OS Support for Virtualizing Hardware Transactional Memory

Michael M. Swift, Haris Volos, Neelam Goyal, Luke Yen, Mark D. Hill, David A. Wood
University of Wisconsin–Madison
{swift,hvolos,neelam,lyen,markhill,david}@cs.wisc.edu

Abstract

Transactional memory promises to simplify multithreaded programming. Hardware TM (HTM) implementations promise better performance by augmenting processors with transactional state. However, HTMs interact poorly with the operating system or virtual machine monitor. For example, they often do not tolerate OS actions that virtualize processors and memory, such as context switching and paging. Without support for these actions, an HTM may not execute programs correctly or guarantee forward progress.

We investigate virtualizing transactional memory in the context of LogTM-SE. First, we describe an implementation of a kernel module in OpenSolaris that implements transactional virtualization and requires only 1120 lines of code. Second, we find that LogTM-SE interacts poorly with virtual machine monitors due to a reliance on physical addresses. We propose an extension to LogTM-SE, called LogTM-VSE, that addresses these problems and improves context-switching performance. Third, through application tracing on real hardware and full system simulation, we show virtualizing transactions can be necessary for system stability and to support code that voluntarily context switches. However, we find that aborting a transaction is generally faster than virtualizing it, and hence preferable in some cases.

1. Introduction

Transactional memory (TM) is emerging as a technique to simplify concurrent programming [10]. Hardware transactional memory (HTM) depends on processor state to accelerate conflict detection (to support isolation) and version management (to support aborts and atomicity).

The function of this hardware must be preserved when an OS takes actions that virtualize system state, such as reclaiming a processor or a memory page. Indeed, the recently proposed OpenTM API for transactional memory *requires* virtualization [2]. A common approach is to avoid the problem and abort transactions on a virtualization event. This mechanism is simple, fast, and effective in many situations. However, it has three significant drawbacks. First, the OS must reclaim resources in a bounded time, but several HTMs execute user-level code during aborts. This code, for restoring data to memory [21] or undoing nested transactions [17, 22], may block the OS from reclaiming resources and open it to denial-of-service attacks. Second, composing transactional code with code that acquires locks or performs blocking systems calls may require virtualizing transactions if the thread blocks on a lock or in a system call. It may not be possible to execute this code if transactions always abort on context switches. Third, aborting transactions reduces the generality of TM by limiting it to short transactions that do not exceed hardware or software resource limits. For these reasons, we investigate mechanisms for virtualizing transactional memory for those situations where it is necessary to allow transactions to survive context switches and paging.

This paper studies OS and virtual machine monitor (VMM) support for the *Log-based TM—Signature Edition* HTM [32]. We selected LogTM-SE as a platform because (1) it gracefully handles cache evictions in hardware without requiring virtualization, (2) it stores old versions of memory in a virtual-memory log, which is already virtualized, (3) it provides a hardware mechanism, *summary signatures*, for detecting conflicts with virtualized transactions, and (4) gives a short sketch of possible OS support for context switches and paging. This paper actually develops the needed OS support.

Contribution: OS implementation of virtualization. We implemented a kernel module in a commercial OS, OpenSolaris [29], that virtualizes transactional memory. This module, the Transaction Virtualization Manager (TVM), manages summary signatures to ensure continued conflict detection after context switches and page faults. TVM consists of 1120 lines of code that hook the kernel in nine places. We provide details of the algorithms how they interact with the kernel.

Contribution: Identifying challenges in virtualizing HTMs within a VMM. Virtual machine monitors also virtualize processors and memory with context switching and paging. We identify and fix two issues with virtualizing LogTM-SE in the presence of VMMs. First, a guest OS may not have access to physical addresses required for managing summary signatures. Second, when the VMM takes an action that requires virtualizing a transaction, it does not have information about what process is running or when a virtualized transaction completes.

We fix LogTM-SE's deficiencies with *LogTM-Virtual Signature Edition (LogTM-VSE)*, which incorporates modest hardware changes. It adds an additional signature containing virtual addresses and converts LogTM-SE's summary signature to virtual addresses. LogTM-VSE allows the OS to completely manage virtualization most of the time, with the VMM participating only when needed.

We sketch the design of a software system for virtualizing LogTM-VSE in both an OS and VMM. The guest OS virtualizes transactions with TVM under normal conditions and the VMM virtualize transactions with a separate implementation of TVM when it reclaims memory or processors.

Contribution: Exploration of virtualization's utility. Through tracing of multithreaded applications on real hardware, we show that virtualizing events, such as context switching and paging, occur regularly within the critical sections of existing lock-based programs. We find that aborting is often faster than virtualization and may suffice for involuntary context switches (when the kernel preempts a thread). However, it may not be appropriate in all cases, such as when the cost of aborting is high or when a transaction voluntarily blocks in the kernel.

We prototype LogTM-VSE and TVM in a full system simulator based on Simics [15] and Wisconsin GEMS [16]. In simulation, we

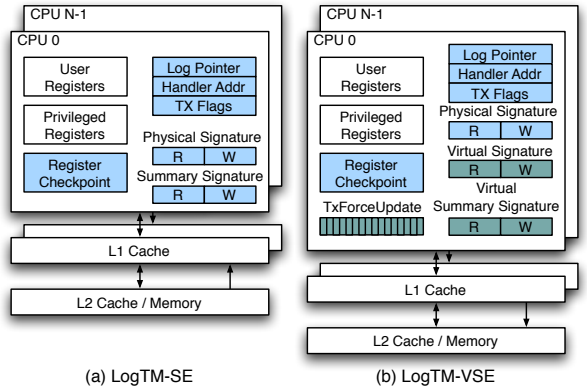

Figure 1. The (a) LogTM-SE and (b) LogTM-VSE (Section 4) architectures. Transactional memory structures are shaded and the VSE extensions are darkened.

show that aborting a transaction is generally faster than virtualization. However, in some cases aborts may require over a million cycles, and hence virtualization is then required for fast preemption. When virtualization is used for functionality, such as to support I/O or blocking on a lock, we show that the overhead is less than 7% for all our workloads, and more commonly under 3%.

