Solving Difficult HTM Problems Without Difficult Hardware

Owen S. Hofmann, Donald E. Porter, Christopher J. Rossbach, Hany E. Ramadan, Emmett Witchel

Department of Computer Sciences, University of Texas at Austin

{osh,porterde,rossbach,ramadan,witchel}@cs.utexas.edu

Abstract

There are several classes of operations, including I/O and memory allocation, that are considered difficult to perform as part of a transaction. To allow such operations inside of transactions, previous hardware transactional memory systems have proposed additional mechanisms such as *open-nested transactions* that use *hardware management of software handlers*. Open-nested transactions are not necessary, and add significant complexity to both HTM systems and the software written to take advantage of them.

MetaTM is an HTM system designed to run TxLinux, an operating system that uses transactions for some synchronization. Inside the operating system, it is necessary to efficiently handle I/O and memory allocation. MetaTM and TxLinux handle both of these without requiring the significant extra hardware or overhead associated with open-nested transactions. The TxLinux kernel uses *cooperative transactional spinlocks* which provide the concurrency of transactions with the mutual exclusion needed to perform I/O. Through *explicit management of transactional system calls*, TxLinux ensures strong atomicity for system calls within a transaction, providing user-level transactions with a powerful and simple transactional programming model.

1. Introduction

> The location of the point that separates useful transaction concepts (in the sense that they help simplify and better control applications) from those concepts that are too baroque (and thus create more problems than they solve) cannot be determined by rigorous measurement. It can, though, be determined to a certain degree by preferences of style, experience and other such criteria.
>
> –Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*

The transactional model of programming has recently gained attention because hardware manufacturers are finding it easier to scale the number of processing contexts on a chip rather than the performance of an individual context. Transactions can provide a simpler, more productive programming tool than traditional, lock-based code, which suffers from many well-known shortcomings, such as deadlocks, convoys, and priority inversions. Transactions may proceed in parallel as long as they do not have *conflicts* in which one transaction writes data read or written by another transaction. The transactional programming model can have performance equal to or greater than lock-based programming when the conflicts are rare. Transactions hold the promise of the perfect systems solution: a simpler programming model with the same or better performance.

The problem is that some transactional programming models, especially hardware-assisted models, have waxed baroque: the programmer's life is not simplified, and hardware implementation is complex. Recent hardware transactional memory (HTM) designs share some undesirable properties with CISC instruction sets—bloated primitives with expansive semantics that slow the common case. In particular, hardware models for transaction *nesting* can lead to implementing complicated semantics in hardware.

Nesting—allowing a parent transaction to contain many child transactions—is required for composability. Among the nesting models commonly proposed for transactional memory are *flat* and *open nesting*. In *flat nesting*, a child transaction is completely subsumed by the parent transaction. Data modified during a child transaction will be committed when the outermost parent commits, and any conflicts on data touched during a child transaction will cause the outermost parent transaction to restart. In *open nesting*, the modifications of a child transaction can be observed as soon as the child commits. The low-level isolation on the child's memory cells is replaced by isolation at a higher level of abstraction, e.g, memory allocated in an open-nested transaction is rolled back by a call to `free`, rather than by reverting the memory cell values of the allocator to the pre-transaction state.

Open nesting is used for several purposes: (i) to increase concurrency by avoiding low-level memory-cell conflicts (e.g. in the memory allocator) that do not represent conflicts at higher levels of abstraction (ii) to record information about transactions that never commit, (iii) to perform system calls inside transactions and (iv) to perform I/O in transactions. However, open nesting requires additional hardware and complicates the semantics of transactional memory systems. Moravan et al. [12] offer the most thorough examination of the semantic difficulties associated with open nesting,

and show that efficient implementations of open nesting impose restrictions on the average programmer that we believe are easy to violate. Open nesting also requires modifying data structures to enable control of isolation at higher levels of abstraction.

Hardware open nesting requires invoking stacks of abort handlers or compensating actions [10]. These handlers must synchronize themselves [8], and thus cannot easily perform actions such as I/O, without reverting to primitives such as locks. Open nesting also increases hardware complexity, adding features such as hardware management of software commit/abort handlers. Such generality is not needed. When software must be invoked on a transaction commit, restart or abort, a simple return code from the instruction that starts the transaction suffices. A return code renders superfluous elaborate schemes where the hardware manages stacks of handlers.

We believe that open nesting is a poor fit for hardware transactional memory (HTM), and that simpler solutions can provide all of the functionality and performance of open nesting without complicated hardware or semantics. This paper makes the case for flat-nested transactions with return codes and elimination of hardware-managed software handlers. It illustrates how problems for which open nesting is proposed as a solution, such as system calls and kernel I/O, can be solved with simpler and more effective mechanisms.

Section 2 describes the architectural mechanisms (Section 2) of our HTM model. Section 3 presents data for the low-level memory-cell sharing patterns of memory allocators and system calls in the Linux kernel, concluding that it is better to avoid synchronization than optimize it. Section 4 proposes *informing transactions* that enable detailed logging of transactional control flow while attempting to minimize access to uncommitted transaction state. In Section 5 we consider how the kernel can provide a transactional abstraction of system state. In Section 6 we present a novel mechanism for allowing I/O in transactions called cooperative transactional spinlocks. Section 7 describes related work, and Section 8 concludes.

