Architectural Semantics for Practical Transactional Memory

Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi,
Christos Kozyrakis and Kunle Olukotun

Computer Systems Laboratory
Stanford University
{austenmc, jwchung, bdc, caominh, hchafi, kozyraki, kunle}@stanford.edu

Abstract

Transactional Memory (TM) simplifies parallel programming by allowing for parallel execution of atomic tasks. Thus far, TM systems have focused on implementing transactional state buffering and conflict resolution. Missing is a robust hardware/software interface, not limited to simplistic instructions defining transaction boundaries. Without rich semantics, current TM systems cannot support basic features of modern programming languages and operating systems such as transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions.

This paper presents a comprehensive instruction set architecture (ISA) for TM systems. Our proposal introduces three key mechanisms: two-phase commit; support for software handlers on commit, violation, and abort; and full support for open- and closed-nested transactions with independent rollback. These mechanisms provide a flexible interface to implement programming language and operating system functionality. We also show that these mechanisms are practical to implement at the ISA and microarchitecture level for various TM systems. Using an execution-driven simulation, we demonstrate both the functionality (e.g., I/O and conditional scheduling within transactions) and performance potential (2.2× improvement for SPECjbb2000) of the proposed mechanisms. Overall, this paper establishes a rich and efficient interface to foster both hardware and software research on transactional memory.

1 Introduction

As chip-multiprocessors become ubiquitous, providing architectural support for practical parallel programming is now critical. Transactional Memory (TM) [17] simplifies concurrency management by supporting parallel tasks (transactions) that appear to execute atomically and in isolation. Using optimistic concurrency, TM allows programmers to achieve increased parallel performance with easy-to-identify, coarse-grain transactions. Furthermore, transactions address other challenges of lock-based parallel code such as deadlocks and robustness to failures.

Several proposed systems implement transactional memory in hardware (HTM) using different techniques for transactional state buffering and conflict detection [27, 12, 4, 28, 23]. At the instruction set level, HTM systems provide only a couple of instructions to define transaction boundaries and handle nested transactions through flattening. While such limited semantics have been sufficient to demonstrate HTM's performance potential using simple benchmarks, they fall short of supporting several key aspects of modern programming languages and operating systems such as transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions. Moreover, the current HTM semantics are insufficient to support recently proposed languages and runtime systems that build upon transactions to provide an easy-to-use concurrent programming model [13, 14, 8, 2, 7, 29, 20, 11, 1, 6]. For HTM systems to become useful to programmers and achieve widespread acceptance, it is critical to carefully design expressive and clean interfaces between transactional hardware and software before we delve further into HTM implementations.

This paper defines a *comprehensive instruction set architecture (ISA) for hardware transactional memory*. The architecture introduces three basic mechanisms: (1) *two-phase transaction commit*, (2) *support for software handlers on transaction commit, violation, and abort*, and (3) *closed- and open-nested transactions with independent rollback*. Two-phase commit enables user-initiated code to run after a transaction is validated but before it commits in order to finalize tasks or coordinate with other modules. Software handlers allow runtime systems to assume control of transactional events to control scheduling and insert compensating actions. Closed nesting is used to create composable programs for which a conflict in an inner module does not restrict the concurrency of an outer module. Open nesting allows the execution of system code with independent atomicity and isolation from the user code that triggered it. The proposed mechanisms require a small set of ISA resources, registers and instructions, as a significant portion of their functionality is implemented through software conventions. This is analogous to function call and interrupt handling support in modern architectures, which is limited to a few special instructions (e.g., jump and link or return from interrupt), but rely heavily on well-defined software conventions.

We demonstrate that the three proposed mechanisms are sufficient to support rich functionality in programming lan-

guages and operating systems including transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions within transactions. We also argue that their semantics provide a solid substrate to support future developments in TM software research. We describe practical implementations of the mechanisms that are compatible with proposed HTM architectures. Specifically, we present the modifications necessary to properly track transactional state and detect conflicts for multiple nested transactions. Using execution-driven simulation, we evaluate I/O and conditional synchronization within transactions. Moreover, we explore performance optimizations using nested transactions.

Overall, this paper is an effort to revisit concurrency support in modern instruction sets by carefully balancing software flexibility and hardware efficiency. Our specific contributions are:

- We propose the first comprehensive instruction set architecture for hardware transactional memory that introduces support for two-phase transaction commit; software handlers for commit, violation, and abort; and closed- and open-nested transactions with independent rollback.
- We demonstrate that the three proposed mechanisms provide sufficient support to implement functionality such as transparent library calls, conditional synchronization, system calls, I/O, and runtime exceptions within transactions. No further concurrency control mechanisms are necessary for user or system code.
- We implement and quantitatively evaluate the proposed ISA. We demonstrate that nested transactions lead to 2.2× performance improvement for `SPECjbb2000` over conventional HTM systems with flat transactions. We also demonstrate scalable performance for transactional I/O and conditional scheduling.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 motivates the need for rich architectural semantics in HTM systems. We introduce the instruction set semantics of the three mechanisms in Section 4 and describe how they support programming language and system code development in Section 5. In Section 6, we discuss the hardware implementation of the proposed mechanisms. Finally, Section 7 presents a quantitative evaluation, and Section 8 concludes the paper.

2 Background and Related Work

2.1 Transactional Memory Overview

With Transactional Memory (TM), programmers define atomic code sequences (transactions) that may include unstructured flow control and any number of memory references. A TM system executes transactions providing: (1) *atomicity*: either the whole transaction executes or none of it; (2) *isolation*: partial memory updates are not visible to other transactions; and (3) *consistency*: there appears to be a single transaction completion order across the whole system [17]. TM systems achieve high performance through speculation. A transaction runs without acquiring locks, optimistically assuming no other transaction operates concurrently on the same data. If this is true at the end of its execution, the transaction commits its writes to shared memory. If not, the transaction violates, its writes are rolled back, and it is re-executed.

Any TM system must implement the following mechanisms: (1) *isolation of stores* until the transaction commits; (2) *conflict detection* between concurrent transactions; (3) *atomic commit* of stores to shared memory; (4) *rollback* of stores when conflicts are detected. Conflict detection requires tracking the addresses read (*read-set*) and written (*write-set*) by each transaction. A conflict occurs when the write-set of one transaction intersects with the read-set of another concurrently executing transaction. These mechanisms can be implemented either with hardware-assisted (HTM) [27, 12, 4, 28, 23] or software-only (STM) [31, 15, 16, 13, 21, 30] techniques. This paper focuses on HTM systems because they support transactional mechanisms at minimal overheads and make the implementation details transparent to software.

2.2 Hardware Transactional Memory

Recent work has demonstrated that various HTM implementations allow for good performance with simple parallel code [27, 12, 23]. HTM systems implement speculative buffering and track read- and write-sets in caches. For conflict detection, they use the cache-coherence protocol. Since HTM systems are subject to physical resource limitations (cache capacity, paging, context switches), virtualization mechanisms have been proposed that allow transactional state to be stored in virtual memory [4, 28].