In the next section, we review the LogTM-SE HTM architecture. We delay discussion of TVM until after we discuss the problems of virtualizing TM in a virtual machine in Section 3 and the LogTM-VSE extensions that fix these problems in Section 4. We then present TVM in Section 5 and describe how to extend its operation to a VMM in Section 6. Section 7 presents our empirical results. Section 8 discusses related work and we then offer concluding remarks.

2. LogTM-SE Hardware Review

We selected LogTM-SE [32] as a target platform because it provides the basic hardware required for virtualization. Figure 1(a) shows an example of the LogTM-SE architecture. LogTM-SE stores old values in a per-thread log in virtual memory. As a transaction executes, the processor logs the virtual address and old value of every memory block written. New values go directly to memory. On commit, LogTM-SE resets the log pointer. On abort, a software handler walks the log and copies old values back to their original memory locations.

LogTM-SE performs conflict detection with *signatures*, similar to Bloom filters [4], as transactions execute. On every load (store) instruction in a transaction, the processor hashes the physical address of the referenced cache block and adds the hash into a read (write) signature. Together, the read and write signatures form the signature of a transaction. When a processor misses in its cache and sends a coherence request for data, the receiving processors check their signature for conflicts. To prevent false conflicts between different address spaces (processes), the coherence request includes an address space identifier. If a potential signature conflict exists then the receiving processor checks the remote identifier against its local one and signals a conflict only if the address space identifiers match. To allow transactions to exceed the cache size, LogTM-SE requires that the coherence protocol forward memory requests for evicted blocks to processors that last accessed the block.

LogTM-SE resolves conflicts by stalling and aborting transactions. It initially stalls a transaction, waiting for the other to complete. If stalling does not resolve the conflict, it detects a deadlock. LogTM-SE relies on per-transaction *timestamps*, taken when the transaction began, to abort the younger transaction and let the older make progress. The timestamp is a loosely synchronized per-processor cycle counter.

The LogTM-SE hardware supports paging and context switching with a *summary signature* in each processor. As with regular signatures, summary signatures contain separate read and write fields computed from physical addresses. A LogTM-SE processor checks its own summary signature for conflicts on every load and store instruction and traps to software on a conflict. This signature is under software control, which uses it to summarize the conflict detection state of all virtualized transactions in a process. LogTM-SE supports interrupts without virtualization by *escaping* from the current transaction when entering the kernel and resuming the transaction on exit [22]. The LogTM-SE paper sketches requirements for OS support needed to virtualize LogTM-SE, but does not provide a complete design or implementation.

3. Challenges in Virtualizing TM in a Virtual Machine

Virtual machines are rapidly becoming ubiquitous and perform the same actions (context switching and paging) as an OS to virtualize the processor. In the presence of hardware transactional memory, the VMM must support virtualizing transactions to provide these services.

While investigating the use of a VMM with LogTM-SE, we identified several shortcomings of its design. Two problems arise when virtualizing transactions on LogTM-SE within a virtual machine: lack of physical addresses in the OS and lack of knowledge of threads in the VMM. These problems may arise in other virtual HTM systems where the OS must manipulate physical addresses or suspended threads.

Physical Addresses. A guest OS in a virtual machine cannot correctly virtualize transactions because it lacks access to physical addresses. In a virtual machine environment, the VMM presents *real* addresses to the guest operating system and internally maps them onto *physical* addresses [30]. However, signatures operate on physical addresses. As a result, the guest OS cannot compute summary signatures after the VMM remaps a page because it knows only real addresses and not the new physical addresses of pages.

Threads. A related problem arises when the VMM tries to virtualize transactions. Summary signatures are computed from the signatures of all suspended threads in a process. However, threads are a software construct not visible at an architectural level. As a result, a VMM may be unaware of threads or thread context switching. For example, the VMM does not know when the OS migrates a thread between processors and hence cannot update summary signatures appropriately.

4. LogTM-VSE Hardware Extensions

We address the shortcomings of LogTM-SE in a virtual machine environment with *LogTM-Virtual Signature Edition (LogTM-VSE)*. This design maintains the physical signature from LogTM-SE for conflict detection on coherence requests. However, it extends LogTM-SE with three features to address the problems raised previously: virtual signatures, virtual summary signatures, and virtualization traps. LogTM-VSE also extends LogTM-SE with several performance enhancements. Figure 1(b) shows LogTM-VSE's additional processor state.

4.1 Virtual Machine Extensions

LogTM-VSE adds a *virtual signature* to export virtual addresses to software. On every load and store in a transaction, the processor

adds the virtual address into the virtual signature. Unlike the physical signature, it is not checked on coherence requests. However, it may be saved and restored by the OS. LogTM-VSE also changes the existing summary signature to a *virtual summary signature* on virtual addresses. Similar to LogTM-SE, LogTM-VSE checks the virtual summary signature for conflicts on every load and store instruction. Switching the summary signature to virtual addresses allows the OS to ensure that conflicts are detected correctly when pages are remapped without knowing the new physical address of the page (Section 5.2). LogTM-VSE cannot currently handle synonyms (different virtual addresses that point to the same physical address) because of its use of virtual addresses in signatures. In addition, LogTM-VSE does not support shared memory between processes. These restrictions are common to many TM systems that address virtualization.

LogTM-VSE provides *virtualization traps* that notify the VMM when it and the OS simultaneously virtualize transactions. The traps are enabled by the `TxVmmVirt` flag, When set, this flag causes the summary signature and physical signature to be treated as hyper-privileged registers that trap into the VMM when written. These traps notify the VMM when an OS action, such as thread suspension or migration, affects its virtualization of transactions. As we describe in the Section 6, virtualization traps provide the VMM with enough information about threads for it to virtualize transactions.

