Exploring Support for Long-Lived Transactions

Lee Baugh and Craig Zilles
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {leebaugh, zilles}@uiuc.edu

Abstract

If future systems provide efficient support for atomic execution (i.e., by transactional memory), we believe that programmers will use them in two ways: 1) short-lived transactions — corresponding to existing critical sections — for accessing high-contention shared data, and 2) long-running, large-memory footprint transactions to exploit concurrency where little contention is expected, but an absence of contention cannot be guaranteed. While there is much consensus on how the first class of transactions should be supported, much less is known about the second class.

This paper makes the following contributions toward efficiently supporting this second class of transactions: 1) a hierarchical organization for storing transactional data in memory that avoids pointer chasing and can be scaled gracefully without having to re-hash its contents, 2) modifications to a previously proposed transactional memory system to support committing transactions while some of their data is not resident in memory, 3) 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 4) 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.

I. INTRODUCTION

A number of challenges to further improvement of single-core performance have led to an industry-wide shift towards multi-core machines, where multiple processors are integrated onto a single silicon chip. While the multi-core approach enables microprocessor vendors to exploit increasing transistor density without corresponding increases in power efficiency and designer productivity, it introduces a serious programming challenge into the mainstream. Even expert programmers find it difficult to write reliable, high-performance parallel programs.

One effort to mitigate this looming challenge is research in Transactional Memory (TM) [1, 7–10, 16, 19], an effort to provide a better programming primitive for managing concurrency. Presently, the dominant primitive for managing concurrency is locks. The problems with locks are well documented (e.g., [21]): they don’t compose, they have a possibility for deadlock, they rely on programmer convention, they represent a trade-off between simplicity and concurrency. TM addresses a number of these problems by providing an efficient implementation of atomic blocks [14], 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.

While TM can be implemented as a purely software construct, significantly more efficient implementations are enabled by a modest amount of hardware support. Herlihy and Moss observed that small transactions could be supported efficiently in hardware with modest changes to the cache and cache coherence protocol [10]. While many critical sections are small, it is undesirable, from a programmer’s perspective, to constrain the footprint of data touched by a transaction to fit in the cache; programmers do not want to think about cache capacity and conflicts. More recently, a collection of hardware TM systems have been proposed that provide unbounded transactions by supporting small transactions using the cache/coherence protocol and falling back on another (generally less efficient) mechanism for large transactions.

Despite having engineered systems that support unbounded transactions, many researchers have concluded that, in practice, transactions will have small memory footprints and will be short lived [1, 16]. We question such conclusions, because they have been largely based on experiments using conventionally-synchronized software that was highly-engineered for concurrency (e.g., the Linux kernel, the SPLASH benchmarks), which researchers have automatically converted to use transactions. To achieve the desired scalability in the context of conventional synchronization, these programs have been engineered to minimize the size of critical sections, but this is exactly the kind of effort that TM is meant to alleviate.

In contrast, we believe that if transactions become a first-class general-purpose programming primitive, then programmers will want to write large-memory footprint, long-lived transactions — what we’ll refer to as large transactions — in situations involving minimal contention. We believe this to be true even if large transactions cannot be implemented with negligible

1Obviously, no system is truly unbounded; we use the term to refer to any system that can handle transactions with a memory footprint that does not fit in the cache. In practice, each system is bounded by one or more of: 1) the size of physical memory, 2) the size of virtual memory, and 3) the maximum length of time between interrupts.
overhead. If large transactions enable a doubling of concurrency, by allowing programmers to introduce concurrency where they otherwise would not, they can tolerate a significant execution overhead.

In this paper, we explore a number of issues related to large hardware-supported transactions that have been heretofore ignored in the literature. Our work comprises the engineering of a hardware TM to efficiently support large transactions (implemented through extensions to a full-system simulator and modifications to the Linux kernel: Section III) and the adaptation of primitives proposed in the context of software TM to a hardware TM. Specifically, we make the following four contributions:

1) We propose a novel organization for storing memory-resident transaction data that enables efficient storage and searching, while scaling gracefully between small and large overflowed transactions without needing to rehash the contents (Section III-B).