There are several differences among the HTM proposals. Speculative writes can be stored in a write-buffer until commit [27, 12] or stored in shared memory directly if an undo-log is maintained [23]. Conflicts can be detected eagerly as transactions access memory [4, 23] or lazily on transaction commit [12]. An HTM system may allow non-transactional code to see partial updates (weak atomicity) or provide full isolation of uncommitted updates (strong atomicity) [5]. Transactions can be used for localized non-blocking synchronization [4, 23] or in a continuous manner [12]. Finally, HTM systems differ in other aspects such the granularity of state tracking and the use of snooping or directory-based coherence [22, 23].

The semantics supported by current HTM systems are typically limited to instructions that define transaction boundaries. UTM [4] and LogTM [23] suggest the use of a software violation handler for contention management on conflicts. Our work presents a general mechanism for transactional handlers on violations, user-initiated aborts, or commits. We are the first to describe the hardware and software necessary to implement handlers and how they support a wide range of system functionality. Moss and Hosking have discussed nested transactions and potential implementations [24]. We define alternative semantics for open nesting, which are more appropriate for composable software.

3 The Need for Rich HTM Semantics

Current HTM systems provide instructions to define transaction boundaries which are sufficient to support programming constructs such as `atomic{}` and demonstrate the HTM performance potential with simple benchmarks. However, they fall short of supporting key aspects of modern programming environments [19]. Moreover, there is now a significant body of work on languages and runtime systems that builds upon transactions to provide an easy-to-use concurrent programming model [13, 14, 8, 2, 7, 29, 20, 11, 1, 6]. To achieve widespread acceptance, HTM systems must support a full programming environment and allow for innovation in transaction-based software. This section reviews the basic software requirements that motivate the semantics proposed in Section 4.

Composable Software (libraries): Modern programs use hierarchies of libraries, which have well-defined interfaces, but their implementation is hidden from users. Since libraries called within transactions may include atomic blocks, transactions will often be nested. Current HTM systems deal with nested transactions by subsuming (or flattening) all inner transactions within the outermost one [12, 4, 23]. Flattening can hurt performance significantly as a conflict in a small, inner transaction may cause the re-execution of a large, outer transaction. Such conflicts may even occur due to bookkeeping data maintained by the library, which are only tangentially related to the concurrency in the overall program. To avoid such bottlenecks without nesting, a programmer must be aware of the implementation details of the library code, which is completely impractical. Hence, HTM systems must support independent abort of nested transactions.

Contention and Error Management: Current HTM systems handle conflicts by aborting and re-executing the transaction. Recent proposals, however, require software control over conflicts to improve performance and eliminate starvation [11]. Language constructs such as `tryatomic` [2] and `ContextListener` [13] allow alternate execution paths on transaction aborts. With nested transactions, programmers may define separate conflict policies for each nesting level. Finally, it is necessary for both error handling (e.g., `try`/`catch`) and debugging to expose some information about the aborted transaction before its state is rolled back. Hence, HTM systems must intervene on all exceptional events and manage transactional state and program control flow.

Conditional Synchronization: Transactions must replace locks not only for atomic execution but also for conditional synchronization (e.g., `wait`/`notify`). Conditional synchronization is useful with producer/consumer constructs and efficient barriers. Recently proposed languages include a variety of constructs that build upon transactions to provide conditional synchronization without explicit `notify` statements (conditional `atomic` [15], `retry` and `orElse` [14], `yield` [29], `when` [8], `watch` and `retry` [6]). To support such constructs, software needs control over conflicts and commits, and HTMs must also facilitate communication between uncommitted transactions.

State	Type	Description
Basic State		
`xstatus`	Reg	Transaction info: ID, type (closed, open), status (active, validated, committed, or aborted), nesting level
`xtcbptr_base`	Reg	Base address of TCB stack
`xtcbptr_top`	Reg	Address of current TCB frame
Handler State		
`xchcode`	Reg	PC for commit handler code
`xvhcode`	Reg	PC for violation handler code
`xahcode`	Reg	PC for abort handler code
`xchptr_base`	TCB	Base of commit handler stack
`xchptr_top`	TCB	Top of commit handler stack
`xvhptr_base`	TCB	Base of violation handler stack
`xvhptr_top`	TCB	Top of violation handler stack
`xahptr_base`	TCB	Base of abort handler stack
`xahptr_top`	TCB	Top of abort handler stack
Violation & Abort State		
`xvPC`	Reg	Saved PC on violation or abort
`xvaddr`	Reg	Violation address (if available)
`xvcurrent`	Reg	Current violation mask: 1 bit per nesting level
`xvpending`	Reg	Pending violation mask: 1 bit per nesting level

Table 1. State needed for rich HTM semantics. State may be in a processor register or stored in a TCB field.

System Calls, I/O, and Runtime Exceptions: HTM systems prohibit system calls, I/O, and runtime exceptions within transactions [4, 23] or revert to sequential execution on such events [12]. Both approaches are unacceptable as real programs include system calls and cause exceptions, often hidden within libraries. To avoid long transactions through system code invoked explicitly (system calls) or implicitly (exceptions), system code should update shared-memory independently of the user transaction that triggered it. There should also be mechanisms to postpone system calls until the corresponding user code commits or compensate for the system call if the corresponding user code aborts. Furthermore, system programmers should be able to use the atomicity and isolation available with transactional memory to simplify system code development.

4 HTM Instruction Set Architecture

To provide robust support for user and system software, we introduce three key mechanisms to HTM architectures: *two-phase commit*, *transactional handlers*, and *closed- and open-nested transactions*. This section describes their ISA-level semantics and introduces the necessary state and instructions. We discuss their implementation in Section 6. The instructions should be used by language and system developers to implement high-level functionality that programmers access through language constructs and APIs. The instructions provide key primitives for flexible software development and do not dictate any end-to-end solutions.

Throughout this section, we refer to Tables 1 and 2 that summarize the state and instructions necessary for the three mechanisms. Some basic instructions are already available in some form in HTM systems (e.g., `xbegin`, `xabort`, `xregrestore`, and `xrwsetclear`), but we need to modify their semantics. The exact encoding or name for instructions and registers depends on the base ISA used and is not important.

Each transaction is associated with a *Transaction Control Block (TCB)* in the same manner as a function call is associated with an activation record. The TCB is a logical structure that stores basic transaction state: a status word,

Instruction	Description
Transaction Definition	
xbegin	Checkpoint registers & start (closed-nested) transaction
xbegin_open	Checkpoint registers & start open-nested transaction
xvalidate	Validate read-set for current transaction
xcommit	Atomically commit current transaction
State & Handler Management	
xrwsetclear	Discard current read-set and write-set; clear xvpending at current nesting level
xregrestore	Restore current register checkpoint
xabort	Abort current transaction; jump to xahcode; disable further violation reporting
xvret	Return from abort or violation handler; jump to xvPC; enable violation reporting
xenviolrep	Enable violation reporting
Optional (Performance Optimizations)	
imld	Load without adding to read-set
imst	Store to memory without adding to write-set
imstid	Store to memory without adding to write-set; no undo information maintained
release	Release an address from the current read-set

Table 2. Instructions needed for rich HTM semantics.

the register checkpoint at the beginning of the transaction (including the PC), the read-set and write-set addresses, and the write-buffer or undo-log. Conceptually, all TCB fields can be stored in cacheable, thread-private main memory. In practice, several TCB fields will be in caches (e.g., the read-set, write-set, and write-buffer) or in registers (e.g., the status word) for faster access. Figure 2 summarizes the final view of the TCB.

