ which would exceed Nested LogTM's k levels of R/W bits. Masking this exception supports unbounded nesting by converting deeply nested open transactions to flattened closed transactions and ignoring the handlers registered at commit. Masking this exception is only appropriate for open nested transactions used to enhance concurrency, but not those used for two-way communication. Enabling an *open-level* exception allows software (error) handling when full open nested semantics are required, e.g., when they are used for communication between transactions.

LogTM raises a maskable *condition-O1* exception if T_{open} writes data written by an ancestor (violating condition O1). If this exception is not raised, the execution is known to obey the simpler semantics of condition O1. Programmers may choose richer semantics by handling or masking this exception. In that case, memory locations modified by both the parent and the open child remain isolated (i.e. part of W_{parent}).

Condition O1 can trigger false violations when stack locations used by a parent transaction are reused by an open child. These are not true violations, because the parent implicitly deallocated those stack addresses by changing its stack pointer. To prevent these unnecessary exceptions, Nested LogTM could provide a mechanism to identify the bottom of the stack when a transaction begins, and mask O1 violations for addresses on the stack page below this address. For the common case, where a transaction begins and ends at the same language scoping level (i.e., the same stack frame), the transaction begin would specify the address of the first stack variable written by the child transaction.

Figure 2 (parts a and c) illustrates version management and conflict detection with open nested transactions. Recall that Figure 2a shows Nested LogTM state when a nested transaction is ready to commit within a top-level transaction (after step ❼ in Figure 3). Note that the level 1 frame records the write from step ❷, while the level 2 frame records the writes from steps ❹, ❻ and ❼. Figure 2c shows Nested LogTM state after the inner open transaction commits, leaving only the top-level transaction active.

Correctness argument. Here we sketch why the changes to support open nesting in Nested LogTM are correct. An open transaction commit by thread i at level j performs in FIFO order any commit actions registered by committed descendent transactions, discards T_{open}'s frame (effectively promoting T_{open}'s changes to the nearest enclosing value map or memory) and clears $R_i(j)$ and $W_i(j)$ (to release isolation for data not accessed by a parent transaction). The abort handler processes the log in LIFO order, performing memory undos and compensation actions in the inverse order. Thus, even if a transaction violates condition O1, the compensating action sees an ancestral memory state equivalent to that which existed when the forward action committed.

6. Escape Actions

While open nested transactions promise greater concurrency, their semantics are ill-suited for invoking a conventional operating system, which will not tolerate stalls or aborts due to data conflicts.

To this end we propose *escape actions* which (a) are not transactional (no atomicity or isolation), (b) may not invoke a

Table 1: Escape Actions and Open Solaris System Calls

Category	#	Examples
Read-only calls	57	Getpid, times, stat, access, mincore, sync, pread, gettimeofday
Undoable calls with (at most) per-process side effects	40	Chdir, dup, umask, seteuid, nice, seek, mprotect
Undoable calls with global side effects (not currently handled)	27	Chmod, mkdir, link, mknod, stime
Other calls *not* handled by escape actions	89	Write, kill, fork, exec, umount

transaction, (c) may register commit and compensating actions, and (d) have no effect on enclosing transactions.

Escape actions are low-level escapes to software that is non-transactional (the rest of the way down), thereby allowing debuggers and I/O libraries to access and modify uncommitted transactional data without aborting transactions. They provide an interface to deal with non-transactional code, which for the foreseeable future includes operating systems and device drivers. Even transactional operating systems, though, may use escape actions to interact with non-transactional I/O devices. To support non-transactional system calls, escape actions implicitly begin (end) when entering (exiting) kernel mode, gracefully handling simple exceptions like software TLB fills. The escape_begin() and escape_end() calls provide explicit escape actions.

Similar to processor management instructions that are only available in assembly code, we envision escape actions being used to implement low-level functions (e.g., exception handlers, debuggers, and other run-time support), but are hidden from high-level language programmers. Thus, the complexity of writing escape actions is a one-time expense for library and OS developers but not application developers in general.