2) We adapt a previously proposed TM system to support committing transactions while their footprint is partially non-resident in memory, thereby increasing the system’s performance isolation (Section III-C).

3) We discuss how to efficiently deal with conflicts between large transactions by de-scheduling one of the threads until the other commits (Section IV). We further show how this idea can be extended to allow a transaction to wait on many conditions, much like the `select()` system call.

4) We demonstrate how non-transactional work can be supported in the context of a hardware transaction using a transaction `pause` primitive (which temporarily disable the hardware transaction support) and compensation code (Section V).

Pausing transactions enables supporting O/S activity and avoiding conflicts on frequently shared data in cases where true atomicity is not required at the application level.

In this paper, our goal is not to quantify the performance of any TM system, but, rather, to understand the implementation requirements to support large transactions and demonstrate new functionality that can be achieved. Furthermore, we focus here exclusively on user-mode (application) transactions where we see the greatest motivation for large transactions; while kernel implementers will likely benefit from hardware-supported transactions, it is unclear whether the kernel should use unbounded transactions. Finally, this work builds heavily on the previously proposed Virtual Transactional Memory (VTM) [19], which we briefly overview in the next section.

II. BACKGROUND: VIRTUAL TRANSACTIONAL MEMORY (VTM)

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, the transaction read and write sets are maintained in a centralized data structure called the transactional address data table (XADT) shown in Figure 1. 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 [23, 24] — 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). The XSW holds the transaction’s current state and, because it is shared by all of a transaction’s XADT entries, enables the logical commit or abort of a transaction with a single update to an XSW.

![Fig. 1. Virtual Transactional Memory. a) transaction read/write sets are stored in a central XADT; b) VTM transaction state transition diagram.](image)

In VTM, a transaction can be in any of seven states, as shown in Figure 1. 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 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 commit, 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).

III. Efficiently Supporting Large Transactions

In this section, we present the challenges and solutions we identified relating to supporting large transactions in a transactional memory system. After providing an overview of operating system support, we discuss: 1) how a hierarchically-structured XADT can efficiently support both large and small transactions, 2) how the commit protocol of VTM must be modified to support committing partially swapped transactions, and 3) an alternate filter mechanism that eliminates some of the challenges presented by the VTM’s transaction filter (XF).

Our variant of VTM was implemented through extensions to the x86 version of the Simics full-system simulator [15] and the Linux kernel, version 2.4.18. The primary difference in our implementation from Rajwar et al.’s description [19] is that, like LogTM [16], 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. We do not include performance results for most of the concepts discussed, because they represent new kinds of functionality for which there is no existing mechanism to which we could compare. Nevertheless, 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.

A. Operating System Support

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 transaction state segment (GTSS) holds the overflow count, a pointer to the XADT, an indicator of size of the currently allocated XADT, and a bitmap that indicates which XADT pages are non-zero (discussed below). 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 IV and V. 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 LTSS 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 kernel memory that is 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.
typedef struct global_xact_state_s {
    int overflow_count;
    xadt_entry_t *xadt;
    int max_level;  // XADT size (discussed below)
    void *xadt_page_bitmap;  // indicates non-zero XADT pages
    int tlb_flush_doorbell;  // for communicating tlb flushes
    /************* 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_t;  // discussed in Section 5
    /**** the following are software only fields, described in Section 4 ****/
    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;

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

B. Data structure issues

A TM system is only unbounded to the degree that its transaction log (e.g., VTM’s XADT) is unbounded; thus the structure of the log is of primary importance to the feasibility of a transactional memory system. It can have a first-order impact on the performance and degree of performance isolation that can be achieved by a transactional memory system. Yet, the previous literature on hardware TMs has left this subject almost entirely unexplained.