4.1 Two-phase Commit

Semantics: We replace the monolithic commit instruction in current HTM systems with a *two-phase commit* [10]. The xvalidate instruction verifies that atomicity was maintained (i.e., no conflicts) and sets the transaction status to *validated*. Its completion specifies that the transaction will not be rolled back due to a prior memory access. The xcommit instruction marks the transaction *committed*, which makes its writes visible to shared memory. Any code between the two instructions executes as part of the current transaction and has access to its speculative state. The code can access thread-private state safely, but accesses to shared data may cause conflicts and should be wrapped within open-nested transactions (see Section 4.5). The code can also lead to voluntary aborts instead of an xcommit.

Use: Two-phase commit allows the compiler or runtime to insert code between xvalidate and xcommit. This is useful for commit handlers that help finalize system calls and I/O. It also enables the transaction to coordinate with other code before it commits. For example, we can run a transaction in parallel with code that checks its correctness (e.g., for memory leaks, stack overflows, etc.) [26]. Alternatively, we can coordinate multiple transactions collaborating on the same task for group commit [20].

4.2 Commit Handlers

Semantics: Commit handlers allow software to register functions that run once a transaction is known to complete successfully. Commit handlers execute between xvalidate and xcommit and require no further hardware support. Everything else is flexible software conventions. It is desirable that transactions can register multiple handlers, each with an arbitrary number of arguments. Hence, we define a *commit handler stack* in thread-private memory. The base and top of the stack are tracked in the TCB fields xchptr_base and xchptr_top, respectively. To register a commit handler, the transaction pushes a pointer to the handler code and its arguments on the stack. An additional TCB field, (xchcode), points to the code that walks the stack and executes all handlers after xvalidate. To preserve the sequential semantics of the transaction code, commit handlers should run in the order they were registered. As transactions may include multiple function calls and returns, handler developers should rely only on heap allocated data.

Use: Commit handlers allow us to finalize tasks at transaction commit. System calls with permanent side-effects execute as commit handlers (e.g., write to file or sendmsgs).

4.3 Violation Handlers

Semantics: Violation handlers allow software to register functions to be automatically triggered when a conflict is detected in the current transaction. The mechanisms for invoking and returning from the violation handler resembles a user-level exception. On a conflict, the hardware interrupts the current transaction by saving its current PC in the xvPC register and the conflict address in the the xvaddr register (if available). Then, it automatically jumps to the code indicated by the xvhcode register. To avoid repeated jumps to xvhcode if additional conflicts are detected, we automatically disable violation reporting. Additional conflicts detected are noted by in the xvpending register. The violation handler returns by executing the xvret instruction, which enables violation reporting and jumps to the address in xvPC (which may have been altered by software). If xvpending is set, the violation handler is invoked again and xvpending is cleared.

A transaction can register multiple violation handlers with arbitrary arguments in the *violation handler stack* stored in thread-private memory. The stack base and top are tracked in the TCB fields xvhptr_base and xvhptr_top, but code located at xvhcode is responsible for running all registered handlers. Violation handlers should run in the reverse order from which they were registered to preserve correct undo semantics.

Like commit handlers, violation handlers start as part of the current transaction and have access to its speculative state. They can safely access thread-private state, but should uses open-nested transactions to access shared state[1]. By manipulating xvPC before returning, violation handlers can continue the current transaction (i.e., ignore violation), roll back and re-execute, or roll back and run other code. To roll back the transaction, the handler must flush the write-buffer or process the undo-log, discard the read-set and write-set using xrwsetclear, and restore the register checkpoint with xregrestore. xrwsetclear also clears the xvpending register to avoid spurious violations.

[1] Violation reporting should be re-enabled before the nested transaction.

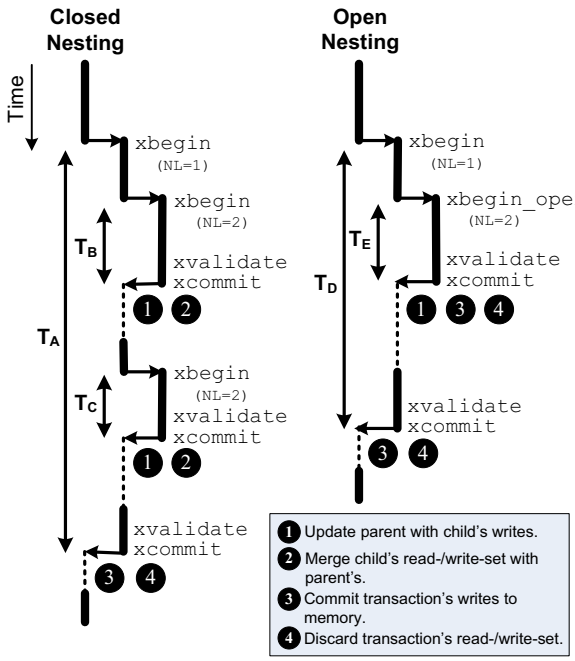

Figure 1. Timeline of three nested transactions: two closed-nested and one open-nested.

Use: Violation handlers allow for software contention management on a conflict [14, 11]. It also allows for compensation code for system calls that execute within the transaction if it rolls back (e.g., `read` or `lseek` to a file).

4.4 Abort Handlers

Semantics: Abort handlers are identical to violation handlers but they are triggered when a transaction uses an explicit `xabort` instruction. A separate register points to the code to invoke (`xahcode`). Abort handlers have a separate stack bounded by the `xahptr_base` and `xahptr_top` fields in the TCB. The uses of abort handlers are similar to those of violation handlers.

4.5 Nested Transactions

We define two types of nesting, explained in Figure 1.

Closed Nesting Semantics: A closed-nested transaction starts when an `xbegin` instruction executes within another transaction (T_B and T_C in Figure 1). The HTM system separately tracks the read-set, write-set, and speculative state of the child transaction from that of its parent. However, the child can access any state generated by an ancestor. If a child detects a conflict, we can independently roll back only the child, without affecting any ancestors. When a child commits using `xcommit`, hardware merges its speculative writes (❶) and read-/write-set (❷) with that of its parent, but no update escapes to shared memory. We make writes visible to shared memory only when the outermost transaction commits (❸, ❹).

Open Nesting Semantics: An open-nested transaction starts when an `xbegin_open` instruction executes within another transaction (see T_E in Figure 1). Open nesting differs from closed nesting only in commit semantics. On open-nested commit, we allow the child transaction to immediately update shared memory with its speculative writes (❸, ❹). The parent transaction updates the data in its read-set or write-set if they overlap with the write-set of the open-nested transaction. However, conflicts are not reported and no overlapping addresses are removed from the parent's read-set or write-set. If we want to undo the open nested transaction after it commits and its parent aborts, we need to register an abort and/or violation handler.

Our closed nesting semantics are identical to those presented by Moss and Hosking [24]. However, our open nesting semantics differ. Moss and Hosking propose that open-nested transactions remove from their ancestors' read-set or write-set any addresses they update while committing. Their motivation is to use open nesting as an *early release* mechanism that trims the read-/write-set for performance optimizations [7]. We find these semantics non-intuitive and dangerous: an open-nested transaction within library code that is a black box to the programmer can change the atomicity behavior of the user's code in an unanticipated manner.