4.2 Performance Extensions

LogTM-VSE includes three extensions that decrease the cost of virtualizing transactions. First, LogTM-VSE removes the need for synchronous communication with other processors when managing summary signatures with *lazy summary update*. In LogTM-SE, summary signatures of all running threads in a process must be updated synchronously during a context switch, by interrupting all other processors in the system. With lazy summary update, a thread defers updating its summary until it requests data from the cache of the processor that ran the virtualized transaction. To detect this, each processor has a `TxForceUpdate` register containing a bit map of the processors in the system. On a context switch, system software sets all bits in this register before allowing a new thread to run. We rely on the invariant that the directory coherence protocol forwards memory requests to processors that last accessed the block. When a coherence request subsequently arrives from a processor with its bit set, the receiving processor clears the bit for the requesting processor and responds with a NACK message. The NACK indicates that the receiving processor's physical signature does not protect all the blocks in its cache. This response causes the requester to trap so it can reload its summary signature before reissuing the request.

A single bit map register per processor cannot provide performance isolation between processes because a process may receive a NACK reload message due to virtualization of a transaction in another process. This could be addressed with additional bit maps, tagged by address space, or by disabling lazy summary update.

Second, LogTM-VSE replaces double bit-select as a hash function with H_3 for both physical and virtual signatures, which has fewer false positives [27]. This is particularly important for virtualized transactions, whose long duration raises the potential for conflicts.

Third, LogTM-VSE provides a hardware flag to indicate that the current transaction has been virtualized. The `TxVirtualized` flag, set by software, causes a trap when the running transaction completes. In response, system software can cleanup state related to the transaction, such as summary signatures. In contrast, LogTM-SE has no mechanism for detecting when a virtualized transaction completes.

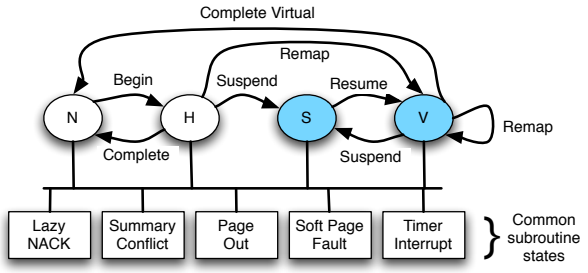

Figure 2. Thread state machine for virtualizing transactions within an OS. Arcs are labeled with events that cause a state transition. Both aborts and commits complete a transaction, and a suspended transaction must resume before it can abort. Virtualized states are shaded. The subroutines shown along the bottom may be entered from any of the states above and return back to that state.

5. Virtualizing Transactions

In this section we discuss the operations of the Transaction Virtualization Manager (TVM), a new kernel module that implements the software support for virtualizing transactional memory. It ensures that transactional semantics are enforced by the hardware when the OS reclaims a processor or memory page from a process. However, it does not currently address I/O in transactions, which require additional changes to the OS. We implement TVM in OpenSolaris [29]. Hooks in the kernel invoke TVM when *virtualization events* (*e.g.*, context switches) occur that force a transaction to virtualize or affect a previously virtualized transaction.

A thread may be in one of four transactional states:

- *N:* The thread is *non*-transactional.
- *H:* The thread is executing a *hardware* transaction that has not been virtualized.
- *S*: The thread has been *suspended* while executing a transaction.
- *V*: The thread is executing a transaction that has been *virtualized*.

The transitions between these states are shown in Figure 2. We include suspended non-transactional threads in this N state, as they are treated identically by TVM. There are two non-virtualized states, N and H. In the common case where transactions are not virtualized, threads move between the N and H state without OS involvement. When the OS causes a thread to virtualize, indicated by the *Remap* and *Suspend* events, threads move into one of the two virtualized states, S and V. We term threads in these states *virtualized threads*. In the S state, isolation for the thread is completely provided by summary signatures. In the V state, isolation is maintained by a combination of physical and summary signatures. A transaction stays in the S and V states until it completes, by committing or aborting.

In addition to the events that transition a thread between transaction states, TVM executes in response to other OS events and traps. Along the bottom of Figure 2 are shown five events that invoke TVM but leave the thread in the same state. These events are optimizations that reduce the virtualization overhead and we describe them later.

We now discuss in detail how TVM virtualizes transactions in response to context switching and paging.

5.1 Context Switching Transactional Threads

Thread and process context switching present two related challenges. First, conflict detection must continue while a transaction is

```
tvm_context_switch_out(Thread T) {
  merge_sig(T->save_sig,get_sig());
  T->process->last_virt_time = current_time;
  set_TxForceUpdate();
}

tvm_complete_virt_tx(Thread T) {
  clear_sig(T->save_sig);
  T->process->last_virt_time = current_time;
}

tvm_reload_summary(Thread T) {
  if (T->summary_time <
      T->process->last_virt_time) {
    foreach (Thread U in T->process,U != T)
      merge_sig(T->summary,U->save_sig)
    T->summary_time =
      T->process->last_virt_time;
  }
  reload_summary(T->summary);
}
```

Figure 3. Pseudocode for salient functions that calculate summary signatures on thread context switches.

suspended to ensure that other threads cannot read addresses written or write addresses accessed by a suspended transaction. Thus, saving and restoring signatures on a context switch is not sufficient to ensure isolation. Second, if the kernel migrates a thread between processors, a directory coherence protocol will not direct requests to the new processor; they will go to the old processor where the data is in the cache. As a result, we cannot rely on all coherence protocols to ensure that conflicts are checked with a migrated transaction.

Our approach to this problem is to ensure that when memory cannot be isolated by a physical signature on a specific processor, it is isolated by summary signatures on other processors. A LogTM-VSE processor checks its own summary signature for conflicts on every memory instruction, which relieves the coherence protocol of directing the request to the correct signature. The goal of TVM is to ensure that data referenced by a transaction is isolated by summary signatures at other processors before TVM allows a physical signature to be reused by a different thread.