6.1 Case Study: Open Solaris Systems Calls

There are many reasons why TM applications may need to access conventional operating system services within a transaction. Using the memory allocator example from the introduction, transaction S may invoke sbrk() to grow the heap. But LogTM (or any other HTM) clearly cannot continue transactional operation within a non-transactional kernel like Open Solaris. Doing so would cede isolation control of kernel memory to user-level code.

Escape actions provide a bridge between transactional and non-transactional software. As summarized in Table 1, escape actions suffice to handle almost one hundred Open Solaris system calls for core OS services, such as processor and thread management, file I/O, and synchronization [30]. Escape actions are trivially correct for read-only system calls, such as getpid(), because concurrent callers cause no errors and thus need no isolation or compensation on aborts. Also in this category (but not shown in the table) are traps, such as TLB misses, that read kernel state without causing user-visible changes (even if they cause transparent changes).

Escape actions can also be made correct for system calls, such as sbrk(incr), that affect only threads in the current process. There are two key steps to invoking such system calls: (1) isolating other transactions from the effect of the call with a lock or sentinel, (2)

```
(void *)logtm_sbrk(int incr) {
    sbrk_sentinel = 0;
    escape_begin();
    tmp = sbrk(incr);
    if (tmp != NULL) {
        escape_end(
            abort_action(sbrk(-incr)));
    } else escape_end(NULL);
    return (tmp);
}
```

Figure 6: Wrapper for sbrk()

registering a compensating action to roll-back the effects of the call on transaction abort, and (3) releasing isolation when the enclosing transaction commits. A sentinel is a location in the process's virtual memory that acts as a transactional "lock" on kernel state. Writing to this sentinel location before invoking a system call in effect locks the sentinel and prevents other transactions from invoking the call until the outer transaction commits. Figure 6 illustrates a transactional wrapper around the sbrk() system call, which changes the process data segment size. The wrapper first writes a sentinel that prevents other threads from calling sbrk(incr) until the enclosing transaction commits or aborts. After invoking sbrk() successfully, the wrapper registers a compensating action that undoes the effect of the system call, in this case by resetting the data segment back to its original size with a call to sbrk(-incr). The wrapper begins and ends the escape action explicitly to ensure that the system call and compensating action registration appear atomic (since an enclosing transaction may abort immediately after the escape action ends). While sentinels are sufficient for simple system calls, more general locking mechanisms are necessary for more complex calls. Like open transactions, escape actions need commit actions to allow system calls to release isolation when the enclosing transaction commits.

Escape actions, however, cannot make all system calls safe. For example, calls that manipulate data in the file system are not undoable because the effects may have been observed by other processes. Similarly, calls such as kill() or unlink() cannot be undone, because the data associated with the process or file has been destroyed. Executing such calls safely inside a transaction may require a serialization-based technique such as *unrestricted transactions* [3].

Nevertheless, escape actions allow common exceptions, such as TLB miss handlers and protection fault handlers, and many useful system calls to execute correctly without disrupting running transactions. This greatly simplifies transactional programming by increasing the amount of code that may run within a transaction.

6.2 Requirements

Escape actions, like open transactions, have the potential for complex interactions between compensation and undo. In addition, escape actions' bypassing of transaction conflict detection and version management (weak atomicity) can lead to further programming challenges. As a result, we recommend that an escape action X executed by thread i at transaction level j obey the following conditions:

Condition X1 (No Writes to Data Written by Ancestors): X does not write any data written by its ancestors ($W_i(k)$ for all $k \leq j$).

Condition X2 (No Writes to Data Accessed by Others): X does not write any data read or written by active transactions of other threads ($R_m(k) \cup W_m(k)$ for $m \neq i$ and all k).

Condition X3 (No Reads to Data Written by Others): X does not read any data written by active transactions of other threads ($W_m(k)$ for $m \neq i$ and all k).

Conditions X1 and X2 restrict an escape action from modifying uncommitted updates by its ancestor transactions and other threads' transactions, respectively. Condition X3 restricts an escape action from reading other threads' uncommitted updates.