Use: Closed-nested transactions allow for independent rollbacks and contention management at each nesting level, which typically leads to better performance. Open-nested transactions can both rollback and commit independently from their parents, which provides a powerful tool for system code development. We can use them within a transaction to perform system calls without creating frequent conflicts through system state (e.g., `time`). We can use them for calls that update system state before parents commit (e.g., `brk`). We also use them in handlers to access shared state independently. Note that within an open-nested transaction, we still provide atomicity and isolation, hence system code does not have to use locks for synchronization. In many cases, open-nested transactions must be combined with violation handlers to provide undo capabilities.

4.6 Nested Transactions and Handlers

Nested transactions can have separate handlers. To properly track all information, each transaction has its own TCB frame. We implement a stack of TCB frames in thread-private memory as shown in Figure 2, with a frame allocated before `xbegin` or `xbegin_open` and deallocated on `xcommit` or a rollback. The base and current top of the TCB stack are identified by registers `xtcbptr_base` and `xtcbptr_top`. TCB frames have fixed length as the the read-set, write-set, and speculative state of the transaction are physically tracked in caches. For each transaction, `xstatus` tracks the current nesting level. Overall, TCB management for transactions is similar to activation record management for function calls.

A single stack is necessary to store all registered handlers of a certain type. Each transaction has separate base and top pointers to identify its entries in the stack. At commit, a closed-nested transaction merges its commit, violation, and abort handlers with those of its parent by copying its top pointer (e.g. `xchptr_top`) into the parent's top pointer. The fixed length of TCB frames makes such an operation trivial. On an open-nested commit, we execute commit han-

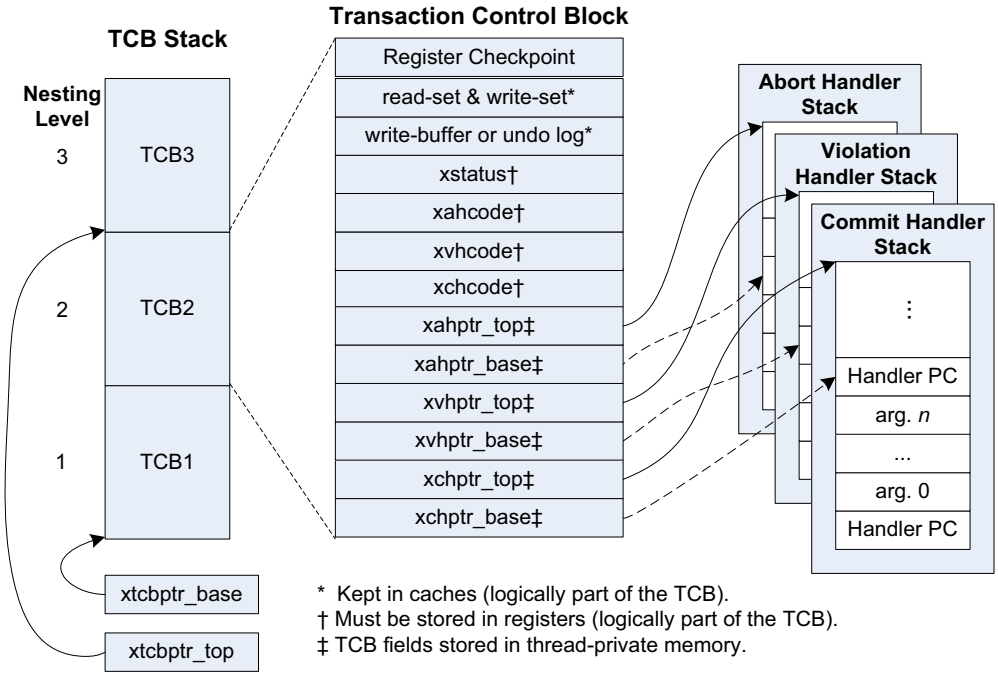

Figure 2. The Transaction Stack containing three Transaction Control Blocks (TCBs), one per active nested transaction. The second entry is shown in detail, complete with commit, violation, and abort handler stacks.

dlers immediately and discard violation and abort handlers. On nested rollback, we automatically discard its handlers without modifying the parent's pointers.

With nesting, conflicts can be detected for a transaction at any active nesting level and some conflicts may affect multiple levels at once. On a conflict, the hardware sets a bitmask, the `xvcurrent` register, to indicate which level(s) are affected. Similarly, the `xvpending` register uses a bitmask to remember any conflicts while conflict reporting is disabled. We always jump to the violation handler of the innermost transaction (top TCB), even if the conflict involves one of its parents. This is convenient as it allows software to run violation handlers at all levels as needed. It is also required for open-nested transactions that execute system code, as system handlers should be the first to be invoked. Note that it is up to software to clear the bitmask in the `xvcurrent` at conflicts are handled using the `xrwsetclear` instruction. Upon return from the violation handler, if any bits in `xvpending` or `xvcurrent` remain, `xvpending` is logically OR'd into `xvcurrent` and the innermost violation handler is invoked.

4.7 Discussion

Transactions often access thread-private data such as various fields in their TCB. To reduce the pressure on HTM mechanisms on such accesses, we provide immediate loads and store (see Table 2). An immediate load (`imld`) does not add the address to the current transaction read-set. An immediate store (`imst`) updates memory immediately without a commit and does not add the address to the current write-set. We also introduce an idempotent immediate store (`imstid`) that does not maintain undo information for the store in the write-buffer or the undo-log either. Immediate accesses can be interleaved with regular accesses tracked by HTM mechanisms. However, they should only be used when the compiler or system developer can prove it accesses thread-private or read-only data.

We also provide an early release instruction (`release`), which removes an address from the transaction's read-set. This instruction is attractive for performance tuning, but can complicate programming. We use it in low-level code for the conditional synchronization scheduler, but do not advocate its use in a high-level programming language [7]. Early release is difficult to implement consistently in some HTM systems: if the read-set is tracked at cache-line granularity, and an early release instruction provides a word address, it is not safe to release the entire cache line.

We do not support mechanisms to temporarily *pause* or *escape* a transaction and run non-transactional code. While, such mechanisms may seem attractive for invoking system calls, we find them redundant and dangerous. Open-nested transactions allow us to run (system) code independently of the atomicity of the currently running user transaction. Moreover, open-nesting provides independent atomicity and isolation for the system code as well. With pausing or escaping, a system programmer would have to use lock-based synchronization and deal with all its shortcomings (deadlocks, races, etc.). We believe the benefits of TM synchronization should be pervasive even in system code.

5 Language and System Code Uses

Section 4 generally described the uses of the proposed HTM mechanisms. This section provides specific examples that implement language or system code functionality to showcase the expressiveness of the mechanisms.

Transactional Programming Languages: We studied the proposed languages for programming with TM including Harris et al. [15, 13], Welc et al. [37], Transactional Haskell [14], X10 [8], Chapel [7], Fortress [2], AtomCaml [29], and Atomos [6]. The proposed ISA semantics are sufficient to implement these languages on HTM systems. Some languages formally support closed nesting [14, 6], and Atomos supports open nesting [6]. Additionally, open nesting can be used to implement the `AbortException` construct in Harris [13], conditional synchronization [14, 8, 29, 6], and transactional I/O [14, 6]. Two-phase commit and commit handers are used for I/O as well. Violation and abort handlers are used for error handling [14], the `tryatomic` construct in X10 [8], and in most implementations of conditional synchronization and I/O.