5.1.1 Context Switching Algorithms

TVM manages summary signatures to ensure continued isolation after the OS suspends a transaction. It saves the transaction state and updates other thread's summary signatures. When the transaction completes (commits or aborts), it updates summary signatures to drop the virtualized transaction's contribution.

We added code to the OpenSolaris kernel in five places to notify TVM of events related to context switching: thread suspension, thread resumption, virtual transaction completion (commit or abort), summary signature conflict, and a lazy NACK. Next, we describe the specific actions taken by TVM when these events occur.

Suspend Thread. When the kernel suspends a transactional thread, TVM adds the thread to its list of virtualized threads. TVM saves the thread's transactional state and merges its signature with a previously saved signature, if it exists. TVM also updates a per-process timestamp, `last_virt_time`, that records the time of the latest virtualization event. In LogTM-SE, the OS had to synchronously notify other processors to reload their summary signatures. With *lazy summary update*, though, TVM relaxes the goal of protecting data either with a physical or a summary signature. Instead the `TxForceUpdate` register provides isolation. TVM loads the `TxForceUpdate` register before reusing the physical signature. This forces other processors to reload their summary signatures before accessing any data that was isolated by the in the processor's physical signature.

Figure 3 shows pseudocode for the `tvm_context_switch_out` function that TVM executes when context switching a thread. It simply saves its virtual signature and updates a process-wide variable to the current time. Later, when other threads learn they need to update their summary signatures (described below), they execute `tvm_reload_summary`.

We use OpenSolaris' existing `installctx` method to receive callbacks when the kernel calls the `savectx` and `restorectx` context switch routines. TVM augments the kernel's thread data structure with fields to store transaction state, shown in Figure 1(a), which includes a register checkpoint, log pointer, handler addresses, flags, and the signature. TVM also maintain a list of the virtualized threads in a process.

Resume Thread. Our hooks invoke TVM when the kernel executes `restorectx` to reschedule a thread. For all threads, TVM calls `tvm_reload_summary` to calculate and load a new summary signature if necessary. For transactional threads, TVM reloads the saved transaction state except for the physical signature; summary signatures at other threads subsume the signature's function. TVM also sets the `TxVirtualized` flag to cause a trap when the virtualized transaction completes.

Complete Virtual Transaction. When a virtualized transaction completes (commits or aborts), TVM must calculate new summaries without the completed transaction. The previously described `TxVirtualized` flag causes the processor to a trap when a transaction in the V state completes. We install a handler for this trap that notifies TVM to invoke the `tvm_complete_virt_tx` routine in Figure 3, which clears the thread's saved signature and updates the timestamp of the last virtualization event. To avoid synchronous communication, other threads defer reloading their summary signatures until they conflict with it.

Summary Signature Conflict. When a thread conflicts with its summary signature, LogTM-VSE traps to a handler in TVM to resolve the conflict. TVM detects whether the thread's summary is out of date, reloads the summary if necessary, and restarts the thread. The `tvm_reload_summary` function detects whether a new summary is needed and if so, calculates one. For simplicity, we leave out optimizations, such as reducing the number of computations by treating all non-transactional threads identically.

If the thread's summary is already up to date, then the trap is passed to a user-mode contention manager to resolve the conflict. The contention manager may abort the current transaction, signal another transaction to abort, or queue the current transaction behind another [33].

Lazy NACK. When a processor first requests data from another processor that recently virtualized a transaction, it will receive a NACK due to lazy summary update. This NACK, which indicates that it must reload its summary signature before proceeding, causes a trap. TVM executes `tvm_reload_summary` and restarts the thread, allowing the memory request to proceed.

5.1.2 Context Switching Example

Figure 4 illustrates how LogTM-VSE supports context switching. In the initial state on the left, all three processors are executing transactions in process P1. When the kernel preempts the thread on CPU 0 and schedules a thread in process P2, the system virtualizes transaction T1 by adding its signature to the summary signature on CPU 1 and 2. As a result, these processors continue to correctly detect conflicts with T1.

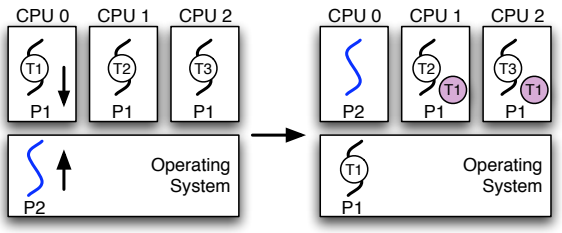

Figure 4. Context switching on LogTM-VSE with TVM.

5.2 Paging Transactional Data

Three challenges arise when the kernel removes a page from memory and reloads it at a different physical address. First, LogTM-VSE's physical signatures cannot detect conflicts at the new physical address. Coherence protocols send only physical addresses, and sending virtual addresses as well would add 100% overhead to some coherence messages. Second, directory coherence protocols send messages only to the processors with the block in their cache. The new physical page, having just been transferred from disk, will be in main memory and not in any cache. Hence, the coherence protocol may not direct requests for the new page to the appropriate processor. Third, LogTM-VSE does not mark which pages were accessed transactionally, so the system must use other mechanisms to identify whether a page contains transactional data.

Remapping of transactional pages should be a rare event. Only when the OS unmaps a page and remaps it, both *while the same transaction is running without virtualizing*, is action required. The average time to service a page fault on our Sparc T1000 system (see Section 7) is 4.6 ms as measured by lmbench [19]. This is much longer than most transactions, and hence transactions will rarely experience both the unmap and remap of a page. Our design therefore strives to have low overhead for detecting transactional paging events and is less concerned with the cost of handling them once detected.

Our approach to paging transaction data is to virtualize all transactions that *may have* accessed a remapped page, so that virtual addresses in summary signatures isolate the page's data. TVM conservatively detects whether pages contain transactional data based on timestamps: a page may contain transactional data if a transaction that began *before* the page was removed is *still* active when the page is brought back in.