When conditions X1, X2, and X3 hold, escape action X operates as if it were non-transactional (level 0) except that: (1) X may not begin a transaction and (2) ending X resumes execution within thread i's level j transaction.

An escape action may register commit and compensating actions C_{escape} and A_{escape}. These handlers differ from their open transaction counterparts in that they execute as escape actions.

Escape actions that violate one or more of conditions X1, X2, and X3 may be important for debuggers and exception handlers—e.g., to allow updates to erroneous values before resuming a transaction. For this reason, the next section describes LogTM's escape action behavior including X1-X3 violations. However, the behavior of these cases in other HTMs may differ in subtle implementation-dependent ways.

When conditions X1, X2 and X3 hold, escape actions resemble Zilles and Bauch's *pause/unpause* [36]. Each provides an escape mechanism for non-isolated accesses to memory to support non-transactional operations, but escape actions do not maintain pause's strong atomicity for user-mode code. Zilles now concurs that weak atomicity [2] is necessary to prevent deadlock if paused regions access transactionally isolated data [Zilles, personal communication].

Escape actions also resemble the *external actions* of Harris's STM system for Java [9]. However, Harris's external actions take special and potentially costly steps to maintain Java memory safety, such as pre-registration of future external actions, whereas escape actions are a lightweight low-level mechanism useful for language design and system interactions.

6.3 Escape Actions in LogTM

Implementing escape actions in Nested LogTM is straightforward. An escape action requires a new `Escape` flag per thread (see Figure 1) that indicates whether a transaction is escaped. If `Escape` is not set, Nested LogTM behaves as before. An escape action begin sets `Escape`, and an escape action end clears it. Nesting of escape actions can be implemented in software, as there is no hardware action required when a nested escape action ends. If an ancestor transaction aborts while an escape action is running, the abort is delayed until the `Escape` flag is cleared.

Version management. When executing escape actions, thread i makes no changes to its log. Reads and writes always access coherent memory to return the value from the latest, possibly uncommitted, write (see below).

Conflict detection. When executing escape actions, thread i makes no changes to any of its R[k] and W[k] bits. It ignores conflicting accesses from other threads, except for the forced writeback described below.

Correctness argument. Here we sketch why changes from Nested LogTM to support escape actions are correct when conditions X1,

Table 2: System Model Parameters

	System Model Settings
Processors	32, 1 GHz, single-issue, in-order, non-memory IPC=1
L1 Cache	16 kB 4-way split, 1-cycle latency
L2 Cache	4 MB 4-way unified, 12-cycle latency
Memory	4 GB 80-cycle latency
Directory	Full-bit vector sharer list; migratory sharing optimization; Directory cache, 6-cycle latency
Interconnection Network	Hierarchical switch topology, 14-cycle link latency

X2 and X3 hold. When not executing escape actions, no behavior is affected. When executing an escape action X on thread i invoked from a transaction at level j, X reads values from M^+ (i.e., written by any transaction or from memory) and only writes data in memory M^+ (and not the $R_m()$'s and $W_m()$'s of any thread including i). Thus, no transactionally-modified data is accessed.

LogTM and Violations of X1, X2 and X3. In LogTM, reads in escape actions always return the value of the most recent, even uncommitted, write by any thread. Reads to data modified by a transaction in another thread (X3 violation) return the uncommitted data as an uncacheable block [15]. Similarly, writes to blocks read or modified by a transaction in another thread (X2 violation) update values immediately. Such writes invalidate remote caches—forcing modified blocks back to memory—and leave blocks in the sticky-S or sticky-M state [21]. Writes to memory locations modified by ancestors (X1 violation) update values, but do not affect ancestors' W bits or log.

7. Methods

This section describes target system assumptions and simulation techniques for evaluating Nested LogTM versus Flat LogTM.

Table 2 summarizes the parameters of our system model. The system has 32 processors, each with two levels of private cache. A MOESI directory protocol maintains coherence over a high-bandwidth switched interconnect. The single-issue in-order processor model assumes an aggressive, single-cycle non-memory IPC. The detailed memory system model includes most timing intricacies of the transactional memory extensions.