2. The MetaTM framework

MetaTM is our implementation of hardware transactional memory, described in detail in previous work [20]. MetaTM appears as an x86 shared memory multiprocessor or chip multiprocessor with additional instructions for providing transactional features. Conflict detection and version management in MetaTM are eager [11]; conflicts are detected and handled as they occur, and updated values are written in-place in main memory and must be rolled back on a restart. MetaTM provides *strong atomicity*, meaning that conflicts between transactional and non-transactional threads must be resolved in favor of non-transactional threads [4].

Table 1 shows the hardware primitives used in MetaTM to control transactions. Transactions are started and ended by **xbegin** and **xend**, respectively. MetaTM uses *flat nesting*: if a thread that already has an active transaction invokes **xbegin**, the nested transaction is subsumed by the outer transaction. The thread ends its transaction when **xend** has been called the same number of times as **xbegin**. This nesting model is trivial to implement, as it only requires tracking the nesting depth.

Flat nesting is a type of *closed nesting*, which also subsumes child transactions within parent transactions. Closed nesting allows for partial restart [12]. When a nested transaction commits its data, it becomes part of the uncommitted data of its parent. When a transaction rolls back, it must roll back only to the first ancestor without conflicts. Partial rollback can reduce the amount of lost work on a data conflict. While MetaTM does not support closed nesting, adding support for closed nesting would not complicate the transactional programming model, which is our primary design goal.

MetaTM provides support for multiple active transactions per thread through the **xpush** and **xpop** primitives. These primitives suspend a transaction, making a transactional thread non-transactional. Once a transaction is suspended, the same thread may start a new transaction that is independent from the paused transaction. Transactions are pushed and popped in LIFO order. Independent transactions are simpler to reason about than nested transactions. The **xpush** primitive is distinguished from the similar **xact_pause** [23] and escape actions [12] primitives from the literature, by disallowing a thread to read uncommitted transaction state from a suspended transaction and allowing a thread with a suspended transaction to start a new transaction.

The **xpush** and **xpop** primitives allow transactional threads to execute non-transactional code. Updates to memory will be visible immediately and will not be rolled back if the outer transaction aborts. If the outer transaction aborts and then follows a different code path, these updates will have appeared to occur with no cause. Because using **xpush** and **xpop** requires significant effort in reasoning and code structuring, most programmers should not use these primitives. Instead they allow systems programmers to provide services such as system calls and transactional introspection that are normally difficult to implement in TM systems.

A transaction's ability to communicate information about why it has restarted is the foundation for implementing higher-level transactional abstractions in software. The **xbegin** primitive returns a status code with information about the active transaction. From the status code, a thread may determine whether the current transaction has restarted, and what caused the restart. A restart may have been caused by a conflict with another thread or an action within the transaction that is either not allowed, or requires different exclusion guarantees. I/O, as we shall see shortly, cannot be performed in a simple transactional context: MetaTM restarts

Primitive	Parameters	Returns	Definition
xbegin		*restart_reason*	Begin a transaction. Returns a status code indicating if and why this transaction has restarted.
xend			End a transaction, committing if there are no outstanding conflicts.
xrestart	*[restart_reason]*		Force the current transaction to restart. Allows a parameter to be passed that will be returned as a result of **xbegin** when this transaction restarts.
xgettxid		*tx_id*	Determine if there is an active transaction. A return value of 0 means no active transaction.
xpush			Suspend the active transaction.
xpop			Resume a transaction suspended with **xpush**.
xwait	*mem_addr, wait_val*		Wait for a memory location to take a specific value and then place it in the read set of the active transaction if it exists. Subjects non-transactional threads to contention manager policy.
xwait_CAS	*mem_addr, wait_val, new_val*		Wait for a memory location to take a specific value and atomically swap in a new value. Subjects non-transactional threads to contention manager policy.

Table 1. Transactional memory instruction set of MetaTM

such transactions and sets a corresponding flag. Users may also initiate a restart with the **xrestart** primitive. In this case, a user-defined code may be passed that will be returned from **xbegin** when the transaction restarts.

MetaTM also provides **xwait** and **xwait_CAS**, two primitives that allow transactional critical sections to interoperate with critical sections protected by locks. The **xwait** primitive stalls a processor until a memory location takes on a specified value. When **xwait** is called from within a transaction, the address is then added to the read-set of the transaction. **Xwait_CAS** is similar to **xwait**, but also atomically swaps the stored value. Both transactional and non-transactional threads are subject to the contention manager's policies when using these primitives. Section 6.4 describes how this allows transactional and non-transactional threads to contend fairly for the same critical section.

The **xwait** primitives allow threads to busy-wait without executing instructions, therefore allowing the processor to reduce its energy consumption. To implement **xwait**, the cache controller snoops the bus (or enters the directory notification list) for the line containing the **xwait** address. The **xwait_CAS** primitive further requires the controller to get exclusive access to the line if the current value of the variable equals the *wait_val*.

Unlike open nesting, MetaTM's extensions to the basic transactional model do not encode complicated semantics in hardware. However, they allow programmers to implement solutions to several problems that plague HTM systems, as illustrated in Sections 4, 5 and 6.

3. Low-level conflicts

One risk of flat nesting or closed nesting is that low-level conflicts can hinder progress of operations that do not have conflicts at higher levels of abstraction. Consider two threads that each insert elements into a shared hash table. The two threads may each insert elements with different keys, and thus do not have conflicts within the hash table abstraction. Inserting those keys, however, may cause lower-level conflicts due to the implementation of the hash table. For instance, if the two keys map to the same hash value then the insertion of both elements may cause a conflict. Open nesting solves this problem by releasing isolation after the operation is performed, while continuing to detect conflicts at higher levels of abstraction, in this case by detecting operations on the same key. Open nesting, however, is complicated and costly. Using open nesting over flat or closed nesting is only justified if low level conflicts occur often enough to reduce performance. In this section we examine the likelihood of low-level conflicts in two areas that are often considered difficult for transactions: memory allocation and system calls.