5.2.1 Paging Algorithms

TVM provides two functions for handling page remapping. First, it continuously tracks page and transaction timestamps to identify which transactions may have accessed the page before it was unmapped. Second, in response to a page remap event, TVM virtualizes the transactions that may have accessed the page before it was remapped. These signatures contain the virtual addresses of previously referenced blocks.

TVM's code for handling paging executes in response to six events: a page out, a page remap, a soft page fault, a timer interrupt, a virtualized transaction completion, and summary signature conflict. TVM's actions for the last two events are identical to its actions for these events following a context switch. We first describe TVM's actions to identify which transactions may have accessed a page, and then describe how it isolates data on the page.

Page out. TVM tracks page timestamps by hooking kernel code that invalidates or unmaps a page from a process. We added code to the `hat_pageunload` routine in OpenSolaris to notify TVM when the kernel unmaps a page. TVM records the current time (also used by LogTM-VSE for conflict resolution) in the kernel's address space structure.

```
tvm_page_remap(Page P) {
  if ((P->phys == P->old_phys) ||
      (P->inval_time < tvm_oldest_tx_time))
    return;
  foreach (Thread T in Process) {
    if (T.timestamp > P->inval_time)
      continue;
    if (T->running)
       cross_call(T,tvm_force_virtualize);
  }
  foreach (Thread T in Process) {
    tvm_recalc_summary(T);
  }
}
```

Figure 5. Pseudocode for salient functions to update summary signatures when the kernel remaps a page.

Timer Interrupt. TVM records thread timestamps on every timer interrupt. To resolve conflicts, LogTM-VSE records a timestamp when transactions begin. We hook the timer interrupt routine to call into TVM, which records the transaction timestamp of the currently executing thread. If the thread is not in a transaction, we record a timestamp of infinity, indicating that it is "after" all other events.

Page Remap. TVM must isolate data on a remapped page before a thread accesses it. Only active transactions must be considered, as suspended transactions already protect data with virtual addresses. If TVM detects that a running thread may have accessed a page, it issues a cross call to the thread's processor. In this case, TVM virtualizes (or re-virtualizes) the transaction with the `tvm_force_virtualize` function (not shown), which saves the transaction's signature and setting the `TxVirtualized` flag to force a trap when the transaction completes. We add code to the hard page fault path of the `anon_getpage` routine to notify TVM when the kernel maps a page into a process.

Figure 5 shows pseudocode for the `tvm_page_remap` routine that handles a page remap event. For simplicity, the code does not reflect optimizations, such as walking the list of threads in timestamp order to avoid touching threads that could not have accessed the page. Not shown is code for managing timestamp skew between processors, which considers timestamps to overlap if they are within the maximum skew.

Soft Page Fault. TVM does not update summary signatures synchronously for all threads; virtual memory hardware protects the data on the page until the page mapping is entered in a processor's TLB. TVM therefore delays updating the summary signature on threads other than the one accessing the remapped page until they access the page. We add code to the soft page path of `anon_getpage`, which occurs when a processor faults on page that is already in memory. In response, TVM invokes `tvm_reload_summary`.

5.2.2 Paging Example

Figure 6 shows how TVM can detect whether a page may have transactional data from thread and page timestamps. When the kernel remaps page 1 at T=15, TVM finds that the thread timestamps (8,6,7) are newer than page 1's invalidation timestamp (5). Hence, they could not possible have accessed the page. When the kernel remaps page 2 at T=20, TVM detects that thread 3's transaction began at time 10, while page 2 was invalided at time 12. Hence, page 2 may still have transactional data from thread 3, which must be virtualized.

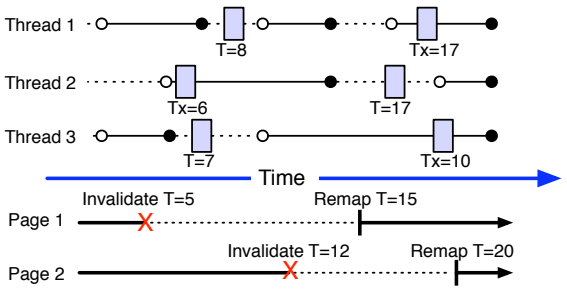

Figure 6. Example of how TVM determines whether a page may have transactional data. The lines indicate when transactions begin and end and when pages are invalided (swapped out) and remapped (swapped in). The shaded boxes indicate when TVM records the transaction timestamp, which is shown below the thread. Timestamps labeled Tx indicate the start time of an active transaction.

5.3 Summary

In summary, the Transaction Virtualization Manager virtualizes transactions by isolating data with summary signatures when the processor cannot isolate it with physical signatures.

TVM maintains an invariant on what *must* be in a summary. The summary signature for a running thread T in process P contains the union of:

1. The read and write sets of all other virtualized threads from P at the time at their last suspension.
2. For all pages R remapped by the OS, for all other threads U from P in transactions that began *before* R was unmapped, U's read and write sets at any time *after* R was unmapped.

This is relaxed with lazy summary update, when invariant 1 is only enforced after attempting to access data previously accessed by a virtualized thread. TVM may also leave more addresses in the summary signature to avoid the cost of reloading it.

6. Virtualizing TM in a Virtual Machine

LogTM-VSE supports virtualizing transactions both within an OS and within a VMM. This support is necessary to allow a VMM to context switch processors and swap memory without either contacting the OS to provide virtualization, or accessing internal OS data structures. While it may be possible to force transactions to abort instead, doing so makes VMM virtualization activities visible to the OS.

We designed a software architecture for simultaneous virtualization by both the VMM and a guest OS based on two principles:

1. The VMM should only be involved when it causes a transaction to virtualize.
2. The interface and interactions between the OS and VMM should be minimal.

