Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions

Craig Zilles Lee Baugh
Computer Science Department
University of Illinois at Urbana-Champaign
[zilles,leebaugh]@cs.uiuc.edu

ABSTRACT

Transactional Memory (TM) is a compelling alternative to locks as a general-purpose concurrency control mechanism, but it is yet unclear whether TM should be implemented as a software or hardware construct. While hardware approaches offer higher performance and can be used in conjunction with legacy languages/code, software approaches are more flexible and currently offer more functionality. In this paper, we try to bridge, in part, the functionality gap between software and hardware TMs by demonstrating how two software TM ideas can be adapted to work in a hardware TM system. Specifically, we demonstrate: 1) a process to efficiently support transaction waiting — both intentional waiting and waiting for a conflicting transaction to complete — by de-scheduling the transacting thread, and 2) the concept of pausing and an implementation of compensation to allow non-idempotent system calls, I/O, and access to high contention data within a long-running transaction. Both mechanisms can be implemented with minimal extensions to an existing hardware TM proposal.

1. INTRODUCTION

While the industry-wide shift to multi-core processors provides an effective way to exploit increasing transistor density, it introduces a serious programming challenge into the mainstream; even expert programmers find it difficult to write reliable, high-performance parallel programs, with much of this difficulty resulting from the available primitives for managing concurrency. The problems with locks, presently the dominant primitive for managing concurrency, are well documented (*e.g.*, [24]): they don't compose, they have a possibility for deadlock, they rely on programmer convention, and they represent a trade-off between simplicity and concurrency.

Transactional Memory (TM) [1, 8, 9, 10, 11, **?**, 18, 22] has been identified as a promising alternative approach for managing concurrency. TM addresses a number of the problems with locks by providing an efficient implementation of *atomic blocks* [15], code regions that must (appear to) not be interleaved with other execution. Atomic blocks, or *transactions* as the recent literature calls them, simplify concurrent programming because, while the programmer must still identify *critical sections* (where shared state is not consistent), they need not be associated with any synchronization variable. By using an optimistic approach to concurrency (*i.e.*, speculate independence and rollback on a conflict), concurrency need only be limited by data dependences, leading to even better performance than fine-grain locking in some cases.

Since the introduction of Transactional Memory, development of TM systems has gone in two distinct directions. First, researchers have explored to what degree transactional memory can be implemented efficiently without hardware support. In this process, these software transactional memory (STM) systems have been extended to support additional software primitives, further increasing the power of the programming model. Concurrently, research in hardware transactional memory (HTM) has yielded approaches that avoid exposing hardware implementation details (*e.g.*, cache size, associativity) to the programmer, but generally without extending the programming model.

In this paper, we show that a number of the extensions developed in the context of STMs can be incorporated into HTMs, and that doing so can be inexpensive, in that it does not require significant extensions to existing HTM proposals. In this paper, we focus on the Virtual Transactional Memory (VTM) proposal from Rajwar *et al.* [22]. We provide background about VTM in Section 2, discussing its salient features and how our implementation differs from its original proposal.

We focus on incorporating two STM features. First, in Section 3, we show how an HTM can cooperate with a software thread scheduler to avoid having transactions busy-wait for long periods of time. This has two applications: 1) stalling one transaction while it waits for a conflicting transaction to commit, and 2) using transactions to intentionally wait on multiple variables, much in the manner of the Unix system call `select()`. We find that the additional required hardware support is limited to raising exceptions to transfer control to software under certain transaction conflicts.

Second, we demonstrate how support for non-transactional actions can be included within transactions (Section 4). This too has two main applications: 1) avoiding contention resulting from accessing frequently modified variables within a long transaction, and 2) performing I/O or system calls in the middle of transactions. The only required hardware extension is the ability to *pause* a transaction without pausing the thread's execution, which requires an additional mode for transactions and two new primitives for pausing and unpausing. With transactional pause in place, we demonstrate how a non-idempotent system call, `mmap()`, can be supported in a hardware transaction using a software-only framework for compensating actions.

In Section 5, we discuss concurrent work to extend HTM's with more STM-like features before concluding in Section 6.

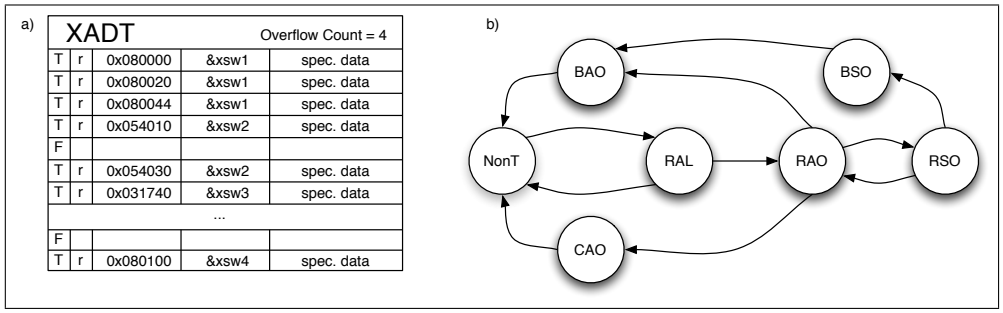

Figure 1: Virtual Transactional Memory. a) transaction read/write sets are stored in a central XADT; b) VTM transaction state transition diagram.