The XADT supports four essential functions: searching for a matching element (to detect conflicts), insertion of a new element, transaction abort, and transaction commit. Because XADT lookups (part of both searches and insertions) are much more common than aborts/commits\(^2\), the two primary design criteria for the XADT are efficiency of search and space efficiency. As long as commit and abort can be completed for arbitrarily-sized transactions, the performance of these operations is of secondary importance, because they can be performed lazily.

In a hash table-based structure (the conventional wisdom on how overflow data will be organized) there is a tension between search and space efficiency; typically search efficiency decreases as the structure fills up. This appears to be particularly problematic for the XADT whose required size can vary dynamically throughout the program’s execution, potentially inducing cycles of enlarging the XADT for capacity and search efficiency then contracting to regain space efficiency after a large transaction has completed. Such cycles are undesirable because resizing generally requires a rehashing of every element; most TM proposals have advocated aborting in-flight transactions when resizing the transaction log.

Furthermore, neither of the conventional ways of organizing a hash table (open addressing and chaining) are particularly well suited for a transaction log. In particular, open addressing, which sequences through a deterministic pattern of alternate locations in the hash table, does not handle deletions well at all. Chaining, where the hash table is an array of pointers to linked lists, has the following undesirable characteristics: 1) the sequential nature of traversing a linked data structure exacerbates memory latency, 2) the space overhead of maintaining the inter-element pointers, and 3) the complexity of manipulating the linked list by the VTM hardware. We desire a structure in which searches are quick and bounded, which offers fairly high utilization of allocated pages, which is not excessively complex, and which can scale both up and down in size gracefully.

We propose a hierarchical transaction log, consisting of a set of arrays of elements of exponentially increasing size, as shown in Figure\(^3\). Each level – that is, each array of elements – is treated as a hash table, where failure leads to the search continuing on the next level down. Each level has its own independent hash function, so as the number of available levels increases, the likelihood of false aliasing dramatically reduces. To perform a lookup, one performs a hash access on successive levels until a match is found or the last level has been searched. Several, or all, levels may be searched in parallel, yielding very quick searches.

Furthermore, growing and shrinking the log is simply a matter of adding and removing levels; none of the data on the upper levels needs to be moved. By mapping a large structure (e.g., 64MB) in virtual memory, we can even allocate physical memory for the levels at a page granularity. A key property of the design is that, by preferentially inserting in upper levels, pages in

\(^2\)The XADT services exactly one abort or commit for each overflowed dynamic transaction, whereas at least one lookup occurs for each distinct memory access within a transaction, as well as some portion of non-transactional accesses when at least one transaction has overflowed..
The structure of a hierarchical XADT. The figure shows a search through the XADT on address $a$. For the purpose of illustration, this example uses an XADT with two XADT entries in the top level and two entries per page. In a), we have a search on $a$ in which one hash value is produced for each XADT level. In b), the same search is superimposed on the XADT bitmap, with shaded entries representing the presence of transaction data in that page. Because the presence bits of only the top two levels are set, only those levels will be accessed.

The lower levels will be densely populated and pages in the upper levels will be completely empty. This is different from a non-hierarchical hash table, in which occupancy is likely to be evenly distributed throughout the table, and every page in the table is equally likely to be used.

Figure 4 relates data on the performance of a hierarchical XADT under a synthetic load. With the assumption that those transactions that overflow will be at least of non-trivial size, we've simulated an XADT whose two lowest levels are 128kB in size each, with upper levels increasing by powers of two. Our hash functions are constructed with low address bits being directly mapped to low index bits, so that locality in the transaction’s reference stream results in locality in the XADT, yielding near perfect occupancy for sequential streams. Random streams and mixtures of short sequential streams and random-accesses are somewhat less storage efficient, but still consistently yield occupancies in the 35-45% range. In all cases, the number of accesses (all of which can be performed in parallel) grows very slowly, at a rate that increases logarithmically with the number of XADT entries allocated.