These principles recognize the goal of keeping VMMs small and simple and let the OS handle the majority of virtualization events.

We duplicate TVM in both the OS and the VMM. TVM$_{OS}$ (TVM in the OS kernel) manages virtualization of *threads* due to OS events, and may execute either on bare hardware or in a virtual machine. TVM$_{VMM}$ manages virtualization of *virtual processors* due to VMM events and steps aside when the OS re-virtualizes the same transaction. Both versions of TVM compute separate summary signatures to virtualize transactions, which are merged together by TVM$_{VMM}$.

The operations of TVM are straightforward when one of the OS or the VMM virtualizes a transaction. When both do so, TVM$_{VMM}$ must coordinate its actions with TVM$_{OS}$. We next consider the four cases of virtualization: (1) in the OS, (2) in the VMM, (3) in the VMM and then the OS, and (4) in the OS and then the VMM. While both paging and context switching cause virtualization, we describe TVM's actions in terms of just context switching.

Virtualization by the OS only. Within the kernel, TVM$_{OS}$ executes as in Section 5.

Virtualization by the VMM only. Within the VMM, TVM$_{VMM}$ virtualizes transactions in response to virtualization events from the VMM, such as when it reallocates a physical processor or memory pages between virtual machines. It uses the same algorithms described in the previous section, but with virtual processors instead of threads.

TVM$_{VMM}$ must know which processors are in the same process so it can update summary signatures. We borrow mechanisms from Antfarm [11] to detect when process switches occur through changes to the previously described address space identifier (ASID).

Virtualization by the VMM then OS. This may occur when the VMM resumes a virtual processor running a thread that the OS then context switches. After TVM$_{VMM}$ virtualizes a transaction it sets the `TxVmmVirt` flag on every processor in the process. This flag causes two additional events to trap into the VMM: clearing a physical signature and updating a summary signature. When TVM$_{OS}$ writes to a summary signature, LogTM-VSE traps so that TVM$_{VMM}$ can merge the new OS summary with its summary and load their union. TVM$_{VMM}$ forwards `TxVirtualized` traps to TVM$_{OS}$ after clearing its own state.

Physical signatures are cleared when a transaction commits or when the OS virtualizes a transaction; at this point, the OS is virtualizing the transaction so TVM$_{VMM}$ need not.

Virtualization by the OS then VMM. This may occur when the VMM preempts a virtual processor after the OS resumes a thread in a transaction on that processor. TVM$_{VMM}$ must only virtualize the state of the thread since it was last scheduled (*i.e.*, provide isolation for data accessed since the thread was resumed), as the OS virtualizes its prior state. Thus, TVM$_{VMM}$ merges the thread's current virtual signature into the summary signatures of other processors in the process and sets the `TxVmmVirt` flag. Again, TVM$_{VMM}$ stops virtualizing when the transaction completes or is virtualized by the TVM$_{OS}$ again.

Summary. This architecture allows the VMM to stay uninvolved in the common case, when it is not reclaiming hardware resources, and to virtualize transactions without OS involvement when necessary.

7. Evaluation

We measure two aspects of our system:

1. *Utility*: When is virtualization likely to be helpful, and how often?
2. *Cost*: What is the performance cost of virtualizing transactions, and why? This is the cost of the OS actions to manage summary signatures.

Our evaluation focuses on the importance and overhead of virtualization and hence does not evaluate possible performance gains from TM.

The few programs currently written to use transactions are applications developed for systems that do not support virtualization or micro-benchmarks that are unlikely to need virtualization. We therefore measure a variety of lock-based multithreaded programs assuming that every lock is converted to a transaction.

Application			Context Switches			Page Fault	
Name	Workload	Threads	Voluntary	Involuntary	Delay μs	Rate	Locked Rate
BIND 9.0	DNSPerf/Update	35	1,494	26	48	0	0
AOLserver	ApacheBench	14	0.1	2	113	0	0
Apache 2.0	SpecWeb99	66	555	0.5	-	0.5	0
Firefox	Browsing	5	12	1.5	-	0.23	0.007
OpenOffice Writer	Editing	5	0.1	2.0	-	0.25	0.13

Table 1. This table lists application names, workload, and workload characteristics. Rates are per second. We present the average of three runs.

7.1 Measurement Platforms

We use multithreaded chip multiprocessor to measure the behavior of multithreaded applications and a simulator to measure the cost of virtualizing transactions. These two platforms allow us to measure long-term application behavior as well as fine-grained virtualization events.

Niagara. We record the frequency of virtualization events on Sun T1000 (Niagara) hardware. The T1000 system has a 1 GHz, 8-core, 32-thread T1 processor and 16 GB of RAM. To increase the frequency of paging, we restrict applications to 256 MB of memory. We rely on DTrace [18] to detect context switches, page faults, and system calls while a thread holds a mutex lock. Our assumption that all mutex locks could be converted to transactions is optimistic for LogTM-VSE because critical sections that contain I/O system calls cannot be converted to transactions automatically.

GEMS. Our Wisconsin GEMS-based simulator [16] models a 32-processor Sparc chip-multiprocessor with a single-issue in-order pipeline and memory latencies similar to the Sun T1 (Niagara) processor [12]. We assume a 1GHz clock rate. While the LogTM-SE paper evaluated 512-byte signatures [32], we evaluate 2,048-byte signatures that may be necessary to avoid false conflicts on long transactions.

Workloads. We selected two classes of multithreaded applications: server and desktop. The server applications consist of the BIND 9.0 DNS name resolver, the AOLserver 4.5.0 web server, and the Apache 2.0 web server. For client applications, we selected Firefox 2.0 and OpenOffice Writer 2.3.0.

7.2 Utility: Opportunities for Virtualization

We measure the frequency of paging and context switching with locks held to determine whether virtualizing transactions is necessary. We run the applications listed in Table 1 on the Sun T1000 platform and measure the frequency of critical sections that would be impacted by an OS virtualization event.