2. VIRTUAL TRANSACTIONAL MEMORY

While small transactions can be supported by the cache and coherence protocol, large transactions require spilling transaction state to memory. In particular, if we want transactions to survive a context switch, we cannot rely on any structures related with a particular processor, including the cache, coherence state, or per-processor in-memory data structures. Rather, the bulk of the transaction state (the read and write sets) must be held in (virtual) memory where it can be observed by any potentially conflicting thread.

In VTM, transaction read and write sets are maintained in a centralized data structure called the *transactional address data table* (XADT) shown in Figure 1a. This data structure is shared by all of the threads within an address space; for the sake of performance isolation — the degree to which the system can prevent the behavior of one application from impacting the performance of others [27, 28] — each virtual address space is allocated its own XADT. Each entry in the XADT stores the address, control state (valid, read/write), data, and a pointer to a *transactional status word* (XSW). Each transacting thread has its own XSW, which holds the transaction's current state. Because the same XSW is pointed to by all of a transaction's XADT entries, a transaction can be logically committed or aborted with a single update to an XSW.

In VTM, a transaction can be in any of seven states, as shown in Figure 1b. When a transaction begins, a transition is made from *non-transactional* (NonT) to *running, active, local* (RAL) where the transaction is held in cache, and abort/commit can be handled in hardware with a transition back to NonT. When the transaction's footprint gets too large, a transition is made to *running, active, overflowed* (RAO). Upon this transition, the transaction must increment the XADT's associated *overflow count*, which signals to other potentially conflicting threads that they must probe the XADT. In order to prevent unnecessary searches of the XADT, VTM provides the *transaction filter* (XF), a counting Bloom filter that can be checked prior to accessing the XADT that conservatively indicates when an XADT access is unnecessary.

From the RAO state, a transaction's XADT entries may be marked as committed or aborted via transitions to *committed, active, overflowed* (CAO) and *aborted, active, overflowed* (BAO), respectively. When the physical commit/abort has completed, by removing the related entries from the XADT, the XSW can be transitioned back to NonT and the overflow counter decremented. The physical commit/abort can potentially be performed lazily — handling committed and aborted XADT entries as they are encountered — and in parallel with the thread's further execution (by allocating the thread a new XSW).

If an interrupt, exception, or trap is encountered, a running transaction (RAL, RAO) is transitioned to the *running, swapped, overflowed* (RSO) state where it no longer adds to the transaction's read/write sets. If a transaction is aborted while it is swapped out, it moves to the *aborted, swapped, overflowed* (BSO) state, and the abort is handled when it is swapped back in (the BAO state).

2.1 Simulated Implementation

Our variant of VTM was implemented through extensions to the x86 version of the Simics full-system simulator [16] and the Linux kernel, version 2.4.18. The primary difference in our implementation from Rajwar *et al.*'s description [22] is that, like LogTM [18], we use *eager versioning*: we allow transaction writes to speculatively update memory after logging the architected values. The VTM hardware was emulated by a Simics module that monitored memory traffic and could be controlled by software through new instructions implemented using Simics' *magic instruction*, a nop (`xchg %bx,%bx`) recognized by the simulator. Although no performance results are included in this paper, we have subjected our implementation to torture tests meant to expose unhandled race conditions, giving us some confidence that our implementation (and hence this text) addresses the salient issues.

While VTM could be implemented as an almost entirely user-mode construct, doing so would rely on the existence of user-mode exception handling. Because x86 currently does not have a user-mode exception handling mechanism, our implementation uses the existing kernel-mode exceptions, and much of the software stack associated with VTM is implemented as part of the Linux kernel. Also, our VTM implementation uses locks in its implementation (so that it doesn't depend on itself), but its critical sections could exploit a technique like speculative lock elision [21].

In keeping with the spirit of VTM, we wanted to minimally impact the execution of processes that are not using transaction support. To this end we add only two new registers that must be set on a context switch, add less than 100 bytes of process state, and add two instructions to the system call path. All other kernel modifications are only encountered by transacting processes.

The VTM hardware/software interface is embodied by two main data structures, shown in Figure 2. The *global*

```
typedef struct global_xact_state_s {
  int            overflow_count;
  xadt_entry_t  *xadt;
  /*************  the following fields are software only  ************/
  int            next_transaction_num;  // for uniquely numbering LTSSs
  spinlock_t     gtss_lock;             // guards the allocation of GTSS fields
  spinlock_t     xact_waiter_lock;      // guards modification of waiter fields
} global_xact_state_t;

typedef struct local_xact_state_t {
  xsw_type_t                  xsw;
  int                         transaction_num;  // for resolving conflicts
  x86_reg_chkpt_t            *reg_chkpt;
  comp_lists_t               *comp_lists;       // discussed in Section 4
  /**** the following are software only fields, described in Section 3 ****/
  struct transaction_state_s *waiters;
  struct transaction_state_s *waiter_chain_prev;
  struct transaction_state_s *waiter_chain_next;
  struct task_struct         *task_struct;
} local_xact_state_t;
```

Figure 2: Data structures for the global and local transactional state segments (GTSS and LTSS, respectively).

transaction state segment (GTSS) holds the overflow count, and a pointer to the XADT. In addition, our kernel allocates additional state for its own use (also discussed below). The *local transaction state segment* (LTSS) holds the XSW, a transaction priority for resolving conflicts, a pointer to storage for a register checkpoint, and additional fields discussed in Sections 3 and 4. The kernel allocates one GTSS per address space (as part of `mm_struct`) and LTSSs on a per thread (or, in Linux terminology, *task*) basis. Pointers to these data structures are written into the two registers (the GTSR and LTSR, respectively) on a context switch.