The behavior of a hierarchical XADT varies smoothly over a large range of operating conditions. As number of entries is increased, the number of memory accesses to perform a lookup grows as the log of the number of entries, and the occupancy remains around 35-45%. Note in (b) that the line for the sequential stream overlays the 100% occupancy line.

xadt_entry_t *xadt_lookup(address_t address) {
    for (int i = 0 ; i < gtsr.max_level ; ++ i) { // iterate through all levels
        unsigned index = do_hash(i, address); // get the hash for address at level i
        if (get_xadt_page_bitmap_entry(gtsr.xadt_page_bitmap, i, index) & HW_BIT) { // bit set means nonempty page, must take the search to the xadt
            xadt_entry_t *entry = get_xadt_entry(gtsr.xadt, i, index); // get the entry
            if (entry->address == address) { // found a match
                return entry;
            }
        }
    }
    return NULL;
}

The XADT lookup algorithm. While written as sequential code, the hardware overlaps the execution of the loop iterations, because they are independent.

On a lookup, we would like to quickly know which parts of the XADT are in active use, to avoid examining pages of the transaction log which contain no valid entries. Not only does it save on memory bandwidth, it avoids unnecessarily allocating pages to physical memory. We therefore maintain the xadt_page_bitmap, which indicates which pages of the XADT are
not empty. Two bits are associated with each page, both of which are set when the hardware writes an entry to a page; the first (read by the VTM hardware) denotes whether the page is currently occupied, and the second (used by software) facilitates determining when pages are clear and can be deallocated, as described shortly. As for the XADT itself, we rely on the cache coherence protocol to keep this bitmap coherent. At two bits per page, a single 4kB page bitmap, which can be easily cached in the on-chip memory hierarchy, can serve a 64MB XADT.

To deallocate XADT pages and reclaim XSWs, the operating system periodically performs a sweep of the XADT, visiting the pages in the transaction log whose bits in the xadt_page_bitmap are set. First, the O/S clears the second (software) bit of the xadt_page_bitmap, then it traverses the page for valid entries. For each valid entry, the software invokes the VTM hardware to scrub the entry, releasing a committed entry, rolling-back an aborted entry, and trying to push an active entry to a lower level of the XADT. If all of the entries on the page can be scrubbed and the software xadt_page_bitmap bit is still clear, then the O/S can attempt to clear the hardware xadt_page_bitmap bit, via a compare-and-swap operation on the bitmap entry. When a complete sweep of the XADT has been completed since an XSW was marked committed or aborted — tracked in software by recording the sweep that was ongoing when an XSW was replaced — then the XADT is known to be free of entries associated with a given XSW, so the XSW can be transitioned to NonT and reused, and the overflow count decremented.

C. Supporting Out-Of-Core Commits

An important component to performance isolation is the ability for applications to execute with an arbitrary allocation of physical memory; with this requirement in place, the kernel can fairly allocate resources and each application can make forward progress at the rate allowed by that resource allocation. To ensure forward progress in an unbounded transactional memory system given an arbitrary physical memory allocation, we have to support out-of-core commits (i.e., committing a transaction when some (or all) of its data is not resident in physical memory). As originally proposed [19], supporting out-of-core commits is not practical in VTM; we discuss simple modifications to VTM to enable out-of-core commit.

The difficulty with VTM derives from its attempt to excuse non-transacting threads from having to detect conflicts. Instead, as proposed, transacting threads are responsible for detecting conflicts created by non-transacting threads. When a transaction is active (allocated to a processor) the coherence protocol can be used to detect conflicts (somewhat conservatively for overflowed transactions); when the transacting the thread has been de-scheduled, it becomes more challenging. The mechanism proposed is that when a swapped transaction is re-scheduled on a processor, it tries to re-validate its read set by comparing the values that it read (presumably stored in the XADT) to the current values in memory. Not only does this require storing much more information in the XADT (blocks not written would otherwise have no reason to store their data, and blocks read before written would need to buffer two copies of the data), but it also assumes that all the data is resident in physical memory.