Our simulation framework uses *Virtutech Simics* [16] in conjunction with customized memory models built on *Wisconsin GEMS* [17, 33]. Simics, a full-system functional simulator, accurately models the SPARC architecture but does not support transactional memory. Support for LogTM was added using Simics "magic" instructions: special no-ops that Simics catches and passes to the memory model.

8. Experiments

This section presents four experiments that isolate the performance differences for the different nesting alternatives. Earlier studies have examined the benefits of transactions over locks [1, 7, 21, 28].

8.1 Exercising Nesting with Sorted List

First, we use two versions of a sorted list microbenchmark to (a) verify that Nested LogTM performs as expected and (b) illustrate
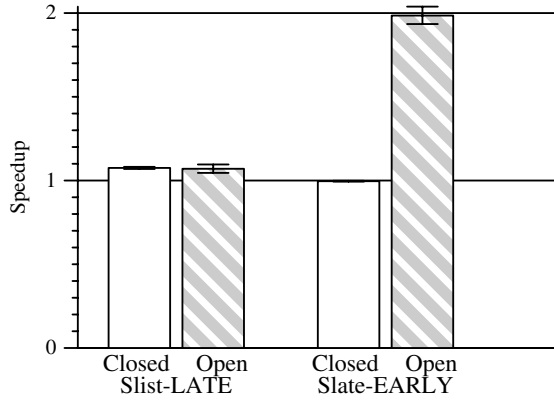
Figure 7: Slist Speedup of Closed Nesting with Partial Aborts or Open Nesting (over a Flat Implementation)

the conditions under which closed nesting with partial aborts (Closed) and open nesting (Open) can outperform flattening (Flat).

The sorted list microbenchmark searches a shared, sorted linked-list of complex elements. Each thread repeatedly begins a top-level transaction that examines list elements to find a matching field chosen at random. Both microbenchmark versions update a global, shared counter in a level-two nested transaction, but do so at different points in the execution. Slist-EARLY performs the update before the search, while Slist-LATE performs it afterwards. Since searches are read-only, all contention occurs on the counter update. Counter updates do not require a compensating action because the counter value must be unique but not continuous.

For Slist-LATE (the left side of Figure 7), Nested LogTM with both Closed and Open improves performance relative to Flat LogTM by about 10%. Performance improves because both avoid complete aborts when the nested transactions conflict. The two versions perform similarly, because the top-level transaction commits soon after the level-two transaction commits, limiting how long Closed and Flat extend isolation on the counter.

For Slist-EARLY (the right side of Figure 7), closed nesting with partial aborts offers no performance improvement, while open nesting provides dramatic improvement. Closed does not help since either (a) the abort occurs within the level-two transaction, and there is little difference between a partial abort and a complete abort since the nested transaction occurs early in the outer transaction, or (b) the abort occurs after the level-two transaction commits, forcing a complete abort since the outer transaction subsumes the inner. Open achieves much better performance since it releases isolation on the counter when the level-two transaction commits, greatly increasing concurrency.

These results illustrate that closed nesting with partial aborts can only improve performance when conflicts arise for nested transactions that occur late in the top-level transaction. Conversely, because open nesting releases isolation, it improves performance especially for nested transactions that occur early in the top-level transaction.