To meet our goal of minimally impacting non-transacting processes, we delay allocation of data structures until they are required. Specifically, large structures (*e.g.*, the XADT) and per thread structures (*e.g.*, the LTSS) are allocated on demand; if a thread tries to execute a *transaction_begin* and its LTSR holds a NULL, the processor throws an exception whose handler allocates the LTSS, as well as an XADT if necessary. The `gtss_lock` is used to prevent a race condition where multiple threads try to allocate XADTs. The only structure not allocated on demand is the GTSS, because (in our implementation) even threads that are not transacting need to monitor the `overflow_count` field. By allocating the GTSS at process creation time, we avoid having to notify other threads (via interprocessor interrupt) that they need to update their GTSR. Since the GTSS contains only a few scalars and pointers, it results in a small per-process space overhead.

For simplicity, all of the small structures (*e.g.*, GTSS, LTSS) are allocated to pinned memory (*i.e.*, not swapped) to avoid unnecessary page faults. For performance isolation reasons, large structures (*e.g.*, the XADT) are allocated in the process's virtual memory address space. If executing an instruction requires access to XADT data not present in physical memory, the VTM hardware causes the processor to raise a page fault. After servicing the page fault — we made no modifications to the page fault handling code — the operation can be retried.

3. DE-SCHEDULING TRANSACTIONS

While VTM provides support for swapping out threads without aborting their running transactions (and continuing their execution on another processor), this support was intended to handle swapping that results from conventional system activity (*e.g.*, timer interrupts). In this section, we discuss how the VTM system can coordinate with a software scheduler to support de-scheduling/re-scheduling processes based on VTM actions. We present two cases: first, we demonstrate how a transaction conflict can be resolved by de-scheduling one thread until the other thread's transaction either commits or aborts. Second, we show how Harris *et al.*'s intentional wait primitive `retry` can be implemented in an HTM like VTM.

3.1 De-scheduling Threads on a Conflict

A conflict does not necessitate aborting a transaction, an observation made in previous transactional memory systems [18, 20] and earlier in database research [23]. In particular, the conflict is asymmetric: when two transactions conflict, one of them (which we call T1) already owns the data (*i.e.*, it belongs to the transaction's memory footprint) and the other transaction (T2) is requesting the data for a conflicting access, as shown in Figure 3. By detecting conflicts eagerly (*i.e.*, when they occur rather than at transaction commit time) we can prevent the conflict from taking place by stalling transaction T2. For short-lived transactions, stalling T2 briefly can allow T1 to commit (or abort) at which point T2 can continue. If T1 does not commit/abort quickly, we need to resolve the conflict. This conflict can be resolved in many ways (*e.g.*, [12]). If T2 is selected as the "winner," then T1 must be aborted to allow T2 to proceed. In contrast, if T1 "wins," T2 can either be aborted or further stalled, provided the conflict resolution is repeatable so as to avoid deadlock.

If T1 is a long running transaction, T2 may be stalled for a significant time, unnecessarily occupying a processor core. This situation corresponds to the case in a conventionally synchronized critical section where a lock is spinning for a long time. In this section, we demonstrate how our system can be extended to allow such stalled transactions to be de-scheduled until T1 commits/aborts, in much the same way that a `down` on a unavailable semaphore de-schedules a

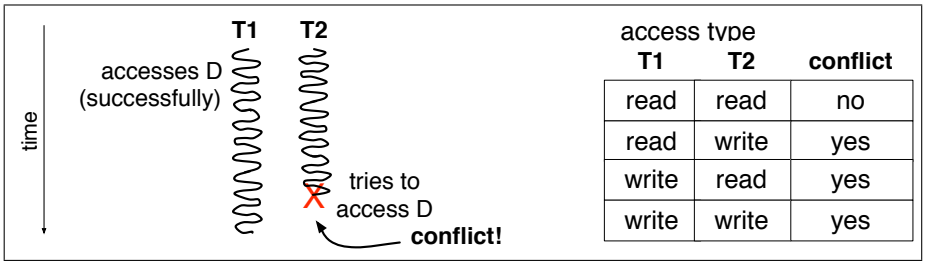

Figure 3: The asymmetric nature of transaction conflicts. Transaction T1 added the data item D to its memory footprint, then transaction T2 tried to access that data in a conflicting way.

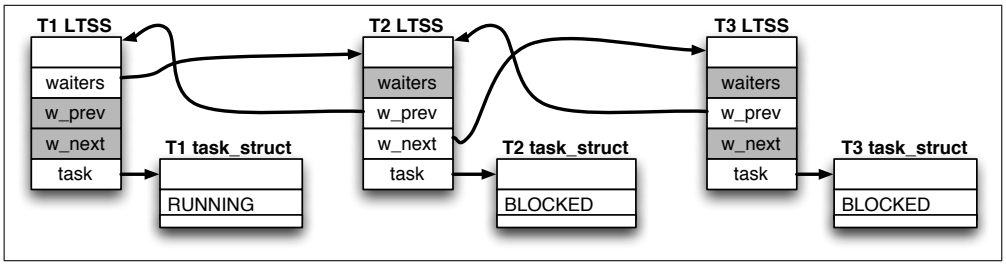

Figure 4: The responsibility for waking up de-scheduled processes is maintained by linking the LTSSs. Shaded fields represent `NULL` pointers. Each LTSS includes a pointer to the `task_struct` for waking the thread.

thread. In the description that follows, we describe an operating system-based implementation that uses the traditional x86 exception model. The same approach could be implemented completely in user-mode, with a user-mode thread scheduler and user-mode exceptions [25].