Conflict source	Conflicting Calls	Avg. Conflict (Bytes)
Allocation	1.5%	6
System calls - self	45.8%	26
System calls - other	9.1%	17

Table 2. Low-level conflicts in memory allocation and system calls in the Linux kernel running the Modified Andrew Benchmark. Conflicting Calls is the percent of all calls that would conflict if invoked during a transaction. Avg. Conflict is the average number of bytes that conflict for calls with any conflicts.

3.1 Open nesting for memory allocation

Memory allocation is a frequently invoked example of low level conflicts possibly hindering forward progress [12, 13, 23]. Memory allocators that are shared among threads must maintain some global pool of memory. Transactions that

Allocator	Tx/sec	Expected Restarts	Time
SLOB - tx	25306	32.45	> 25.00s
Slab - tx	576608	0.02	4.27s
Slab - nontx	–	–	4.29s

Table 3. Memory allocator performance in TxLinux for a Modified Andrew Benchmark, including the number of transactions created per second, the expected number of restarts for each transaction, and execution time

allocate memory may unnecessarily conflict on this global state. This example is especially salient with a simple heap-based allocator with a single free list, such as the Simple List Of Blocks (SLOB) allocator in the Linux kernel. Any allocation or deallocation both reads and writes a shared pointer to the free list, so there is no possibility for concurrency between transactions that allocate memory.

Although simple memory allocator implementations will exhibit many low-level conflicts, memory allocators designed for high performance avoid synchronization, primarily by avoiding global data sharing. Hoard [2], McRT-Malloc [9] and the Slab allocator used in Solaris and Linux [6] achieve performance and parallelism by avoiding sharing and synchronization between threads, dramatically reducing the frequency of unnecessary low-level conflicts. Although these allocators will occasionally cause unnecessary low-level conflicts, open nesting for memory allocation is only justified if conflicts are frequent.

3.2 Measuring low-level conflicts in Linux

To examine the probability of low-level conflicts in the Linux kernel, we used *syncchar*, a tool for collecting detailed data about synchronization, including addresses read and written during critical sections [16]. Every time the memory allocator was called, we compare the working set of that operation to the 128 previous operations. The results are presented in Table 2. The vast majority of memory allocator operations (over 98%), do not conflict with each other. Even if transactions use flat nesting or closed nesting for memory allocation, spurious conflicts are unlikely.

Linux's scalable allocator has few spurious conflicts. Table 3 shows performance on a Modified Andrew Benchmark for transactional Linux kernels with Slab and SLOB allocators, and a non-transactional kernel with the Slab allocator. Under the SLOB allocator, 97% of transactions must restart. With the Slab allocator, less than 2% of all transactions restart (including restarts not caused by the allocator). The SLOB allocator decreases performance by a factor of at least 5, while the slab maintains its performance when using transactions. Scalable memory allocators cause few low-level conflicts, so the complexity of open nesting is not necessary to maintain a high level of concurrency (a result also demonstrated by Moravan et al. [12]. We believe that the complexity of a scalable allocator is preferable to the programming complexity of open nesting.

```
xbegin;
...
if(value > 10) {
    xpush;
    printf("Value is greater than 10\n");
    xpop;
}
...
xend;
```

Figure 1. Illustration of using **xpush** and **xpop** to release isolation during a transaction.

System calls also present the possibility of low-level conflicts hindering forward progress. Through system calls, even unrelated threads can modify shared state in the kernel. As with memory allocation, we measured conflicts between system calls using syncchar (Table 2). When calling different functions, over 90% of dynamic system calls are data independent—they contain no conflicts. When calling the same function, the majority of calls are data independent.

4. Recording transactional events

In transactional systems, events that occur on control paths that end in transaction restart are never visible outside of the transaction. This is often the desired behavior. When transactions are used to express optimistic concurrency, the user generally considers completed transactions as work accomplished and uncompleted transactions as work wasted, the equivalent of time spent busy-waiting in a lock-based program.

Sometimes a program wants to record events that happen during transactions that do not commit. Open nesting proposes a solution to this problem by dropping isolation before recording the event, without registering an abort handler (or compensatory action) to remove the record. The open nested transaction can read and record uncommitted transaction state, even for transactions that will eventually abort.

MetaTM provides two ways to record events during an uncommitted transaction, depending on the type of data. MetaTM provides **xpush** and **xpop** to suspend the current transaction and record anything that does not involve uncommitted transaction state. On an **xpush**, the thread becomes non-transactional, so it no longer has access to the uncommitted state of the **xpush**ed transaction, however the thread's position in program control flow can provide useful information, as shown in Figure 1. A programmer can often construct the critical region such that recording only the PC provides vital clues to how transactions progress.