Conditional Synchronization: Figure 3 illustrates the concept of conditional synchronization for producer/consumer code in the Atomos programming language [6]. When a transaction wishes to wait for a value to change, it adds the corresponding addresses to a *watch-set* and yields the processor. If any values in the watch-set change, the thread is re-scheduled. This has the attractive property of avoiding the need for an explicit notify statement: the notifier does not need to know explicitly that someone is waiting for this value as the system automatically detects the change using conflict detection.

We implement this functionality using open nesting and violation handlers. An open-nested transaction communicates a waiting thread's watch-set to a scheduling thread that incorporates it as part of its read-set. Hence, the scheduler will then receive conflicts when any values in the watch-set change. Its violation handler will then add the proper thread to the run queue. To communicate with the scheduler, the waiting thread uses open nesting to write to a command queue and then violates the scheduler via the shared `schedComm` variable. Further details about conditional scheduling in Atomos are available [6]. With the proposed HTM mechanisms, we can implement similar runtime systems for other languages that support conditional synchronization within transactional code [14, 29, 8].

System Calls and I/O: To illustrate the implementation of system calls within transactions, we discuss I/O such as `read` and `write` calls without serialization. For input, we perform the system call immediately but register a violation handler that restores the file position or the data in case of a conflict. The system call itself executes within an open-nested transaction to avoid dependencies through system code. For output, we provide code that temporarily stores data in user buffers and registers a commit handler to perform the actual system call. This transactional scheme works with simple request/reply I/O, often the common case in applications [25, 9]. In a similar manner, we can implement other system calls within transactions that read or write system state. For example, a memory allocator can execute as an open-nested transaction including the `brk` call. For C and C++, a violation handler is registered to free the memory if the transaction aborts. For managed languages like Java and C#, no handler is needed, as garbage collection will eventually deallocate the memory.

6 Hardware Implementation

This section summarizes the hardware implementation of the mechanisms presented in Section 4. Our goal is to demonstrate that they have practical implementations compatible with current HTM proposals [12, 4, 28, 23].

6.1 Two-Phase Commit

The `xvalidate` instruction is a no-op for closed-nested transactions. For outer-most or open-nested transactions, the implementation must guarantee that the transaction cannot violate due to prior memory accesses once `xvalidate` completes. For HTM systems with eager conflict detection [4, 23], `xvalidate` must block until all previous loads and stores are known to be conflict-free by acquiring exclusive (for stores) or shared (for loads) access to the corresponding data. If timestamps are used for conflict resolution, the conflict algorithm must guarantee that a validated transaction is never violated by an active one even if it has a younger timestamp. For HTM systems that check conflicts when a transaction completes [12], `xvalidate` triggers conflict resolution, which typically involves acquiring ownership of cache lines in the write-set. For a system using tokens to serialize commits [22], `xvalidate` corresponds to acquiring the token.

The `xcommit` instruction atomically changes the transaction status to committed. Finishing the transaction also involves either resetting the undo-log or merging the contents of the write-buffer to shared memory. These steps can be executed within `xcommit` or in a lazy manner after `xcommit` returns. We discuss the implementation of `xcommit` for nested transactions in Section 6.3.

6.2 Commit, Violation, and Abort Handlers

The stack management for handlers is done in software without additional hardware support, other than some TCB fields stored in registers (see Table 1). Handlers allow for additional functionality in HTM systems at the cost of additional overhead for commit, violation, or abort events. Since transactions with a few hundred of instructions are common [9], our handler registration and management code is based on carefully tuned assembly. The code is also optimized for the common case of a commit without any registered commit handler or a violation that restarts the transaction without any registered violation handler. We present quantitative results in Section 7. Note that the same assembly code for handler management can be used by all languages or system code that builds upon the proposed semantics.

6.3 Nested Transactions

For nested transactions, the hardware must separately manage the speculative state, read-set, and write-set for

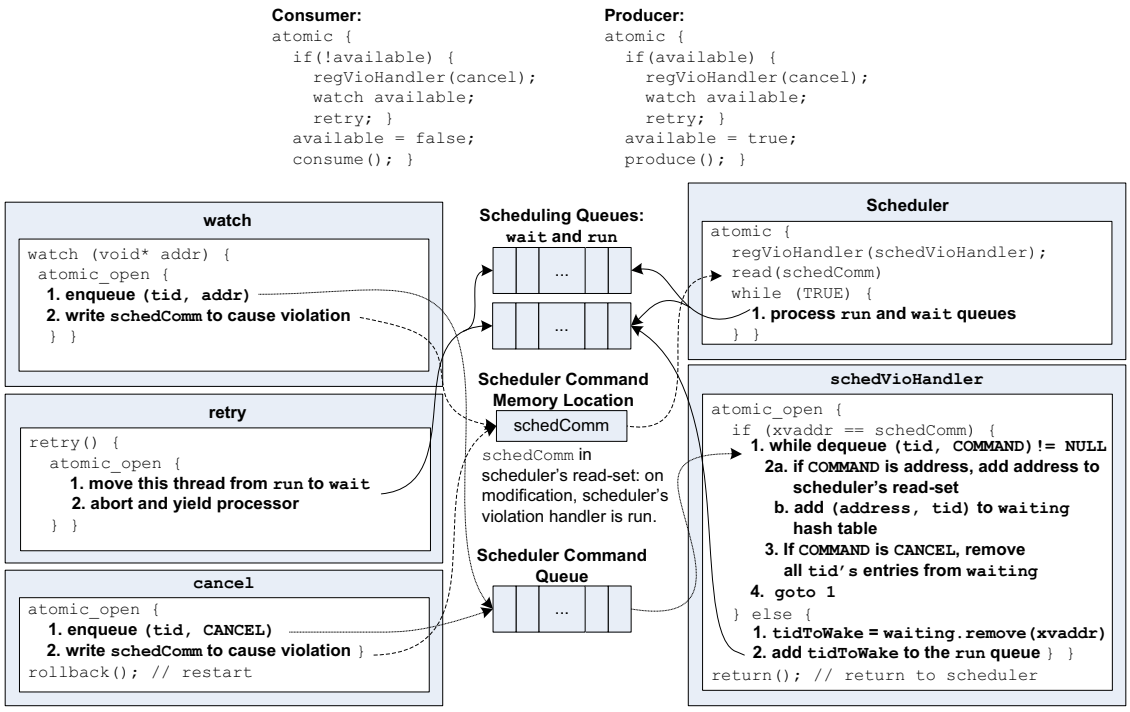

Figure 3. Conditional synchronization using open nesting and violation handlers for producer/consumer code in the Atomos programming language [6].

each active transaction. On a nested commit, we must merge its speculative state, read-set, and write-set with those of its parent. An additional complication is that multiple transactions in a nest may be writing the same cache line, which requires support for multiple active versions of the same data. While it is attractive to disallow this case, this will overly complicate the development of transparent libraries where arguments and return values can be frequently written by two transactions in a nest.