In order to de-schedule a thread on a transaction conflict, we need to communicate a microarchitectural event up to the operating system. We implement this communication by having T2 raise an `xact_wait` exception, whose handler marks T2 as not available for scheduling and calls the scheduler. The only challenging aspect of the implementation is ensuring that T2 is woken up when T1 commits or aborts.

For T1 to perform such a wakeup, it needs to know two things: 1) that such a wakeup is required, and 2) who to wake up. The first requirement is achieved by setting a bit (`XSW_EXCEPT`) in T1's XSW to indicate that a `xact_completion` exception should be raised when the transaction commits or aborts. The second requirement is achieved by building a (doubly-) linked list of waiters; we use the LTSSs (recall Figure 2) as nodes to avoid having to allocate/deallocate memory, as shown in Figure 4. We also include in the LTSS a pointer to the thread's `task_struct`, which holds the thread's scheduling state.

Code for the `xact_wait` exception handler is shown in Figure 5; we used conventionally synchronized code, but this would be an ideal use for a (bounded) kernel transaction. As part of raising the exception, T2's processor writes the address of T1's LTSS to a control register (`cr2`). A key feature is our transferral of the responsibility of waking up T2 from itself to T1. In particular, we don't want to transfer responsibility if T1 has already committed or aborted. By doing a compare-and-swap on T1's XSW, we can know that T1 was still running when we set the `XSW_EXCEPT` flag, and, therefore, that responsibility has been transferred. Now, T1 will except on commit/abort. In the `xact_completion` exception handler (not shown), it acquires the same lock, ensuring that it will find node T2 inserted in its waiter list.

The only remaining race condition is one that can result from T1 committing and recycling its XSW for another transaction between the conflict and the `xact_wait` exception executing. This is not a problem in our implementation that only slowly recycles XSWs. If this were a problem, it could be handled by either having the VTM unit monitor T1's XSW (via the cache coherence protocol) or by using sequence numbers, but space limitations preclude a detailed discussion.

3.2 Implementing an Intentional Wait

In their software TM for Haskell, Harris *et al.* propose a particularly elegant primitive for waiting for events, called `retry` [9]. The `retry` primitive enables waiting on multiple conditions, much like the POSIX system call `select` or Win32's `WaitForMultipleObjects`, but in a manner that supports composition. Its use is demonstrated by the code example in Figure 6, which selects a data item from the first of a collection of work lists that has an available data item. If all of the lists are empty, then the code reaches the `retry` statement, which conceptually aborts the transaction and restarts it at the beginning.

However, as Harris *et al.* rightly point out, "there is no point to actually re-executing the transaction until *at least one of the variables read during the attempted transaction is written by another thread.*" Because the locations read have already been recorded in the transaction's read set, we can put the transacting thread to sleep until a conflict is detected with another executing thread.

Doing so in the context of our VTM implementation requires a modest modification to the described system. Specifically, two pieces of additional functionality are required: 1) a software primitive is required that allows a transaction to communicate its desire to wait for a conflict, and 2) when another thread aborts a transaction that is waiting, the conflicting thread must ensure that the waiting thread is re-scheduled.

```
asmlinkage void xact_wait_except(struct pt_regs * regs, long error_code) {
  // puts this thread to sleep waiting for T1 to abort or commit
  struct task_struct *tsk = current;  // get pointer to current task_struct
  xact_local_state_t *T1, *T2, *T3;
  xsw_state_t T1_xsw;

  __asm__("movl %%cr2,%0":"=r" (T1)); // get ptr to winner's (T1) xact state
  T2 = tsk->thread.ltsr;              // get ptr to our (T2) xact state
  tsk->state = TASK_UNINTERRUPTIBLE;  // deschedule this thread

  spin_lock(&tsk->mm->context.xact_waiter_lock); // get per address-space lock
  do {
    if ((T1_xsw = T1->xsw) & (XSW_ABORTING|XSW_COMMITTING)) { // already done
      spin_unlock(&tsk->mm->context.xact_waiter_lock);
      tsk->state = TASK_RUNNING;
      return;
    }
  } while (!compare_and_swap(&T1->xsw, T1_xsw, T1_xsw|XSW_EXCEPT))


  T3 = T1->waiters;
  T1->waiters = T2;               // insert into doubly-linked list
  T2->waiter_chain_prev = T1;
  if (T3 != NULL) {
    T3->waiter_chain_prev = T2;
    T2->waiter_chain_next = T3;
  }

  spin_unlock(&tsk->mm->context.xact_waiter_lock);
  schedule();
}
```

Figure 5: Code for de-scheduling a thread on a transaction conflict. In this implementation, a per-address space spin lock is used to ensure the atomicity of transferring to T1 the responsibility for waking up T2.

```
element *get_element_to_process() {
  TRANSACTION_BEGIN;
  for (int i = 0 ; i < NUM_LISTS ; ++ i) {
    if (list[i].has_element()) {
      element *e = list[i].get_element();
      TRANSACTION_END;
      return e;
    }
  }
  retry;
}
```

Figure 6: An illustrative example demonstrating the use of `retry`. Retry enables simultaneously waiting on multiple conditions (multiple lists in this case); conceptually, the transaction is aborted and re-executed when the `retry` primitive is encountered.