MetaTM allows indirect access to information about transactions that never commit through *informing transactions*. As shown in Figure 2, several architectural registers provide information about the current transaction. These

```
xbegin;
xpush;
last_pc = read_reg(TX_LAST_PC);
restart_reason = read_reg(TX_RESTART_REASON);
printf(
   "Transaction restarted at %p because %d",
   last_pc, restart_reason);
xpop;
...
xend;
```

Figure 2. Illustration of using informing transactions to record detailed transactional control paths.

registers may be read even when the transaction has been suspended with **xpush**, allowing the programmer to record information about uncommitted transactions. Through these registers the programmer may access the return code of the previous **xbegin**, and thus determine if the transaction has restarted and why (either through hardware-set flags or codes passed explicitly to **xrestart**). The programmer can also read the address of the last instruction executed by the transaction before restarting, providing a detailed view of transaction control flow.

Programmers can use informing transactions for performance debugging of the transactional system itself. They are flexible enough to perform most types of introspection into the transactional system. The programmer has control over how much the return codes and the PC of the last transaction instruction indicate about a program's semantics. It is easy for a programmer to assign different return codes to each event she cares about in an uncommitted transaction. The PC provides additional execution context, avoiding the need to create excessive quantities of return codes.

Informing transactions do not provide the same level of access to uncommitted transaction state as open-nested transactions. While informing transactions allow less information to flow out of a transaction than open nesting, information leakage generally leads to semantic difficulties. Leaked information can compromise the correctness of compensating actions that are intended to provide isolation for unsuccessful transaction attempts. Informing transactions expose some implementation details of the HTM (different versions of the hardware might provide different sets of informational registers), but this is a more manageable problem than the erosion of transactional semantics caused by open nesting.

Another problem with recording transactional events is that the code that records events can cause conflicts between transactions that would not normally conflict. For instance, events could be recorded by updating a global pointer to an output buffer. This problem is not specific to recording transactional events, but is a specific example of the general problem of low-level memory-cell conflicts that do not reflect high-level data structure conflicts. This is addressed in Section 3.

5. Transactions and the user/kernel boundary

The relationship between user and kernel mode transactions is central to a successful programming model. The issues of I/O in a transaction and a system call in a transaction are conflated by the current literature [5, 12, 23] and the conflation has harmed the programming model, e.g., system calls made within a user transaction are not isolated, and several proposals forbid the OS from starting a transaction if it is called from a user-level transaction [12, 23]. We believe that the operating system, as a performance-critical parallel program with extremely complicated synchronization, should be able to benefit from transactional memory [20].

Transactions require the ability to rollback, and it can be impossible to rollback I/O. A pure transactional model cannot accomodate I/O without resorting to heavy-weight mutual exclusion, i.e., only one, globally distinguished transaction that can perform I/O at a time [5]. However, only the OS kernel actually changes the state of I/O devices, so it is possible to shield user-initiated transactions from the I/O problem. MetaTM and TxLinux provide a pure transactional model for user code, even user code that includes system calls, while they mix transactions and locking in the operating system kernel (explained in Section 6).

5.1 Handling system calls with open nesting

Existing proposals for performing system calls during a transaction, including escape actions [12], **xact_pause** [23], and unrestricted transactions [5], address problems of non-idempotency and rollback of system calls. These proposals, however, do not necessarily maintain isolation between transactional and non-transactional threads, or even among transactional threads. Even open nesting requires significant OS modifications to provide the atomicity and isolation guarantees of transactional memory.

Open nesting is one proposed mechanism for allowing system calls in transactions [10]. When a thread in an active transaction enters the kernel, an open-nested transaction is started. For system calls with side effects, the open-nested transaction registers an abort handler to undo the effects of the system call in case the outer transaction must restart.

Open-nested transactions allow transactional threads to perform actions with side effects on kernel data structures. Because isolation on kernel data is released when the open nested transaction commits, conflicts between unrelated system calls are reduced. This approach has several problems. In theory, open-nested transactions should be able to update data modified by an ancestor [10], e.g. a call to `read` may fill a user-supplied buffer. Existing proposals that contain enough information to implement open nesting in hardware [12], however, do not allow this property. Moravan

```
void
process_queue(queue *q, int rfd, int wfd){
    char buf[4096];
    int nbytes, i;
    xbegin;
    // User writes buf
    nbytes = write_output(q, buf, 4096);
    write(wfd, buf, nbytes);
    // Syscall writes buf
    nbytes = read(rfd, buf, 4096);
    parse_input(q, buf, nbytes);
    xend;
}
```

Figure 3. Code to write a response to one file descriptor (wfd), and then read and parse a new request from another file descriptor (rfd) while reusing a temporary buffer. The buffer is written both by the user code (in `write_response`), and by the `read` system call. The code illustrates the difficulty of guaranteeing that a buffer passed to a system call is not modified by the user.

et al. refer to the condition that a descendant transaction may not write the same data written by an ancestor transaction as O1, and they encourage most programmers to write code that obeys the condition. Violating O1 risks subtle issues surrounding the leakage of uncommitted transactional state [13], and semantic complications for undo actions.

Consider the code in Figure 3, where the user writes a response to one request and then reads and parses a new request. The `buf` buffer is written both by the user (in `write_response`) and by the system in an open-nested transaction for the `read` syscall. This directly violates condition O1.

There are many subtler ways a programmer can violate condition O1. For instance, in Figure 3, instead of stack-allocating a single buffer, the user might allocate and free the buffer for the write, and again allocate and free the buffer for the read. However, an allocator that uses LIFO ordering of memory blocks (which increases the probability of dynamically allocated memory being in-cache) could return the same memory for both calls to `malloc`, creating the same reuse problem in Figure 3.