If the transaction’s data (both its XADT entries and the pages in normal memory to which they correspond) are not memory resident, it will incur page faults during the re-validation attempt. A page fault typically leads to the faulting thread being de-scheduled, requiring the re-validation attempt to be started anew when the thread is swapped back in. If the O/S is limiting the application to a given physical memory allocation, it will reclaim other physical page frames (containing other subsets of the transaction’s working set) as it allocates new ones, leading the transaction to forever attempt fruitlessly to re-validate.

This issue is easily addressed by requiring non-transacting threads to detect conflicts as if they were transacting. With this change in place, we demonstrated that our VTM implementation can perform out-of-core commits by completing transactions that span more pages than can fit in the physical memory of the simulated machine. Furthermore, the VTM implementation is actually thereby simplified, because a number of special cases are eliminated. The only potential downside is the cost to non-transacting threads of having to detect conflicts when a transaction within the process has overflowed.

In most applications, the number of true conflicts between transactions and non-transactional execution is expected to be small, so the true cost of conflict detection is likely to be the cost of accessing the transaction filter (XF) and any false positives requiring accesses to the XADT. Rajwar et al. proposed the XF as an Bloom filter stored in memory, requiring multiple memory accesses for each cache line brought in and consuming a non-trivial amount of storage to ensure a low false positive rate. We are currently studying an alternate XF design where the XF state is carried with the cache blocks, using bits made available by re-encoding ECC at a larger granularity, as was done to hold the 21364’s directory [6]. Such an implementation requires no extra memory accesses, and, if bits are provided at the same granularity as data is stored in the XADT, then no false positives result from aliasing. Furthermore, without aliasing it is not necessary to maintain a count of the number of associated XADT entries; instead, an XF bit can be cleared whenever an XADT lookup fails to find a match for the associated block. The only challenge with this approach is storing the XF data (or some conservative approximation of it) when pages are swapped to disk, an area we are currently exploring.

IV. DEALING WITH CONFLICTS EFFICIENTLY

With large transactions, there is an increased potential for conflicts, because transactions are longer lived and they touch more data. Because the cost of aborting a transaction is also increased—more work will be thrown away—it is important to
efficiently handle conflicts. In this section, we demonstrate how transactions can be de-scheduled — rather than aborted — on a conflict and re-scheduled when the conflicting transaction commits. We then elaborate this mechanism to allow a thread to intentionally wait on an arbitrary number of data structures.

A conflict does not necessitate aborting a transaction, an observation made in previous transactional memory systems [16, 18] and earlier in database research [20]. 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 6. 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 T1 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., [11]). 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.

<table>
<thead>
<tr>
<th>access type</th>
<th>T1</th>
<th>T2</th>
<th>conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>read</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6. 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.

If T1 is a large transaction, T2 may be stalled for a significant time, unnecessarily occupying a processor core. In the first subsection, 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 an unavailable semaphore de-schedules a thread. In the second subsection, we demonstrate how this mechanism can be slightly generalized to enable simultaneous waiting on multiple conditions, in principle much like the select() system call.

A. De-scheduling Threads on a Conflict

In order to de-schedule a thread on a transaction conflict, we need to communicate a microarchitectural event up 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 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 using a bit (XSW_EXCEPT) in the XSW to indicate that a xact_completion exception should be raised when the transaction commits/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 7. 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 8; 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 or by using sequence numbers, but space limitations preclude a detailed discussion.

```c
asm linkage 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";} // 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();
}
```

Fig. 8. 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.

B. Implementing an Intentional Wait

In their software TM for Haskell, Harris et al. propose a particularly elegant primitive for waiting for events, called retry [8]. 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 9, 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.

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, and raises a retry_wake_up exception. This exception handler reads the task_struct field from the aborted transaction’s LTSS and wakes up the thread using try_to_wake_up(). Also, a potential race condition exists that requires adding a check to the code in Figure 8 to verify that the transaction is not waiting on a retrying transaction, before it calls schedule().