Hence, nesting requires significant changes to the caches used in HTM systems for read-/write-set tracking and speculative buffering. We propose two basic approaches: (1) the *multi-tracking scheme* that allows each cache line to track the read-set and write-set for multiple transactions, and (2) the *associativity scheme* that uses different lines in the same cache set to track multiple versions of the same data [35]. Even though there are numerous possible combinations of these two schemes with all other HTM design options (see Section 2.2), we will briefly describe two practical design points that illustrate their implementation details.

6.3.1 Nesting Support with Multi-tracking Lines

We consider the multi-tracking scheme for HTM designs using undo-logs [4, 23]. Each cache line tracks read-/write-set membership for multiple transactions in the nest, but the multiple versions of the data are buffered in the undo-log. The log is a stack structure in thread-private memory that holds the old version of data modified by an active transaction. When a nested transaction writes a cache line for the first time, we push the previous value in the undo-log. Hence, the undo-log may contain up to one entry per cache

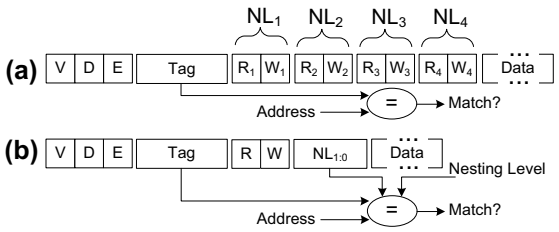

Figure 4. Cache line structure for (a) multi-tracking, and (b) associativity schemes with four levels of nesting.

line per nested transaction. Each transaction tracks the base point of its entries in the undo-log using a separate register or TCB field. On a closed-nested commit, the log entries are automatically appended to those of its parent without any action. On a nested conflict, we can roll back by processing the undo-log entries for this nesting level in FILO order. The only complication is that if an open-nested commit overwrites data also written by its parent, we must update the log entry of the parent to avoid restoring an incorrect value if the parent is later rolled back. This requires an expensive search through the undo-log.

Figure 4-(a) shows the cache line organization for multi-tracking. A line has multiple copies of the R_i and W_i bits that indicate membership in the read-set and write-set for the transaction at nesting level i (NL_i). If word-level tracking is implemented, we need per-word R and W bits. On a memory access, the hardware uses the nesting level counter in `xstatus` to determine which bits to set. On an cache lookup for conflict detection, all R and W bits are

checked in parallel. Hence, we can detect conflicts for all active transactions in parallel and properly set the bitmasks in `xvpending` and `xvcurrent`. On a rollback at NL_i, we gang invalidate (flash clear) all R_i and W_i bits. On a closed-nested commit at NL_i, we must merge (logical OR) all R_i bits into R_{i-1} and all W_i bits into W_{i-1}. Such merging is difficult to implement as a fast gang operation. To avoid latency proportional to the read-set and write-set size of the nested transaction, we can merge *lazily* while continuing the execution at NL_{i-1}: on a load or store access to a cache line, we perform the merging if needed with a read-modify-write as we lookup the cache line and update its LRU bits. During the lazy merge, the conflict resolution logic should consider R and W bits at both levels as one. The merging must complete before a new transaction at the level NL_i level is started. On an open-nested commit at NL_i, we simply gang invalidate all R_i and W_i bits.

6.3.2 Nesting Support with Associative Caches

We consider the associativity scheme for HTM designs using a write-buffer [27, 12]. As always, the cache tracks read-/write-set membership, but also buffers multiple version of old data for nested transactions. Figure 4-(b) shows the new cache line organization. Each line has a single set of R and W bits, but also includes a nesting level field (NL) to specify which transaction it holds data for. We reserve $NL = 0$ for cached data that do not yet belong to any read- or write-set. If the same data is accessed by multiple nested transactions, we use different lines in the same cache set. The most recent version is in the line with the highest NL field. Hence, cache lookups can return multiple hits and additional comparisons are necessary to identify the proper data to return, so cache access latency may increase in highly associative caches. On an access by a transaction at NL_i, we first locate the most recent version. If the cache line has $NL = 0$ (potentially after a cache refill), we change $NL = i$ and set the R or W bits. If there is another speculative version at level $i - 1$ or below, we first allocate a new line in the same set that gets a copy of the latest data and uses $NL = i$. On an external lookup for conflict detection, we check all lines in the set with $NL \neq 0$ to detect, in parallel, conflicts at all nesting levels.

On a rollback at NL_i, we must invalidate all cache entries with $NL = i$. This can be implemented as a gang invalidate operation if the NL field uses CAM bits. Alternatively, the invalidation can occur *lazily* as we access cache lines and before we start a new transaction at NL_i. On a closed nested commit at NL_i, we must change all lines with $NL = i$ to $NL = i - 1$. If an $NL = i - 1$ entry already exists, we merge its read-set and write-set information into the older entry and then discard it. Again, the best way to implement this is lazily. While lazily merging, the conflict detection logic must consider cache lines with $NL = i$ and $NL = i - 1$ to belong to the same transaction. An open nested commit is similar, but now we change entries from $NL = i$ to $NL = 0$. If there are more versions of the same data for other active transactions, we also update their data with that of the $NL = i$ entry without changing their R or W bits.

6.3.3 Discussion

Each nesting scheme has different advantages. The multi-tracking scheme does not complicate cache lookups and avoids replication for lines with multiple readers. The associativity scheme scales to a number of nesting levels equal to the total associativity of private caches without significant per-line overhead. A hybrid scheme with multi-tracking in the L1 cache and associativity in the L2 cache can provide the best of both approaches.

Any practical HTM implementation can only support a limited number of nesting levels. Hence, nesting levels must be virtualized just like any other buffering resource in HTM systems. Virtualization schemes like Rajwar et al. [28] can be extended to support unlimited nesting levels by adding a nesting level field in each entry in the overflow tables in virtual memory. Early studies have shown that the common case is 2 to 3 levels of nesting [9], which is easy to support in hardware. The hardware support for nesting can also be used to overlap the execution of independent transactions (double buffering), which can be useful with hiding any commit or abort overheads [22].

Moss and Hosking discuss a nesting implementation similar to our associativity scheme [24]. Apart from the different semantics for open nesting (see Section 4.5), their approach is overly complex. In addition to the NL field, each line includes a variable length stack with pointers to all child transactions with new versions of the data. To maintain the stacks, we need to push or pop values from multiple cache lines on stores, commits, and aborts. The two schemes we propose are simpler and introduce lower area and timing overheads.

The proposed nesting schemes do not support nested parallelism: the ability to execute a single transaction atomically over multiple processors [20, 2]. Nested parallelism requires that we can merge read-sets and write-sets across processors as parallel tasks complete within a transaction. Such functionality can be implemented using a shared L2 or L3 cache between collaborating processors that serves as a directory for the latest version of any data. Note, however, that the semantics of nesting defined in Section 4.5 are correct even for nested parallelism.

7 Evaluation

We implemented the proposed mechanisms as an extension to the PowerPC ISA using an execution-driven simulator for HTM chip-multiprocessors. Our goal is to examine optimization opportunities and use the new mechanisms to implement the programming constructs and runtime code functionality discussed in Section 3. An extensive comparison of alternative implementations for the proposed semantics is beyond the scope of this paper.