Condition O1 unacceptably complicates the programming model of transactions. Code that is correct with one memory allocator might fail with another. Code that was correct can become incorrect if the user initializes a buffer written by a system call. While Moravan et al. admit that an advanced programmer can violate O1 under certain circumstances, the above discussion illustrates how system calls in open-nested transactions invite subtle, difficult bugs into software.

In order to avoid transactions in the operating system, there have been proposals to pause transactions and execute system calls in a non-transactional context [12, 23] called an *escape action*. Unfortunately, Moravan et al. identify condition X1, which is identical to O1, but applies to these non-transactional escape actions. Because escape actions can read uncommitted transactional state, they have the same problems when an ancestor transaction writes the same memory as an escape action. Using escape actions for system calls invites the same subtle, difficult bugs into the software as open-nested transactions.

In addition to the semantic quirks associated with implementing open nesting, there are cases where user-level commit and abort handlers cannot undo the effects of system calls. Suppose a thread maps `file_A` into its address space, as in Figure 4. The file is then closed and unlinked, so that the memory mapping is the only remaining link (keeping the file from being reclaimed by the file system). The thread begins a transaction, and then maps a different file over `file_A`. This mapping implicitly removes the first, thus causing deletion of `file_A` by removing its last link. No compensating action registered when calling `mmap` can be sufficient to undo the effects of the system call—it cannot restore the original mapping because the filesystem has reclaimed `file_A`. If the transaction must restart, the system will be unable to restore pre-transaction state, compromising correctness.

One technique to make open-nesting or escape actions a more powerful programming technique is to modify data structures to manage uncommitted results explicitly. Moravan et al. use the example of a B-tree with a lock field that is set explicitly in open-nested transactions and then cleared on a commit. The performance impact of commit handlers is not discussed in this work, but other work on the Linux kernel [20] indicates that performance degrades noticeably

```
fd_A = open("file_A");
void *map_A = mmap(size=4096, fd=fd_A)
close(fd_A);
unlink("file_A");

xbegin;
modify_data(map_A);
fd_b = open("file_B")
void *map_B = mmap(start=map_A, size=4096,
    fd=fd_B);
xrestart;
```

Figure 4. Code that maps two files at the same address. The program closes and unlinks `file_A`. Mapping `file_B` over `file_A` implicitly removes the mapping for `file_A`, thus removing the last link to `file_A` and causing the file to be deleted. When the transaction restarts, no compensating action can return it to its original state. The code illustrates that user-level abort actions (as a part of the `mmap` call) may not be able to roll back the effects of common system calls.

(more than 10%) once commit handlers average more than 1,000 cycles. Concurrent accesses to the B-tree that find entries with the lock field set ignore those entries. Managing isolation explicitly is the subject of the next Section.

5.2 Decoupling I/O from system calls

Most system calls, even those that change state visible to other processes, do not actually change the state of I/O devices. For example, creating a file in the file system changes kernel data structures, it does not (necessarily) write anything to disk. If TxLinux can buffer in memory the effect of system calls initiated by a user transaction, then it can decouple I/O from system calls.

The task of decoupling I/O from system calls reduces to making sure enough system resources are available for a user-initiated sequence of system calls to complete having updated only memory. To achieve this, the OS might need to free system resources, e.g., creating more free memory by writing back data from the disk cache that is unrelated to the current transaction. In order to free up resources, the kernel **xpush**es the current transaction, and performs the I/O outside of the transactional context. Enough information must leak out of the transaction to let the kernel learn the type and amount of resources that must be made available.

If the kernel cannot free enough resources to perform a user-initiated sequence of system calls using only memory, then it kills the user process. Transaction virtualization is important for hardware limits like cache size, but MetaTM cannot support a transaction whose updates are larger than available memory.

5.3 Explicit OS management of transactional syscalls

We now sketch how an OS could be modified to support transactional system calls by explicitly managing atomicity and isolation. Some system calls can safely execute non-transactionally (using **xpush**) even when called from a user-level transaction, e.g, `getpid`. When a user program makes a system call that is not always safe, the OS marks the transaction status word, setting the `syscall` bit. The kernel performs any necessary actions to allow the user thread to proceed as if the call were not speculative, such as reading a file into the page cache. The kernel maintains the necessary information to commit or undo the effects of all speculative system calls. On an attempted commit of a transaction that has the `syscall` bit set, the hardware traps to the OS. The OS tries to commit the effects of all of the system calls performed by the current transaction. In order to commit the effects atomically, the OS can use a hardware transaction.

To avoid flowing large amounts of information out of a transaction, TxLinux and MetaTM allow user-level transactions to flow into the kernel. Consider the `write` system call which communicates a large user-supplied data buffer to the kernel. In Linux, the majority of the code path for `read` and `write` system calls deals with bringing the necessary pages into the page cache, actions which do not change the abstract state of the system. The necessary data is copied to or read from the user buffer once the OS has made space in the page cache. To execute these system calls transactionally, TxLinux uses **xpush** to suspend the user transaction. The necessary pages are brought into the page cache. TxLinux then executes **xpop** to resume the user transaction while still in kernel mode. The speculative data is copied to or read from the page cache before returning to user mode. Thus, the TM hardware manages versioning and conflict detection for file data, providing the same strong atomicity as for memory updates.

Different transactional threads might perform system calls that conflict, e.g., both try to open the same file name with exclusive permissions. In this case, the OS contention manager decides which thread should win, and updates the data structures accordingly. When the losing thread makes its system call while trying to commit, the system call returns with an error code. Either the OS or the hardware can restart the user transaction.