Columns 4-6 in Table 1 show the frequency of voluntary context switches, involuntary context switches, and how far into critical sections, in μs, context switches occur (labeled delay). Voluntary context switches occur when the thread relinquishes the processor voluntarily through an explicit yield or blocking system calls. Involuntary context switches occur when the the scheduler preempts a thread.

These measurements have two implications. First, voluntary context switches *caused by the transaction* are common in these programs. BIND yields the processor within a critical section 1,494 times per second. Aborting transactions with voluntary context switches may prevent these programs from running correctly. We further analyzed BIND and AOLserver to determine which system calls caused these context switches. These calls break down into three categories: communication (`recvmsg` and `doorfs`), file I/O (`read` and `write`) and locking (`lwp_park` and `yield`).

While our system does not currently support communication or I/O in transactions, recent work points towards possible solutions [3] that require virtualization support. Calls to `lwp_park` and `yield` occur when blocking on a lock, which could occur when transactions interact with legacy code that still uses locks. This again requires virtualization support to avoid busy waiting [31, 33].

Second, involuntary context switches may be efficiently handled by aborting the running transaction. The time to abort in LogTM-VSE is proportional the length of the transaction but generally much faster. Involuntary context switches occur, on average, 77 μs into critical sections in BIND and 113 μs into critical sections in AOLserver, so virtualization will improve performance only if it takes substantially less time.

Despite reducing memory to 256 MB, paging is infrequent for these workloads and only occurred a few times with locks held. Thus, it may not be important to handle paging quickly, as long as detecting when paging impacts transactions is efficient. TVM's use of timestamps fits this criteria.

These results demonstrate that (1) programs often perform blocking actions in critical sections and virtualization may be required to convert these critical sections to transactions with LogTM-VSE and (2) virtualizing must be fast to perform better than aborting for involuntary context switches.

7.3 Utility: Necessity of Virtualizing

In LogTM-VSE, it is possible for abort to take an arbitrarily long time. The old contents of memory overwritten by the transaction are stored in a log that must be restored in software on abort. The time to perform an abort is determined by the length of the log and the time taken to restore old values. For long-running transactions, aborting may take much longer than virtualizing. To demonstrate this, we wrote the `TLB-stress` program that accesses blocks on 2048 different pages. Our simulated hardware has a 512 entry TLB. When we force the transaction to abort just before completing, abort processing takes over 1 ms (1,000,000 cycles). This is due to TLB misses encountered while restoring data to memory. While this is a contrived example, a system that aborts running transactions before yielding the processor to a higher-priority process could suffer reduced responsiveness without the ability to virtualize transactions.

7.4 Cost: Microbenchmarks

We measure the cost of virtualizing transactions with micro-benchmarks that exercise the virtualization code. We measure results on GEMS with and without lazy summary update.

The `CS-stress` micro-benchmark measures context-switch processing time. It spawns 64 threads, 32 of which execute transactions that call `sched_yield` to force context switches. We measure the number of context switches and the time spent in TVM processing those switches.

The `PR-stress` micro-benchmark measures the page remap time. The program allocates a global array of pages that 31 threads access in transactions while one thread forces pages out with the `memcntl` system call. The transactional threads fault when a page they access has been swapped out. We measure the number of page faults and the time spent in TVM processing page remapping.

Program	Threads	μs Latency	μs Overhead
CS-stress/Lazy	1 / 25	9.7 / 12	151 / 269
CS-stress/Eager	1 / 19	379 / 651	642 / 4,480
PR-stress	1 / 28	1,850 / 5,240	3,900 / 13,363

Table 2. Micro-benchmark results showing overhead and latency of virtualization with and without and lazy summary update.

The average number of virtualized threads in these tests is artificially high; half or more are virtualized at any given time. In our application workloads, on average only one thread holds any lock at a time. We therefore run the same tests with only a single thread executing transactions to measure this scenario.

Table 2 shows, for both microbenchmarks, the average number of virtualized threads, the average latency to handle a virtualization event, and the average total time (overhead) consumed across all threads for a virtualization event. For context switch, latency represents the time to deschedule a transaction. Overhead includes the time to deschedule a transaction and the time taken by all other threads to process the cross-call and reload their summary signature. For paging, latency represents the time to process a hard fault that results on a transactional page remapping. Overhead includes the time to remap the page and the time taken by all other threads to process their subsequent soft page faults and reload their summary signature. We show the context switching results both with lazy summary update (CS-stress/Lazy) and without (CS-stress/Eager). Without lazy update, the thread that is context switching computes new summary signatures for all other threads. It notifies the other threads to reload their summaries with an inter-processor interrupt. For each test, we show the *single* virtualized thread case and the *many* thread case, where TVM virtualizes many transactions simultaneously.

The latency results demonstrate the benefit of lazy summary update. Without this optimization, context switching a transaction can take up to 651 μs, where as with it, only 12 μs are needed. In addition, lazy summary update reduces the total overhead because some summary calculations are not needed.

The overhead results demonstrate that a large portion of TVM's time is spent manipulating signatures. This is visible from the difference between the one and many thread case. Computing summary signatures scales quadratically with the number of virtualized threads, as each thread's summary is comprised of the signatures of all other threads.

These results indicate that the overhead of a context switch or page remap is non-trivial and likely greater than the time to abort a transaction. However, there are many opportunities, both in hardware and software, to further optimize this system. These include better algorithms for manipulating summary signature structures, block instructions that load more than 64 bits at a time, and additional hardware support. In many cases, though, abort may be a better option if possible, as context switches occur on average 77-100 μs into the transaction, much less than the total overhead of virtualizing.