Our implementation provides the first primitive with an instruction that raises a `retry` exception. In the exception handler (not shown), the process is blocked, the transaction's priority is set to a minimum value (so that it will always be aborted when a conflict occurs), and it marks its XSW with a `XSW_RETRY` bit indicating that a conflicting thread is responsible for waking up this sleeping thread. As above, a `compare-and-swap` is used to set this bit, so the software knows that the XSW was not already marked as aborted. If the transaction has already been aborted, the thread is set back to state `TASK_RUNNING` and the process returns from the exception. Otherwise the handler calls `schedule()` to find an alternate thread to schedule on this processor.

When a thread aborts a transaction with the `XSW_RETRY` bit set, it completes the current instruction, copies the XSW address of the aborted thread to a control register (`cr2`), and raises a `retry_wakeup` exception. This exception handler reads the `task_struct` field from the aborted transaction's LTSS and wakes up the thread using `try_to_wakeup` Also, a potential race condition exists that requires adding a check to the code in Figure 5 to verify that the transaction is not waiting on a `retry`ing transaction, before it calls `schedule()`.

4. PAUSING TRANSACTIONS TO MITIGATE CONSTRAINTS

In the previous section, we discussed dealing with conflicts efficiently. In this section, we consider how pausing a transaction (without pausing the thread's execution) can be used to avoid conflicts for data elements with high contention, as well as allow actions with non-memory-like semantics to be performed within transactions. While a transaction is paused, its thread is allowed to perform any action, including system calls and I/O, and its memory operations are not added to the transaction's footprint. We begin this section with an illustrative example and conclude with a collection of dynamic memory allocator-based examples to demonstrate the benefit and use of pausing transactions.

4.1 A Simple Example: Keeping Statistics

In Figure 7a, we show a transaction that increments a global counter to maintain statistics. Such code can be problematic, because transactions that are otherwise independent may conflict on updates to this statistic. While

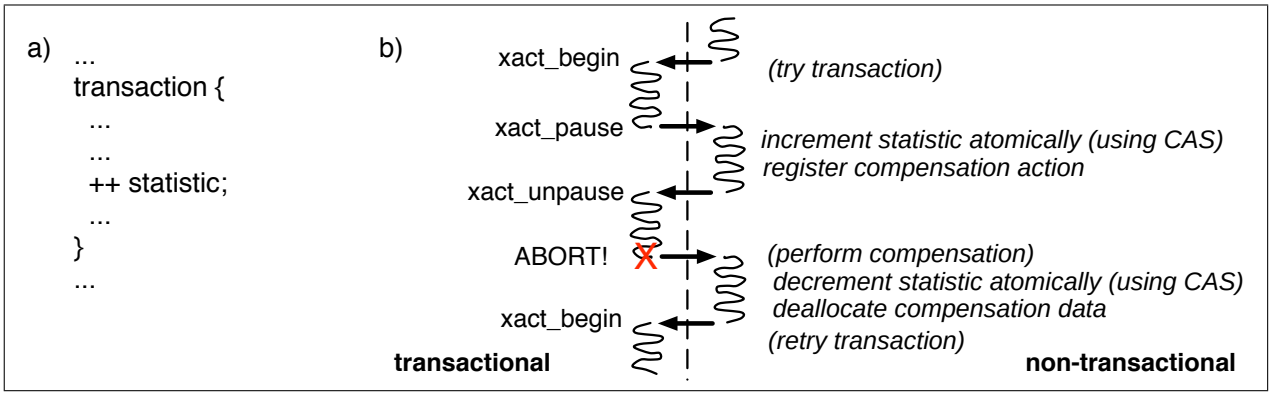

Figure 7: Incrementing statistics using pausing and compensation when precise intermediate value is not required. a) A "hot" statistic is incremented within a transaction, b) conflicts can be avoided by pausing before incrementing (using a compare-and-swap) the statistic and performing compensation if the transaction aborts.

seemingly trivial, such statistics impact the scalability of existing hardware TMs [5]. The problem derives from the fact that the TM is providing a stronger degree of atomicity than the application requires: while the statistic's final value should be precise, an approximate value is generally sufficient while execution is in progress.

We can exploit the reduced requirements for atomicity, by non-transactionally performing the increment from within the transaction. Note that this is not an action automatically performed by a compiler, but, rather, one performed by a programmer to tune the performance of their code. In Figure 7b, we sketch an implementation that pauses the transaction before performing the counter update, so that the counter is not added to the transaction's read or write sets. To preserve the statistic's integrity, we also register a compensation action — to be performed if the transaction aborts — that decrements the counter. Such an implementation achieves the application's desired behavior without unnecessary conflicts between transactions. An alternative implementation could just register an action to be performed after the transaction commits that increments the counter. In the next subsection, we describe the necessary implementation mechanisms.

4.2 Transaction Pause Implementation

Hardware-wise, implementing the transaction pause is quite straightforward; it is simply another bit that modifies the XSW state. We add two new instructions `xact_pause` and `xact_unpause`, which set and clear this bit, respectively.

As previously noted, when a transaction is paused, addresses loaded from or stored to are not added to the transaction's read and write sets (*i.e.*, no entries are added to the XADT). Instead concurrency must be managed using other means (*e.g.*, the use of compare-and-swap instructions to update the statistic). Nevertheless, we check for conflicts with transactions, just as if we were executing non-transaction code. The one exception is that we should ignore conflicts with the thread's own paused transaction. It is not uncommon to want to pass arguments/return values between the transaction and the paused region, and some of these may be stored in memory.