Explicit tracking of transactional system calls is similar to explicit tracking of speculative state, e.g., RPCs in a networked file system [15]. System calls made during a transaction are speculative until the transaction commits. This method also limits user-initiated transactions to sequences of system calls whose effects can be buffered in memory. It requires substantial OS programming effort.

5.4 Limitations to system calls in a transaction

TxLinux and MetaTM do not allow transactional code that would increase the sphere of control of a transaction beyond the current OS thread. User-level code that starts a transaction and then writes a request to the network and reads a reply cannot remain isolated without propagating the transaction across the network. User-level code that does an interprocess request and reply requires propagating the transaction from one OS process to another. These cases are beyond the scope of this work.

A programmer can read from and write to the network during a transaction, but the semantics in TxLinux and MetaTM are that the read is satisfied from kernel memory buffers, and the write goes to kernel memory buffers. This is a generous programming model, and anything stronger (e.g., actually allowing network communication) is contrary to most intuitive definitions of a transaction.

System calls that are conceptually synchronous with respect to an I/O device are not allowed or they are ignored. The `fsync` system call specifies that the file data being synced is on disk before the system call returns. These semantics conflict with the intuitive definition of a transaction. What does it mean to ensure that the state updated by a partially executed transaction must reside on disk? The programmer must verify that the system calls executed in a transaction do not have semantics directly at odds with the semantics of transactions.

6. Cooperative transactional locking

The effects of I/O during a critical section are difficult or impossible to roll back in the event of a transaction abort. Writes to a disk device, for example, would require significant additional logic and device support to roll back. With open nesting, the programmer is able to register compensating actions to undo the effects of I/O without having to incorporate such logic into hardware.

This section describes the cooperative transactional spinlock (cxspinlock), a construct that provides an alternative mechanism for I/O in an operating system kernel. Cxspinlocks are only needed inside the OS kernel. They allow a single critical region to be safely protected by either a lock or a transaction. A non-transactional thread can perform I/O inside a protected critical section without concern for undoing operations on a restart. Many transactional threads can simultaneously enter critical sections protecting the same shared data, improving performance. Simple return codes in MetaTM allow the choice between locks and transactions to be made dynamically, simplifying programmer reasoning.

6.1 Programming with cxspinlocks

As with traditional locking, shared data protected by cxspinlocks is associated with a lock variable. However, cxspinlocks do not always enforce mutual exclusion, instead the functions `cx_optimistic` and `cx_exclusive` are used to choose between mutual exclusion and transactions when acquiring a cxspinlock. If the protected code path requires mutual exclusion, or must be guaranteed not to restart, then `cx_exclusive` is used. Otherwise, `cx_optimistic` is used to protect the same shared data using transactions.

All occurences of a spinlock acquire can be mechanically replaced by `cx_optimistic` and spinlock releases can be replaced by `cx_end`. However, if a particular critical region always performs I/O, the `cx_optimistic` will always restart, which might be a performance concern. In these cases the programmer can optimize the system by calling `cx_exclusive` directly (see Section 6.3). Calls to `cx_optimistic` and `cx_exclusive` nest arbitrarily (Section 6.3).

Because any critical section protected by `cx_optimistic` may be forced to revert to mutual exclusion, the programmer must reason about which locks protect which shared data. In addition, standard locking practices such as enforcing a global lock order must be followed. These necessities seem contradictory to the goal of transactional memory. However, because most uses of cxspinlocks will be transactional, they allow locking structure to be coarse-grained, while retaining the concurrency of fine-grained locks. Reasoning about coarse-grained locks is far easier, and cxspinlocks allow advanced behaviors, such as enforcing mutual exclusion and performing I/O, that are not usually possible when sharing data protected by transactions.

```
void cx_optimistic(lock) {
    status = xbegin;
    // Use mutual exclusion if required
    if(status == NEED_EXCLUSIVE) {
        xend;
        // xrestart for closed nesting
        if(gettxid) xrestart(NEED_EXCLUSIVE);
        else cx_exclusive(lock);
        return;
    }
    // Place the unlocked lock in the read-set
    xwait(lock, 1);
}
void cx_exclusive(lock) {
    if(gettxid) xrestart(NEED_EXCLUSIVE);
    // Wait for 1, atomically make it 0
    // Contention manager arbitrates lock
    xwait_CAS(lock, 1, 0);
}
void cx_end(lock) {
    if(xgettxid) {
        xend;
    } else {
        *lock = 1;
    }
}
```

Figure 5. Functions for acquiring cxspinlocks with either transactions, or mutual exclusion.

In TxLinux, we have used cxspinlocks to convert a number of critical sections that perform I/O to use transactions, and have found them to be ideal cases for cxspinlocks. The converted critical sections all perform I/O at some time, but any individual critical section performs I/O at most 5% of the time. Without cxspinlocks, these critical sections would be forced to always use mutual exclusion. With cxspinlocks, we have observed as many as 14 concurrent threads entering these critical sections on a 15 processor system.

6.2 Implementation

The code for `cx_optimistic` and `cx_exclusive` is given in Figure 5. These functions comprise the main kernel API to create transactions. Either `cx_optimistic` or `cx_exclusive` can start a critical section and `cx_end` ends a critical section. A critical section might allow many, non-interfering threads to execute inside it at once (`cx_optimistic`), or it might allow only a single thread to execute inside it at a time (`cx_exclusive`).