7.5 Cost: Overhead of Virtualization

From the data in Table 1 and the measurements in Table 2, we can estimate the performance cost of virtualizing transactions for our workloads. The cost is the product of the frequency of voluntary context switches and the time to perform a context switch. From our measurements, we estimate that the overhead of virtualizing transactions for BIND, the program with the most frequent context switches, is between 2% and 7% with lazy summary update, depending on the number of threads virtualized at once. For Apache, it is between 0.25% and 1%, and for the other programs it is even smaller. We also calculated that the overhead of 10 page faults per second requiring virtualization (a small subset of all page faults) is between 1% and 2.5%, depending on the number of threads virtualized. These results demonstrate that, because of low frequency of virtualization events, the total cost to virtualize transactions with LogTM-VSE and TVM is low. Furthermore, our study of application behavior indicated that blocking within a transaction is common and therefore useful to programmers.

8. Related Work

There have been several approaches proposed to virtualize transactional memory. We differ from other systems in our single execution mode, graceful support for cache victimization, implementation in an OS, and support for VMMs.

Several systems execute transactions in two modes; one for small, unvirtualized transactions and another for virtualized transactions. Blundell et al. present a simple mechanism for virtualizing transactions, but it supports only a single virtualized transaction at a time and requires extra tags in memory that the OS that must save and restore when paging [5]. Hybrid systems accelerate short transactions with hardware and fall back on pure software when necessary, for example on a virtualization event [8, 9, 13, 14, 26, 28]. While this speeds small transactions, it extends the execution time of long transactions by executing them with software support. In contrast, virtualized transactions on LogTM-VSE execute at the same speed as non-virtualized transactions.

Many other hardware transactional memory systems that support virtualization require software action when the processor evicts transactional data from the cache [1, 7, 8, 23]. Evictions can occur either when the cache is full, or even when a single *set*, often only 4 blocks, is full. This makes virtualization far more frequent. In addition, executing code in response to a cache eviction is difficult, because the handler code must avoid further evictions. In contrast, LogTM-VSE only virtualizes on context switches and page remappings, which are far less frequent, and the software runs with no restraints on cache access.

Other TM systems advocate signatures for virtualization, but do not provide an OS (or VMM) implementation. Similar to LogTM-SE, the SigTM hybrid relies on a physical summary signature for virtualization [20]. Bulk proposed using signatures for TM conflict detection and provided extra signatures at a processor to track suspended threads [6]. It checks signatures in memory when these are exhausted. This system depends on a broadcast protocol and does not yet address paging. In contrast, LogTM-VSE summary signatures are always available, allow use of non-broadcast coherence protocols, and can be updated in software when the OS remaps a page. LogTM-VSE's summary signature owes an intellectual heritage to VTM's transaction filter (XF) [23]—as both over-approximate virtualized read- and write-sets—but summary signatures are never modified by hardware or micro-code. None of these previous systems address VMM issues.

Other systems explore the interaction of transactional memory and the OS. MetaTM supports transaction use *within* a kernel and provides a mechanism for interrupt handlers to use transactions [24, 25]. This complements our work on virtualizing user-mode transaction. Zilles explores coupling OS and transaction scheduling in the context of VTM [33], which could be adapted for resolving conflicts with summary signatures in LogTM-VSE.

9. Conclusion

Hardware transactional memory promises to simplify multi-threaded programming, which will be necessary to take advantage of future processors. To provide this benefit, though, it must support virtualizing transactions when the OS virtualizes the hardware. The contributions of this paper are three-fold. First, we identify shortcomings of LogTM-VSE in the presence of virtual machines.

Second, we describe an implementation of the Transaction Virtualization Manager (TVM), a software architecture for virtualizing TM on within OpenSolaris. Third, we evaluate multithreaded applications and demonstrate that virtualization is useful for interacting with locked-based code and for blocking system calls. We show with a combination of application profiles and simulator measurements that LogTM-VSE and TVM incur little overhead to virtualize transactions.

Acknowledgments

This work is supported in part by NSF grant CNS-0720565. We would like to thank Jayaram Bobba and Mike Marty for input on this paper. Hill and Wood have significant financial interest in Sun Microsystems.

References

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

[2] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM Transactional Application Programming Interface. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2007.

[3] L. Baugh and C. Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *Proceedings of the Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Aug. 2007.

[4] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] C. Blundell, J. Devietti, E. C. Lewis, and M. M. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[6] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33nd Annual International Symposium on Computer Architecture*, June 2006.

[7] W. Chuang, S. Narayanasmy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, O. Colavin, and B. Calder. Unbounded Page-Based Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[8] J. Chung, C. C. Minh, A. McDonald, H. Chafi, B. D. Carlstrom, T. Skare, C. Kozyrakis, and K. Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchango, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[10] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. Technical Report Technical Report 92/07, Digital Cambridge Research Lab, 1992.

[11] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the 2006 USENIX Annual Technical Conference*, June 2006.

[12] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[13] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Mar. 2006.

[14] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Aug. 2007.

[15] P. S. Magnusson et al. SimICS/sun4m: A Virtual Workstation. In *Proceedings of Usenix Annual Technical Conference*, June 1998.

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

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

[18] R. McDougall, J. Mauro, and B. Gregg. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Pearson Professional, 2006.

[19] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, Jan. 1996.

[20] C. C. Minh, M. Trautmann, J. Chung, A. Mcdonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

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

[22] 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 *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

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

[24] 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 *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[25] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Proceedings of the 21st ACM Symposium on Operating System Principles*, Oct. 2007.

[26] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Mar. 2006.

[27] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007.

[28] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. S. III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[29] Sun Microsystems. OpenSolaris. `http://www.opensolaris.org/os/`.

[30] Sun Microsystems. UltraSPARC Virtual Machine Specification, Revision 1.0. `http://opensparc-t1.sunsource.net/specs/Hypervisor-api-current-draft.pdf`, 2006.

[31] H. Volos, N. Goyal, and M. M. Swift. Pathological Interaction of Locks with Transactional Memory. In *Proceedings of the Third ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Feb. 2008.

[32] L. Yen, J. Bobba, M. R. 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 *Proceedings of the Thirteenth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2007.

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