8.2 B-Tree Microbenchmark

The B-Tree microbenchmark represents a common class of concurrent data structures found in many applications. Results show that Closed provides little benefit over Flat, and that Open represents an attractive alternative to restructuring the workload to increase concurrency.
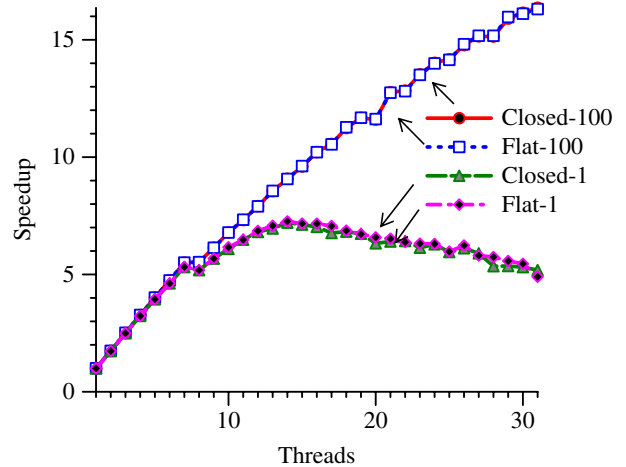
Figure 8: Scalability of B-Tree using Flat Closed Nesting and Closed Nesting with Partial Aborts and 1 (Flat-1 and Closed-1) or 100 (Flat-100 and Closed-100) free lists

In B-Tree, each thread makes repeated accesses to a shared tree, randomly performing a lookup (with 85% probability) or an insert (15%). The tree is a 9-ary B-Tree, initially 6 levels deep. Each high level operation (insert or lookup) is executed as a top-level transaction. Inserts that encounter full nodes split them on the way down the tree (preventing back propagation). Splits occur in a level-two closed nested transaction. New nodes are allocated from a shared free list in open or closed level-three nested transactions. We assume the presence of a garbage collector and therefore do not use a compensating action with open nested transactions.

Figure 8 illustrates the speedup of alternative versions of B-Tree relative to sequential performance. Speedups can exceed 16 on 31 threads. (We use 31, not 32, threads, because Solaris does not allows threads to be bound to all processors.) Such speedups are cost-effective in multiprocessors that cost less than 16 times a uniprocessor [35], which will certainly be the case for chip multiprocessors.

The bottom two lines of Figure 8, labeled Flat-1 and Closed-1, show poor speedup for Flat and Closed. Further investigation revealed that insert transactions frequently contended on the shared free list. Because the entire insert executes within a single parent transaction, both Flat and Closed subsume the free list into the parent transaction's write set, preventing access by subsequent inserts until the parent transaction commits.

Partitioning the free list into 100 separate lists (approximating thread-private allocators) eliminates this bottleneck, at the expense of restructuring the benchmark. The lines labelled Flat-100 and Closed-100 in Figure 8 show good speedup with the improved allocator, but little difference between Flat and Closed.

As discussed earlier, open nesting presents an appealing alternative for allocators. Open-1 uses an open nested transaction to access the simple (non-partitioned) free list. Figure 9 shows that Open-1 (simple allocator) performs much better than Closed-1 (simple allocator) and as well as Closed-100 (partitioned allocator).

These results demonstrate that a programmer (or library writer) can choose between an open nested transaction and the complexity of parallelizing her allocator. We see considerable merit in the former approach, especially for more complex allocators and other data structures.
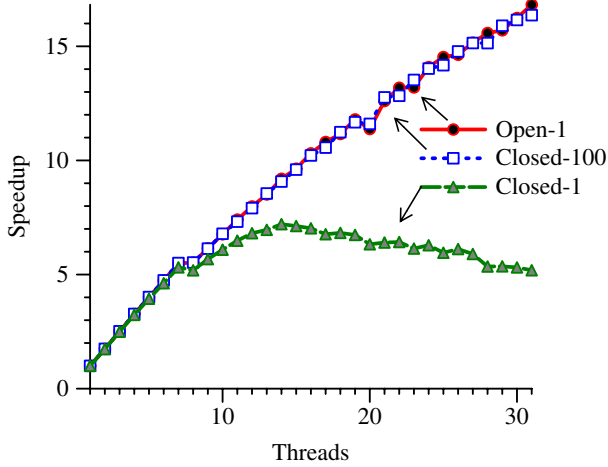
Figure 9: Scalability of B-Tree Using Multiple Allocators or Open Nesting.

8.3 Splash-2's Radiosity, Raytrace, Cholesky

The original LogTM paper [21] converted selected SPLASH-2 benchmarks [34] by replacing locks with transactions and leaving barriers and other synchronization mechanisms unchanged. That study found that Flat LogTM offered good performance that was comparable to or better than locks.