The `cx_optimistic` call creates a transaction and uses the **xwait** primitive to wait for the lock protecting the critical region to hold the value indicating an unlocked condition, typically 1. When this condition is met, the lock variable is added to the set of addresses read by the transaction. While the lock is unlocked, transactional threads can share

the critical region. The `cx_end` function ends the transaction started by the call to `cx_optimistic`.

The `cx_exclusive` function is an implementation of a traditional spinlock, but one that takes advantage of HTM resources to arbitrate for access to the lock, using the **xwait_CAS** instruction. The **xwait_CAS** instruction combines a condition variable-style `wait` with a compare and swap. The instruction blocks until the given memory address has value1, and it then atomically swaps value2 into the memory location. If the swap is unsuccessful, the primitive continues to block.

6.3 Nesting cxspinlocks

Threads may not restart inside a critical section protected by `cx_exclusive`. To allow `cx_exclusive` to be nested inside `cx_optimistic`, return codes are used to communicate information about transaction restarts. If `cx_exclusive` is called during an active transaction (determined using **xgettxid**), **xrestart** is invoked and passed the NEED_EXCLUSIVE flag. When the transaction restarts, this flag is returned from **xbegin**. If `cx_optimistic` receives the NEED_EXCLUSIVE flag, it ends its transaction. If closed nesting is used, `cx_optimistic` must verify that it is no longer in a transaction, and possibly invoke **xrestart** to pass the flag to the outermost transaction.

Using return codes, critical sections protected by either transactions or mutual exclusion may be arbitrarily nested inside each other. However, certain nesting patterns may cause pathologically bad performance. If N transactional critical sections protected by `cx_optimistic` are nested, and then `cx_exclusive` is called, the thread will return to the outermost transaction. The next critical section, however, will optimistically begin a transaction. The thread will likely follow the same code path and again be forced to restart, reverting the next deepest transaction to mutual exclusion, and so on. The transaction will restart N times, wasting a significant amount of work. One solution would be for the outermost lock to record its address in per-thread storage space. Until that lock is released, all nested calls to `cx_optimistic` immediately fall back on `cx_exclusive`. The outermost lock then removes its address when it is released.

Although `cx_optimistic` and `cx_exclusive` can be safely nested, invoking only **xbegin** without checking the return code may result in deadlock if the transaction latter requires exclusion by either performing I/O or by calling `cx_exclusive`. The cxspinlock implementation, however, requires very few instructions and will not adversely affect performance over using raw transactions. **Xbegin** should be used only as an optimization when the code path of the critical section is statically known.

6.4 Role of the contention manager

In transactional memory systems, the contention manager is responsible for deciding which transactions restart when conflicts occur, and is thus responsible for thread progress. All contention for memory resources should go through the contention manager. MetaTM provides strong atomicity: conflicts between transactional and non-transactional threads must be resolved in favor of non-transactional threads. For fairness, however, non-transactional threads should not always be given priority when contending for critical sections protected by cxspinlocks. MetaTM uses the **xwait** and **xwait_CAS** primitives to subject non-transactional threads to the decisions of the contention manager, which may enforce any policy.

A transactional thread entering a critical section uses the **xwait** primitive to stall until the lock variable changes to the unlocked state. The variable is then placed in the read-set of the transaction. Any transaction inside the critical section will have the lock variable in its read-set. A non-transactional thread uses **xwait_CAS** to both wait on the lock as well as atomically acquire the lock. If a non-transactional thread attempts to enter a critical section currently in use by a transactional thread, the contention manager can decide whether to allow the swap in **xwait_CAS**. The contention manager can choose to stall the non-transactional thread and allow the transaction to complete, or to abort the transaction and allow the non-transactional thread to acquire the lock.

The fast path for contention management should be in hardware, but complicated cases can trap to the operating system. The contention manager can bias related critical regions to prefer non-transactional or transactional threads. For instance, critical regions protected by a reader-writer lock can be biased in favor of transactional threads, because multiple readers are the common case.

6.5 Detecting I/O

With cxspinlocks, I/O is handled by restarting a transaction and falling back on locking, so that restarts are not possible. The MetaTM model assumes that the processor can *detect* I/O, e.g., the opcodes in the x86 that write to I/O ports or devices mapped in the memory space. On detecting I/O during a transaction, the processor restarts the transaction, returning the NEEDS_EXCLUSIVE flag from the **xbegin** instruction. The hardware does not roll-back I/O, it simply rolls back any transaction in which I/O occurs, before the I/O operation is issued. The thread then re-executes the critical section, but after acquiring the lock by calling `cx_exclusive`.

6.6 Disadvantages of mixing locks and transactions

Allowing a critical section to be protected by both locks and transactions brings the concurrency of transactions to code which previously would have been incompatible, such as functions that only perform I/O on some code paths. This cooperation, however, also reintroduces some of the problems that transactions are intended to solve.

Like spinlocks, cxspinlocks can enforce mutual exclusion for non-transactional threads. A poor locking discipline can

lead to deadlock, a problem that would normally be solved by transactions. Lock ordering is still needed, but only OS code must handle this complexity and it can be avoided in user code (Section 5).

7. Related Work

Proposals for hardware transactional memory systems incorporate a range of nesting semantics, from flat nesting [1, 7, 19], to closed nesting and open nesting [10, 12]. Most relevant to this paper are the models in LogTM and TCC. With LogTM, Moravan et al. offer the most thorough examination of the semantic difficulties associated with open nesting. We believe that these problems along with the complicated implementation necessary for open nesting make it inappropriate for HTM. In systems similar to TCC, however, threads always execute transactionally. Open nesting is necessary for communicating between threads as well as other operations that require isolation to be released. In these cases, a simpler hardware implementation might serve to accelerate a software implementation that provides more complicated semantics such as open nesting.