Furthermore, when the paused region stores into a memory location covered by the transaction's write set, clean semantics dictate that the write should not be undone if the transaction is aborted. We would like just to remove the written region from the transaction's write set, but the granularity at which the write set is tracked may prevent this. We have implemented this case by causing such stores to write both to memory and the associated XADT entry, so that the write is preserved on an abort. In many respects, the semantics of performing writes in paused regions resemble the previously proposed open commit [19]; while pausing is, in some ways, a weaker primitive than open commit (transaction semantics are not provided in the paused region), in other ways it is more powerful (non-memory-like actions can be performed). Furthermore, pause is simpler to implement, because support for true nesting, which in turn requires supporting multiple speculative versions for a given data item, is not required.

Because the actions within a paused region will not be rolled back if the transaction aborts, it may be necessary to perform some form of *compensation* [6, 7, 13, 26] to functionally undo the effects of a paused region. As such, we allow a thread to register a data structure that includes pointers for two linked lists (shown in Figure 8), one for actions to perform upon an abort and another for actions to perform upon a commit. Each list node includes a pointer to the next list element, a function pointer to call in order to perform the compensation, and an arbitrary amount of data[1] (for use by and interpreted by the compensation function). If a transaction aborts, it performs the actions in the `abort_actions` list and discards the actions in the `commit_actions` list. On a commit, it does the inverse. To ensure that it leaves all data structures in a consistent state, as well as has a chance to register any necessary compensation actions, we don't handle an abort (*i.e.*, restore the register checkpoint) while a transaction is paused. Instead, the abort is handled when the transaction is unpaused.

In the proposed implementation compensating actions are not performed atomically with the transaction. While we have yet to identify a circumstance where this is problematic, an alternative approach would enable the appearance of atomicity by serializing commit. Logically, if we prevent any other threads from executing during the execution of the

[1]To avoid any dependences on the context in which the compensation action is performed, we require the programmer to encapsulate any necessary context information into the compensation action's data structure.

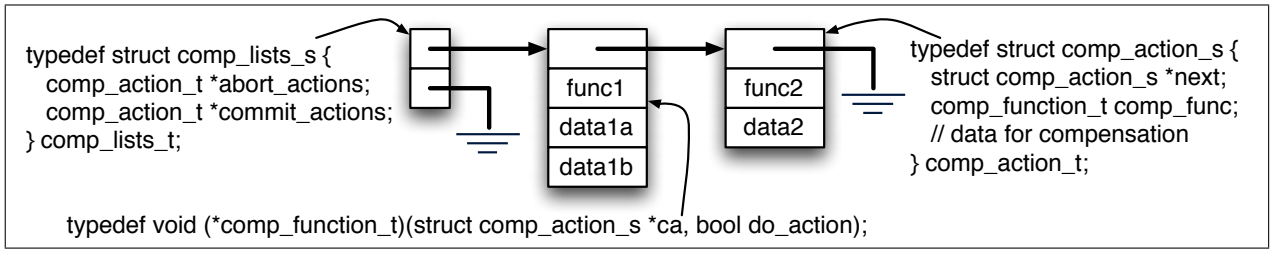

Figure 8: An architecture for registering compensation actions. Each transactions maintains lists of actions to perform on a commit and on an abort. The `do_action` argument of `comp_function_t` indicates whether the compensation should be performed or the `comp_action_t` should just be deallocated.

compensation code, we provide atomicity while enabling arbitrary non-memory operations in the compensation code. The implementation need not be quite this strict, as other transactions can be allowed to execute (but not commit) until they attempt to access data touched by the committing transaction; if the compensation code touches data from another transaction, the other transaction must be aborted. If strong atomicity [3] is desired, non-transactional execution cannot proceed (as each instruction is logically a committing transaction). Because such support for atomic compensation constrains concurrency, it could be designed to be invoked only when it was required.

From a software engineering perspective, it is desirable to be able to write a single piece of code that can be called both from within a transaction (where it registers compensation actions) and from non-transactional code (where no compensation is required). To this end, the `xact_pause` instruction returns a value that encodes both: 1) whether a transaction is running, and 2) whether the transaction was already paused. By testing this value, the software can determine whether compensating actions should be performed. Furthermore, by passing this value to the corresponding `xact_unpause` instruction, we can handle nested pause regions (without the VTM hardware having to track the nesting depth) by clearing the pause XSW bit only if it was set by the corresponding `xact_pause`[2].

Clearly, correctly writing paused regions with compensation can be challenging, but they should not have to be written by most programmers. Instead, functions of this sort should generally be written by expert programmers and provided as libraries, much like conventional locking primitives and dynamic memory allocators. In the next section, we demonstrate how a dynamic memory allocator can be readily implemented using pause and compensation, because programs generally do not rely on which memory is allocated.

4.3 Pausing in Dynamic Memory Allocators

Dynamic memory allocation is a staple of most modern programs and, due to the modular nature of modern software, likely to take place within large transactions. For this discussion, we will concentrate on C/C++-style memory allocation, but, as we will see, the motivation for pause goes beyond these particular languages. While we demonstrate the fundamental issues in a relatively simple `malloc` implementation (Doug Lea's `malloc`, `dlmalloc` [14]), the same issues are present even in advanced parallel memory allocators (*e.g.*, Hoard [2]).

```
void *X, Y, Z = malloc(...);
transaction {
  X = malloc(...);
  free(Z);
  Y = malloc(...);
  free(X);
}
free(Y);
```

Figure 9: Example transaction that includes memory allocation and deallocation.