We modified the flat transaction versions of Radiosity, Raytrace, and Cholesky by adding closed nested transactions. Open nesting cannot be applied, because the data modifications cannot be undone once isolation is released. The other benchmarks did not lend themselves easily to nested transactions. Radiosity contains nested locking which we replaced with closed nested transactions. Raytrace and Cholesky have memory allocation routines that need to walk a list of elements. For these programs, we use nesting to split large transactions into smaller transactions that allow partial abort. The entire list access is placed within a transaction and, additionally, the code that modifies the list is put inside a nested transaction. We also optimized Raytrace to prevent interactions between transactions due to false sharing.

We ran these benchmarks with closed nesting with partial aborts and confirmed Chung et al.'s [5] finding that closed nesting does not help these programs (much). Our result is due, in part, to the rarity of aborts in SPLASH-2 on Flat and Nested LogTM.

8.4 Escaping to Open Solaris

An escape action allows a control transfer from a transactional nest to a non-transactional system (and back). Figure 7 (Table 1) illustrates that escape actions suffice to handle almost one hundred Open Solaris system calls [30]. This section illustrates Nested LogTM escaping to Solaris for TLB traps, loading the TLB in the non-transactional kernel, and resuming user-level activity. Without escape actions, LogTM must first abort the active transaction (to preserve user/kernel isolation), handle the TLB trap in non-transactional mode, then restart the transaction.

We exercise these two implementations of TLB traps with a single-threaded microbenchmark that walks a character array with a 256 B stride. Figure 10 displays the data read rate[1] for varying

---

1. The data rate is low because each reference fetches one byte, misses in the L2 cache, and each TLB miss causes a second-level TSB miss.
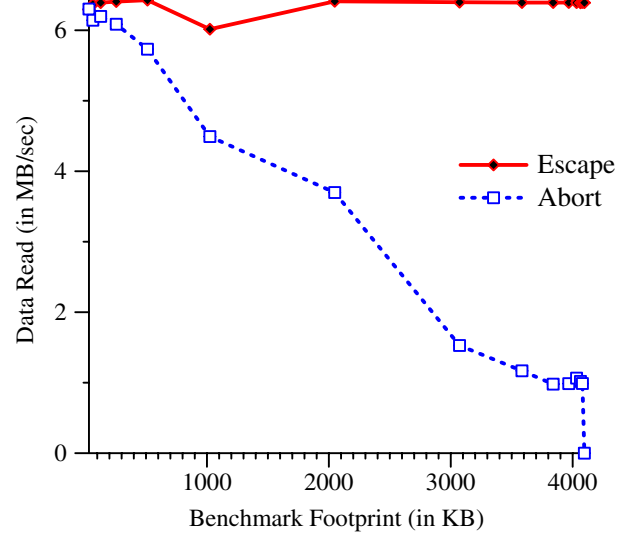

Figure 10: Strided Array Accesses Causing TLB Traps Handled with Escape Actions (Escape) or Aborts.

array sizes. Nested LogTM with escape actions (Escape) achieves roughly constant performance regardless of array size. The second line (Abort) shows that without escape actions, performance degrades rapidly as array size increases. Abort even fails to make forward progress for 4 MB arrays and larger, since with 8 KB pages and a 512-entry TLB, the TLB cannot map the entire array.

8.5 Results Discussion

Nested LogTM provides the greatest performance improvement over Flag LogTM when open nested transactions are used to release isolation early, thereby increasing concurrency. Escape actions improve performance by allowing non-transactional trap code to execute without aborting the current transaction.

We did not see much improvement from closed nested transactions, which only provides a performance benefit when transactions abort. Our SPLASH-2 results concur with Chung et al.'s [5] finding that closed nesting does not help these programs (much). In fact, not much benefit is possible with SPLASH-2 as Flat LogTM rarely aborts on these programs, in part because it *stalls* first [21].