Unrestricted Transactional Memory [5] uses *unrestricted* transactions for performing I/O and system calls, as well as simplifying handling for transactions that overflow hardware resources. Unrestricted transactions cannot restart. However, because no reasoning is possible about shared data protected by unrestricted transactions, only one unrestricted transaction may execute at any time. An unrestricted transaction must stall all other threads in an application. Later work [3] presents an implementation that allows multiple non-transactional threads to execute concurrently with a single transactional thread. Critical sections protected by `cx_optimistic` handle I/O similarly by falling back on `cx_exclusive`, which cannot restart. However, because the shared data protected by cxspinlocks is associated with a lock variable, a critical section that must fall back on `cx_exclusive` does not restrict concurrency of unrelated transactional or non-transactional critical sections. The ability of unrestricted transactional threads to stall non-transactional threads can cause system deadlock in an operating system where non-transactional threads process interrupts and do other activity necessary for the continued operation of the system.

Some HTM designs [3, 19, 22] propose detecting conflicts only between threads within a single process or address space. Unrestricted transactions, for example, require this property for isolation; otherwise I/O or overflow within a single thread in one process could stall transactional threads in all other processes. We believe that this requirement is too restrictive, because it prevents different processes from using transactions to synchronize access to shared memory. Shared memory is used for inter-process communication, often in large, performance-critical applications such as the X server. These applications should be able to benefit from the simpler programming model offered by transactions. Operating system kernels are often mapped into the address space of every process in order to reduce switching overhead. Filtering conflict detection based on address spaces would make using transactions in the kernel much more difficult.

Cooperative transactional spinlocks use transactions to increase concurrency and fall back on locking when transactions fail, an approach similar to speculative lock elision [17, 18]. Analogous techniques have been used to improve performance in software transactional memory (STM) systems where the overhead of acquiring a lock can be much lower than starting a transaction [21]. Locks are used when contention is low, switching to transactions when contention is high and concurrency is more important. Cooperative transactional spinlocks are a programmer-visible construct that can be used for larger code regions and for more types of locks (e.g., reader/writer locks) than speculative lock elision. Moreover, cooperative, transactional spinlocks are intended only as a kernel primitive to satisfy the mutual exclusion required by real device I/O. User-level programs can take full advantage of the semantics and concurrency of transactions without requiring a locking discipline.

Zilles and Baugh [23] use the **xact_pause** primitive to suspend transactions and allow complicated behaviors such as I/O and system calls. Similar to our proposal for system calls, suspended transactions are used to implement software-managed stacks of commit and abort handlers, avoiding the complexity of a hardware implementation.

Open nesting is also used in STM systems for many of the same reasons as HTM systems. Unlike HTM systems, most STM systems do not provide strong atomicity, leading to situations which require some form of open nesting. For instance, memory allocators shared between transactional and non-transactional threads in an STM must use open nesting to isolate transactional threads from non-transactional accesses [9]. Open nesting in STMs shares the semantic and implementation caveats of HTMs. Ni et al. [14] address the care necessary in writing correct commit and compensating actions, and examine implementation issues posed by overlapping read-write sets between ancestor and open nested transactions. These issues are similar to those motivating condition O1 presented by Moravan et al. Although open nesting may be necessary in software TM systems, it remains a difficult programming construct.

8. Conclusion

Hardware transactional memory promises to make concurrent programming both simpler and faster. However advanced transactional features can complicate programming semantics rather than simplifying them; some advanced features are also difficult to implement efficiently in hardware. This paper argues that simple transaction models are effective for HTM systems. Simple transaction models are sufficient to record transactional events, make system calls in a

transaction, and manage I/O. Abandoning transactional features like open nesting simplifies the programming model without reducing the scope of what can be built with HTM.

9. Acknowledgements

We would like to thank the anonymous reviewers for their feedback on this paper. This research is supported by NSF CISE Research Infrastructure Grant EIA-0303609 and NSF Career Award 0644205.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 117–128, New York, NY, USA, 2000. ACM Press.

[3] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 24–34, New York, NY, USA, 2007. ACM Press.

[4] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

[5] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, University of Pennsylvania, Apr 2006.

[6] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[7] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 347–358, New York, NY, USA, 2006. ACM Press.

[8] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[9] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 2006 international symposium on Memory management*, pages 74–83, New York, NY, USA, 2006. ACM Press.

[10] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[11] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[12] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA, 2006. ACM Press.

[13] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[14] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, New York, NY, USA, 2007. ACM Press.

[15] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 191–205, New York, NY, USA, 2005. ACM Press.

[16] D. E. Porter, O. S. Hofmann, and E. Witchel. Is the optimism in optimistic concurrency warranted? In *Workshop on Hot Topics in Operating Systems*, May 2007.

[17] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[18] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 5–17, New York, NY, USA, 2002. ACM Press.

[19] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[20] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 92–103, New York, NY, USA, 2007. ACM Press.

[21] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *European Conference on Object-Oriented Programming*, pages 148–173, Jul 2006.

[22] L. Yen, J. Bobba, , M. Marty, K. E. Moore, H. Volos, M. D. Hill, , M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPC*. Feb 2007.

[23] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.