In Figure 9, we illustrate a short code segment that illustrates the three cases that we have to correctly handle: 1) an allocation deallocated within the same transaction (`X`), 2) an allocation within a transaction that lives past commit (`Y`), and 3) an existing allocation that is deallocated within a transaction (`Z`). In executing this code (and code like it), we want to ensure two things: 1) we don't want to leak memory allocated within a transaction (even if an abort occurs), and 2) we want to free memory exactly once and not irrevocably so until the transaction commits. As will be seen, by correctly handling cases 2 and 3, case 1 is handled as well.

Here, we consider two implementations of `malloc`: the first is quite straightforward (and merely for illustration), executing the whole `malloc` library non-transactionally and the second where pausing and compensation is only used to deal with the non-idempotent system calls `mmap` and `munmap`.

In the first implementation, we construct new wrappers for the functions `malloc` and `free`. The wrappers, which comprise nearly the entire change to the library, are shown in Figure 10. The `malloc` wrapper first pauses the transaction, then (non-transactionally) performs the memory allocation. Then, if the code was called from within the transaction, it registers an abort action that will `free` the memory, preventing a memory leak if the transaction gets aborted. If the transaction succeeds, the `abort_actions` list will be discarded.

The case of deallocation is complementary. When `free` is called from within a transaction, we do not want to irrevocably free the memory until the transaction commits. As such, when executed inside a transaction, our wrapper does nothing but register the requested deallocation in the `commit_actions` list. If the transaction aborts, this list will be discarded. Only when the transaction commits will the deallocation actually be performed. Concurrent accesses to the memory allocator are handled using the library's exist-

[2]A similar idea could be used for `xact_begin` to support transaction nesting without keeping a nesting depth count.

```
void *malloc(size_t bytes) {
   void *ret_val;
   int pause_state = 0;
   XACT_PAUSE(pause_state);
   ret_val = malloc_internal(bytes);
   if (INSIDE_A_TRANSACTION(pause_state)) {  // if in a transaction, register compensating action
      comp_lists_t *comp_lists = NULL;
      XACT_COMP_DATA(comp_lists);           // get a pointer to the compensation lists
      free_comp_action_t *fca = (free_comp_action_t *)malloc_internal(sizeof(free_comp_action_t));
      fca->comp_function = free_comp_function;
      fca->ptr = ret_val;
      fca->next = comp_lists->abort_actions;
      comp_lists->abort_actions = (comp_action_t *)fca;
   }
   XACT_UNPAUSE(pause_state);
   return ret_val;
}

void free(void* mem) {
   int pause_state = 0;
   XACT_PAUSE(pause_state);
   if (INSIDE_A_TRANSACTION(pause_state)) {  // if in a transaction, defer free until commit
      comp_lists_t *comp_lists = NULL;
      XACT_COMP_DATA(comp_lists);           // get a pointer to the compensation lists
      free_comp_action_t *fca = (free_comp_action_t *)malloc_internal(sizeof(free_comp_action_t));
      fca->comp_function = free_comp_function;
      fca->ptr = mem;
      fca->next = comp_lists->commit_actions;
      comp_lists->commit_actions = (comp_action_t *)fca;
   } else {
      free_internal(mem);
   }
   XACT_UNPAUSE(pause_state);
}

typedef struct free_comp_action_s {
   struct comp_action_s *next;
   comp_function_t comp_function;
   void *ptr;
} free_comp_action_t;

void free_comp_function(comp_action_t *ca, int do_action) {
   if (do_action) {
      free_comp_action_t *fca = (free_comp_action_t *)ca;
      free_internal(fca->ptr);
   }
   free_internal(ca);
}
```

Figure 10: Wrappers for `malloc` and `free` that perform them non-transactionally. The original versions of `malloc` and `free` have been renamed as `malloc_internal` and `free_internal`, respectively. When executed within a transaction, `malloc` registers a compensation action that frees the allocated block in case of an abort, and `free` does nothing but register a commit action that actually frees the memory. To register compensation actions, the transaction must dynamically allocate memory (note the use of `malloc_internal`) and insert it into the list of compensation actions stored in the LTSS (recall Figure 2).

ing mutual exclusion primitives.

An alternative implementation executes the bulk of the memory allocator's code as part of the transaction. In the common case, the transactional memory system ensures that memory is not leaked: memory allocated/deallocated by an aborting transaction is restored by undoing the transaction's stores. Only when the allocator interacts with the kernel is there potential for a problem, as kernel activity is not included in the transaction for reasons of performance isolation [28]. Instead, the VTM hardware sets the transaction into a `SWAPPED` state during kernel execution, so system call activity is not rolled back on an abort. While this is perhaps not problematic for idempotent system calls like `brk()` and `getpid()`, it is problematic for `mmap()`, which is not idempotent.

`dlmalloc` uses `mmap()` to allocate very large chunks (> 256kB) and when `sbrk()` cannot allocate contiguous chunks. When `mmap()` is called, the Linux kernel records the allocation (in a `vm_area_struct`), in part to guarantee that it doesn't allocate the memory again. If a transaction calling `mmap()` aborts, the application will have no recollection of the allocation, but the kernel will, resulting in memory leak of the virtual address space[3].

To prevent such a leak, we wrap the call to `mmap()` in a paused region and register a compensation action to `munmap()` the region if the transaction is aborted, much in the same spirit as the `malloc` wrapper in Figure 10. Correspondingly, calls to `munmap` that are performed within transactions are deferred until the transaction commits.