We model a chip-multiprocessor with up to 16 cores, private L1 caches (32KB, 1-cycle access), and private L2 caches (512KB, 12-cycle access). The processors communicate over a 16-byte, split-transaction bus. All non-memory instructions in our simulator have CPI of one, but we model all details in the memory hierarchy for loads and stores, including inter-processor communication and contention. We use an HTM system with a write-buffer for

speculative writes, lazy conflict detection, and continuous transactional execution [22]. We support three levels of nesting using the associativity scheme with lazy merging. None of the evaluated programs uses more than $NL = 2$.

We model all overheads associated with two-phase commit and the software for TCB and handler management. We have carefully optimized the assembly code for common events to avoid large overheads for small transactions. Starting a transaction requires 6 instructions for TCB allocation. A commit without any handlers requires 10 instructions, while a rollback without handlers requires 6 instructions. Registering a handler without arguments takes 9 instructions. Note that some of these instructions access thread-private data in memory and may lead to cache misses. Yet, such misses are rare as private stacks cache well. Overall, the new HTM functionality does not introduce significant overheads that may force programmers to reconsider the size of their transactions.

7.1 Performance Optimizations with Nesting

We used nested transactions to reduce the overhead or frequency of conflicts in scientific and enterprise applications. We used `swim` and `tomcatv` from the SPECcpu2000 suite [33]; `barnes`, `fmm`, `mp3d`, and water-nsquared (called simply `water`) from the SPLASH and SPLASH-2 suites [32, 36]; and a C version of `moldyn` from the Java Grande suite [18]. For these applications, we used transactions to speculatively parallelize loops. We also used a modified version of `SPECjbb2000` [34] running on the Jikes RVM [3].

Figure 5 plots the performance improvement achieved with the proposed nesting implementation over the conventional HTM approach that simply flattens nested transactions. The results were produced running 8 processors. The number above each bar reports the overall speedup achieved with nesting over sequential execution on one processor (maximum speedup is 8). Overall, no application is affected negatively by the overhead of TCB and handler management for nested transactions. Most outer transactions are long and can amortize the short overheads of the new functionality. Most inner transactions are short, hence lazy merging at commit does not become a bottleneck. On the other hand, several applications benefit significantly from the reduced cost of conflicts compared to flattening. For the scientific benchmarks, we applied closed nesting mainly to update reduction variables within larger transactions. This allows us to avoid several outer transaction rollbacks, particularly when the inner transaction is near the end of the outer one. The improvements are dramatic for `mp3d` (4.93×) where we also used nesting to eliminate expensive violations due to particle updates on collisions.

As a three-tier enterprise benchmark, `SPECjbb2000` is far more interesting from the point of view of concurrency. We parallelized `SPECjbb2000` within a single warehouse, where customer tasks such as placing new orders, making payments, and checking status manipulate shared data-structures (B-trees) that maintain customer, order, and stock information. Conceptually, there is significant concurrency

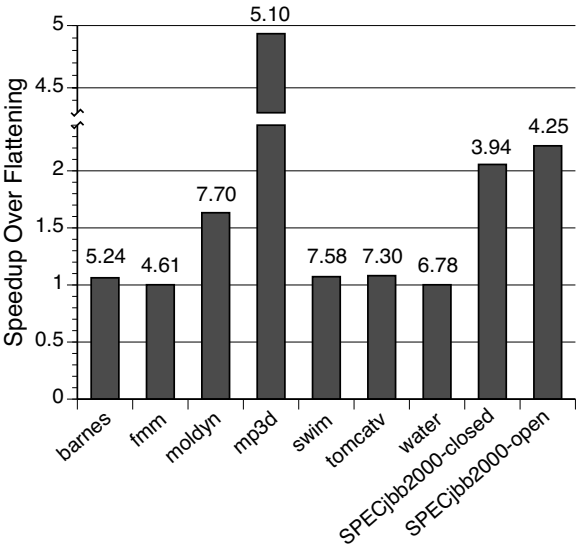

Figure 5. Performance improvement with full nesting support over flattening for 8 processors. Values shown above each bar are speedups of nested versions over sequential execution with one processor.

within a single warehouse, as different customers operate mostly on different objects. Nevertheless, conflicts are possible and are difficult to statically predict. We defined one outer-most transaction for each `SPECjbb2000` operation (order, payment, etc.). Even though we achieved a speedup of 1.92 using this flat-transaction approach, rollbacks significantly degrade performance. With nesting support, we developed two additional versions of the code. The first version, `SPECjbb2000-closed`, uses closed-nested transactions to surround searches and updates to B-trees. Performance is improved by 2.05× (total speedup of 3.94) as the violations occur frequently within the small inner transactions and do not cause the outer-most operation to roll back. The second version, `SPECjbb2000-open`, uses an open-nested transaction to generate a unique global order ID for new order operations. Without open nesting, all new order tasks executing in parallel will experience conflicts on the global order counter. With open nesting, we observe a performance improvement of 2.22× (total speedup of 4.25) as new orders can commit the counter value independently before they complete the rest of their work. Hence, conflicts are less frequent and less expensive. Note that no compensation code is needed for the open-nested transaction as the order IDs must be unique, but not necessarily sequential. We could use both open and closed nesting to obtain the advantages of both approaches, but we did not evaluate this.

7.2 I/O within Transactions

We designed a C microbenchmark where each thread repeatedly performs a small computation within a transaction and outputs a message into a log. We designed a transactional library function that buffers output in a private buffer and registers a commit handler before returning control to the application (see Section 5). If the transaction violates, the local buffer is automatically discarded because it is part

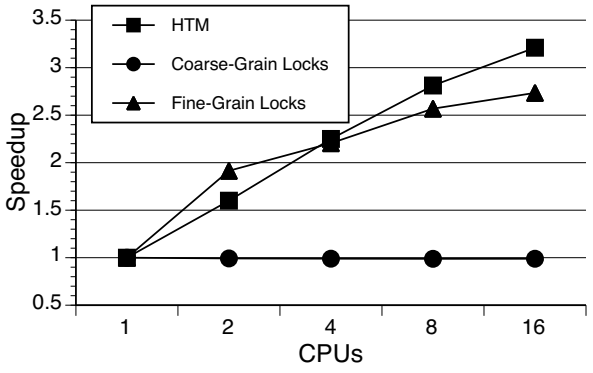

Figure 6. Speedup for the I/O microbenchmark with HTM, fine-grain locks, and coarse-grain locks.

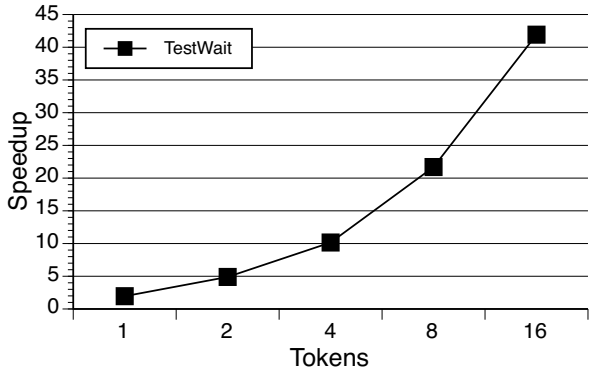

Figure 7. Speedup of token exchanges for `TestWait` with 32 processors, scaling the number of tokens from 1 to 16.

of the write-set. If the transaction successfully validates, the commit handler copies the local buffer to a shared buffer in the operating system.