More recently, the TCC group has adapted workloads where aborts are more common and partial aborts are valuable [6, 18]. For SPECjbb2000 on eight processors [6, Figure 3], for example, violations dominated execution time with flattening (60% of time) and closed nested reduced execution time substantially (40%). We do not (yet) know whether closed nesting will similarly help Nested LogTM on these workloads or whether stalling will mitigate the benefit of partial aborts.

9. Conclusions

Transactional memory (TM) systems can support *closed nested* transactions with complete aborts only (*flattening*) or selective *partial aborts* on nested transactions, *open nested* transactions, and/or newly-proposed *escape actions*. This paper expresses the behavior of these alternatives in a common model.

We implement the above nesting alternatives in the recently-proposed flat *Log-based Transactional Memory (LogTM)*. Nested LogTM supports (a) closed nesting with partial aborts by segmenting the log into a *stack of activation records* and modestly

replicating R/W bits, (b) open nesting by allowing a committing open transaction to release isolation and optionally save commit and compensating actions on the log, and (c) non-transactional escape actions, also with commit and compensating actions.

We evaluate with (mostly) TM microbenchmarks to demonstrate closed and open nesting performance differences (sorted list), concurrency exposed by open nesting (B-Tree), little performance improvement from closed nesting (SPLASH-2 subset), and correct operation for TLB traps via exposed actions (strided array).

Future work includes evaluating closed nesting with partial aborts and open nesting using benchmarks more representative of future TM workloads. We also plan to explore the richer semantics of open nested transactions and escape actions to provide more complete runtime and operating system support.

10. Acknowledgements

11. References

[1] C. S. Ananian, K. Asanovic, 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] C. Blundell, E. C. Lewis, and M. M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[3] C. Blundell, E. C. Lewis, and M. M. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, June 2006.

[4] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional Execution of Java Programs. In *SCOOL Workshop*, Oct. 2005.

[5] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2006.

[6] J. Chung, C. C. Minh, B. D. Carlstrom, and C. Kozyrakis. Parallelizing SPECjbb2000 with Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, June 2006.

[7] 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*, June 2004.

[8] T. Harris. Design Choices for Language-Based Transactions. Technical Report UCAM-CL-TR-572, University of Cambridge, Aug. 2003.

[9] T. Harris. Exceptions and side-effects in atomic blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[10] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2003.

[11] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, June 1991.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Twenty-Second ACM Symposium on Principles of Distributed Computing, Boston, Massachusetts*, July 2003.

[13] 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.

[14] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the sixteenth international conference on Very large databases*, pages 95–106, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[15] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

[16] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[17] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[18] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

[19] S. Microsystems. OpenSolaris: Mutex.c. http://cvs.opensolaris.org/-source/xref/on/usr/src/uts/common/os/mutex.c.

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *Readings in Database Systems*, pages 251–285. Morgan Kaufmann Publishers, 1998.

[21] 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.

[22] J. E. B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[23] J. E. B. Moss. Nesting Transactions: Why and What Do We Need? TRANSACT Keynote Address, June 2006.

[24] J. E. B. Moss. Open Nested Transactions: Semantics and Support. In *Workshop on Memory Performance Issues*, Feb. 2006.

[25] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstraction in recovery management. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 72–83, New York, NY, USA, 1986. ACM Press.

[26] J. E. B. Moss and A. L. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *SCOOL Workshop*, Oct. 2005.

[27] D. L. Parnas. On the criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, Dec 1972.

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

[29] N. Shavit and D. Touitou. Software Transactional Memory. In *Fourteenth ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada*, pages 204–213, Aug. 1995.

[30] I. Sun Microsystems. Solaris 10 Reference Manual Collection: man pages section 2: System Calls. http://docs.sun.com/app/docs/doc/816-5167.

[31] G. Weikum. A Theoretical Foundation of Multi-Level Concurrency Control. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 31–43, Mar. 1986.

[32] G. Weikum and H.-J. Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. Morgan Kaufmann, 1992.

[33] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[35] D. A. Wood and M. D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, pages 69–72, Feb. 1995.

[36] C. Zilles and L. Baugh. Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.