In general, this second approach is likely preferable, because less effort has to be spent registering and disposing of compensation actions. The primary drawback of this approach is that conflicts will result if multiple transactions try to allocate memory from the same pool, but this problem can be largely mitigated by using a parallel memory allocator (*e.g.*, Hoard [2]) that provides per-thread pools of free memory.

5. RELATED WORK

Concurrently with this work, Carlstrom *et al.* proposed an implementation of open nesting to handling high-contention and actions with non-memory-like semantics [17]. In many respects, their implementation of abort/commit actions is similar to ours, with one noteworthy exception: they guarantee that the abort/commit handlers execute atomically with the transaction by performing it during the commit process and preventing other transactions from committing simultaneously. While this programming abstraction is cleaner, it can also serialize commit unnecessarily; for example, atomicity is not required in our malloc example. The best of both worlds may be to support both approaches and allow the programmer to make the simplicity/performance trade-off themselves.

Also noteworthy in the work, they deride the notion of a transactional pause primitive as "redundant and dangerous." In contrast, we don't view the two primitives as mutually exclusive, but rather as representing slightly different trade-offs in software complexity and capability. While open-nesting provides a cleaner programming interface by eliminating the lock-based concerns of paused regions, the fact that both will require compensation code ensures that neither will be written except by expert programmers. Pausing, however, unlike open nesting, enables transactions to contain code not written in transactions. We believe that it is unlikely that transactions will completely replace locks for reasons of performance isolation (especially with respect to kernel execution [28]) as well as legacy code. In addition, because composition of paused regions is handled in software, we do not have the handle the complexity of supporting arbitrary nesting in hardware, a topic not yet handled by the literature for hardware support of open nested transactions.

Also, the ATOMOS extensions to Java [4], work done concurrently with our implementation, also provide an implementation of `retry`. The major differences between the implementations are two-fold: 1) the ATOMOS implementation requires the programmer to explicitly identify the set of values on which to wait using the "watch" primitive; requiring explicit identification of the watch set presents the possibility that a programmer will omit necessary items and as well as a software maintenance headache, without a clear need for the enabled selectivity, 2) the ATOMOS implementation requires a processor to be dedicated to serve as a thread scheduler, a requirement that seems to derive from the fact that transactions cannot live across context switches. In a machine with a conventional virtual memory system, it seems likely that one scheduler processor would be required for each virtual address space, and it is unclear what happens if the composite watch set of many threads exceeds the size of what can be supported directly by the transaction hardware. In contrast, our implementation supports waiting on the whole existing read set and requires no dedicated processors due to VTM's existing support of "unbounded" transactions that can survive context switches.

6. CONCLUSION

With highly-concurrent machines prominently on the mainstream roadmaps of every computer vendor, it is clear that a program's degree of concurrency will be the primary factor affecting its performance. This paper reflects our belief that the power of transactional memory will not be in how it performs on applications that have already been parallelized, but in how it enables new applications to be parallelized. In particular, many applications that have yet to be parallelized have inherent parallelism, but not of a regular sort that can be expressed with DOALL-type constructs. Instead, the parallelism is unstructured — requiring significant effort on the programmer's part to manage the concurrency using traditional means — and exists in varying granularities. The key goal of a transactional memory system should be to allow the programmer to trivially express the existence of this potential concurrency at its natural granularity.

A key component of this strategy is providing the programmer with those primitives that facilitate the expression of parallelism. While previous work on hardware transactional memory has shown to support the atomic execution of arbitrarily sized regions of normal code, it has yet to provide the richness of the interface provided by software transactional memory systems. This paper attempts to shrink the functionality gap between software transactional memory systems and hardware ones, through demonstrating how a hardware TM can interface with a software thread scheduler and by supporting non-transactional memory ac-

[3]To avoid errors of this sort in general, we've modified the Linux kernel to kill unpaused transactions in the `system_call()` interrupt vector.

cesses within a transaction memory system. Furthermore, we show that functionally, these techniques represent small extensions to existing proposals for hardware transactional memory.

7. ACKNOWLEDGMENTS

This research was supported in part by NSF CCR-0311340, NSF CAREER award CCR-03047260, and a gift from the Intel corporation. We thank Brian Greskamp, Pierre Salverda, Naveen Neelakantam, Ravi Rajwar, and the anonymous reviewers for feedback on this work.

8. REFERENCES

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

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

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

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The ATOMOS Transactional Programming Language. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[5] C. Click. A Tour inside the Azul 384-way Java Appliance: Tutorial held in conjunction with the Fourteenth International Conference on Parallel Architectures and Compilation Techniques (PACT), Sept. 2005.

[6] A. A. Farrag and M. T. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, 1989.

[7] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

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

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

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

[11] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[12] W. N. S. III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-Fourth Symposium on Principles of Distributed Computing (PODC)*, 2005.

[13] H. F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 95–106, 1990.

[14] D. Lea. A memory allocator, http://gee.cs.oswego.edu/dl/html/malloc.html.

[15] D. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of the ACM Conference on Language Design for Reliable Software*, pages 128–137, Mar. 1977.

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

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

[18] 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.

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

[20] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.

[21] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

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

[23] D. J. Rosenkrantz, R. Stearns, and P. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[24] H. Sutter and J. Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.

[25] C. A. Thekkath and H. M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[26] S. Vaucouleur and P. Eugster. Atomic features. In *Proceedings of the workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[27] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, Oct. 1998.

[28] C. Zilles and D. Flint. Challenges to Providing Performance Isolation in Transactional Memories. In *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*, pages 48–55, June 2005.