V. 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.

A. A Simple Example: Keeping Statistics

In Figure 10, 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 seemingly trivial, such statistics impact the scalability of existing hardware TMs [3]. 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. In Figure 10, 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.

B. 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

3This description is a slight over-simplification because of the fact that a single instruction may kill multiple transactions, due to multiple readers and x86’s allowance for unaligned memory accesses. Because of the difficulty of communicating an arbitrarily long list of aborted transactions in a single call to the retry_wake_up exception handler, we instead abort the transactions one at a time. If the VTM hardware has already aborted a transaction (T2) with a set XSW_RETRY bit and detects another one (T3) that also conflicts with the current instruction, it does not abort T3 or any subsequent transactions. Instead, it rolls back the current instruction and raises an exception, which inserts T2 into the current transaction’s (T1) wait list. We do not want to immediately wake up T2 because a race condition would exist between T2 re-acquiring the contended variable and completing a retry, and T1 actually writing the contended variable. Instead, upon aborting the last waiting transaction (T3), T1 completes the write then raises an exception to re-schedule not only T3, but any waiters with their XSW_RETRY bits set.
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 [17]; while pausing is, in some ways, a weaker primitive than open commit, it is sufficient for many applications and much easier to implement.

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 [4, 5, 12, 22] 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 11), 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 to perform the compensation, and an arbitrary amount of data (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.

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`.\(^5\)

Clearly, correctly writing paused regions with compensation is likely to 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. Furthermore, because they don’t provide atomicity with respect to the rest of the transaction, pausing and compensation are not appropriate in all circumstances; they are, however, a good fit for the requirements of dynamic allocation (as we show in the next subsection), because programs generally do not rely on which memory is allocated.

### C. Using Pause 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` [13]), the same issues are present even in advanced parallel memory allocators (e.g., Hoard [2]).

In Figure 12, 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.

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

**Fig. 12.** Example transaction that includes memory allocation and deallocation.

Here, we consider two implementations of `malloc`: the first is quite straightforward, 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 13. 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 existing 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 [24]. 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

---

\(^5\)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;
    } else {
        free_internal(ret_val);
    }
    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);
}

Fig. 13. 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).

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.

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 13. 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.

To avoid errors of this sort in general, we’ve modified the Linux kernel to kill unpauaed transactions in the system_call() interrupt vector.
VI. 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.

While the requirements for atomic actions that are short lived and small in memory footprint appear to be easily addressed with relatively minor modifications to hardware caches and the cache coherence protocol, as described in Herlihy and Moss’s landmark paper [10], the issues relating to large-memory-footprint and long-running transactions are significantly different. Because large transactions will not lead to increased concurrency if they observe significant contention, we expect they will be used only in circumstances of (at most) infrequent contention.

Instead, the main concerns regarding large transactions include how they can be supported with minimal overhead and in a manner that does not sacrifice performance isolation, how we can prevent losing a significant amount of work when a conflict occurs, and what primitives can provide a more powerful and/or easier-to-use programming model for programmers. To address these, we presented: 1) a hierarchical organization for storing the read and write sets for overflowed transactions that provides graceful scaling and good memory efficiency without introducing pointer chasing into the lookup process, 2) the first practical implementation of supporting out-of-core transaction commits, 3) an approach for enabling the operating system to de-schedule a hardware transaction until the condition upon which it is waiting is satisfied, and 4) the notion of transaction pausing and the adaptation of the concept of compensation to hardware transactions that increases the power of hardware transactions. In addition, our implementation shows that modifying an operating system to support TM is not too difficult; not counting our VTM data structure definitions and software primitives, we added about 500 lines of code (including blank lines and comments) to the Linux kernel for the functionality described in this paper.

Looking forward, a key area of research will be exploring how applications and the programming process changes with the advent of transactional memory. To this end, we intend to distribute our infrastructure to facilitate application development and programming studies.

REFERENCES