We evaluated three versions of this I/O microbenchmark: the transactional one described above and two using conventional lock-based synchronization. The coarse-grain version acquires a lock at the boundaries of the library function call; so it formats and copies the log message into the shared buffer while holding the lock. The fine-grain version holds the lock only while copying to the shared buffer. Figure 6 compares the performance of the three versions as we scale from 1 to 16 processors. Since there is little work per repetition except printing to the log, the coarse-grain version completely serializes and shows no scalability. The fine-grain version allows for concurrency in message formatting. Still, its scalability is limited by serialization on buffer updates and the overhead of acquiring and releasing the lock. Initially, the performance of the transactional version suffers because it performs two copies: one to a local buffer then one to the shared buffer. However, because the overhead is fixed and does not scale with the number of threads, as the lock overhead does, the transactional version continues to scale. With more processors, we would notice the HTM version flattening as well due to conflicts when updating the shared buffer.

Despite its simplicity, the I/O benchmark shows that the proposed semantics allow for I/O calls within transactions. Moreover, despite the overhead of extra copying, transactional I/O in a parallel system scales well.

7.3 Conditional Synchronization within Transactions

To measure the effectiveness of the conditional synchronization scheme in Figure 3, we use the Atomos `TestWait` microbenchmark, which mimics an experience from Harris and Frasier [15], that stresses thread scheduling in a producer-consumer scenario. `TestWait` creates 32 threads, arranged in a ring of producers and consumers with a shared buffer between each pair. The benchmark scales the number of tokens in the ring and begins passing a fixed number of tokens from one thread to the next. Every thread uses a transaction to perform one token operation (dequeue from receiving buffer and copy to the outgoing buffer). An efficient system should scale with the number of tokens.

Figure 7 shows that passing the tokens scales superlinearly with the number of tokens in the ring; this is expected since with more tokens, a consumer may find an available token immediately without having to synchronize with the producer. See further discussion in the evaluation of the Atomos programming language [6]. Hence, conditional synchronization within transactions using open nesting and violation handlers is quite effective for the studied system. Both the open-nested transactions and the violation handlers are quite small and introduce negligible overhead.

8 Conclusions

For HTM systems to become useful to programmers and achieve widespread acceptance, we need rich semantics at the instruction set level that support modern programming languages and system code. This paper proposed the first comprehensive ISA for HTM that includes three key mechanisms (two-phase commit, transactional handlers, and open- and closed-nested transactions). We described the hardware and software conventions necessary to implement these mechanisms in various HTM systems. Moreover, we have quantitatively evaluated the proposed mechanisms by showing their use for both system code functionality (scalable I/O and conditional synchronization) and performance optimizations through nested transactions.

Armed with these implementation-independent semantics at the instruction set level, the TM community can effectively develop and evaluate hardware proposals that are practical for programmers and support a wide range of applications. Similarly, software researchers can design efficient languages and runtime systems for HTM on top of a rich interface between their programs and the underlying hardware.

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant number NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Additional support was also available through NSF grant 0444470 and through the MARCO Focus Center for Circuit & System Solutions (C2S2), under contract 2003-CT-888. Brian D. Carlstrom is supported by an Intel Foundation Ph.D. Fellowship.

References

[1] A.-R. Adl-Tabatabai et al. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2006.

[2] E. Allen et al. *The Fortress Language Specification*. Sun Microsystems, 2005.

[3] B. Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[4] S. Ananian et al. Unbounded Transactional Memory. In *Proceedings of the 11th Intl. Symposium on High Performance Computer Architecture*, Feb. 2005.

[5] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *ISCA Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[6] B. D. Carlstrom et al. The Atomos Transactional Programming Language. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2006.

[7] *Chapel Specification*. Cray, February 2005.

[8] P. Charles et al. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Conference on Object-oriented Programing, Systems, Languages, and Applications*. ACM Press, Oct. 2005.

[9] J. Chung et al. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th Intl. Conference on High Performance Computer Architecture*, Feb. 2006.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] R. Guerraoui et al. Robust Contention Management in Software Transactional Memory. In *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*, Oct. 2005.

[12] L. Hammond et al. Transactional memory coherence and consistency. In *Proceedings of the 31st Intl. Symposium on Computer Architecture*, June 2004.

[13] T. Harris. Exceptions and Side-effects in Atomic Blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[14] T. Harris et al. Composable Memory Transactions. In *Proceedings of the Symposium Principles and Practice of Parallel Programming*, July 2005.

[15] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th Conference on Object-oriented programing, Systems, Languages, and Applications*, pages 388–402. ACM Press, Oct. 2003.

[16] M. Herlihy et al. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd Symposium on Principles of Distributed Computing*, July 2003.

[17] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Intl. Symposium on Computer Architecture*, May 1993.

[18] Java Grande Forum, *Java Grande Benchmark Suite*. `http://www.epcc.ed.ac.uk/javagrande/`, 2000.

[19] J. Larus. It's the Software Stupid. Talk at the Workshop on Transactional Systems, Apr. 2005.

[20] V. Luchangco and V. Marathe. Transaction Synchronizers. In *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*, Oct. 2005.

[21] V. Marathe, W. Scherer, and M. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th Intl. Symposium on Distributed Computing*, Sept. 2005.

[22] A. McDonald et al. Characterization of TCC on Chip-Multiprocessors. In *Proceedings of the 14th Intl. Conference on Parallel Architectures and Compilation Techniques*, Sept. 2005.

[23] K. Moore et al. LogTM: Log-Based Transactional Memory. In *Proceedings of the 12th Intl. Conference on High Performance Computer Architecture*, Feb. 2006.

[24] E. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*, Oct. 2005.

[25] J. Nakano et al. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *Proceedings of the 12th Intl. Conference on High Performance Computer Architecture*, Feb. 2006.

[26] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[27] R. Rajwar and J. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

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

[29] M. F. Ringenburg and D. Grossman. AtomCaml: First-class Atomicity via Rollback. In *Proceedings of the 10th Intl. Conference on Functional Programming*, Sept. 2005.

[30] B. Saha et al. A High Performance Software Transactional Memory System for a Multi-core Runtime. In *Proceedings of the Symposium Principles and Practice of Parallel Programming*, Mar. 2005.

[31] N. Shavit and S. Touitou. Software Transactional Memory. In *Proceedings of the 14th Symposium on Principles of Distributed Computing*, Aug. 1995.

[32] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 1992.

[33] Standard Performance Evaluation Corporation, *SPEC CPU Benchmarks*. `http://www.specbench.org/`, 2000.

[34] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. `http://www.spec.org/jbb2000/`, 2000.

[35] G. Steffan, C. Colohan, Z. Zhai, and T. Mowry. A Scalable Approach to Thread-level Speculation. In *the Proceedings of the 27th Intl. Symposium on Computer Architecture*, June 2000.

[36] S.Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Intl. Symposium on Computer Architecture*, June 1995.

[37] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional Monitors for Concurrent Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, June 2004.