Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory

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

Abstract

We demonstrate how fine-grained memory protection can be used in support of transactional memory systems: first showing how a software transactional memory system (STM) can be made strongly atomic by using memory protection on transactionally-held state, then showing how such a strongly-atomic STM can be used with a bounded hardware TM system to build a hybrid TM system in which zero-overhead hardware transactions may safely run concurrently with potentially-conflicting software transactions.

We experimentally demonstrate how this hybrid TM organization avoids the common-case overheads associated with previous hybrid TM proposals, achieving performance rivaling an unbounded HTM system without the hardware complexity of ensuring completion of arbitrary transactions in hardware. As part of our findings, we identify key policies regarding contention management within and across the hardware and software TM components that are key to achieving robust performance with a hybrid TM.

1. Introduction

Transactional memory (TM) has been proposed as a concurrency control mechanism for shared memory programs which presents a number of advantages over locks [16, 18]. TM relieves the programmer from tracking the association of shared data with locks, provides the concurrency of fine-grained locking without its programming complexity, and facilitates the composition of critical sections. TM's basic unit, the *transaction*, is a collection of instructions that must appear contiguously in the program's total memory order. Transactions are executed optimistically; a *TM system* detects and resolves conflicts between transactions, rolling back transactions if they cannot commit atomically.

A TM proposal can be characterized by three orthogonal attributes: the programming interface it provides, its performance, and its hardware requirements (discussed in more detail in Section 2). The ideal TM would provide clean semantics and a complete programming model, introduce no overhead to transactions, and require no special-purpose hardware support. While such an ideal TM is likely unobtainable because of the tension between these attributes in the implementation, we contend that existing proposals fall well short of the ideal in one or more of the attributes.

We believe that hybrid TMs represent the most compelling approach to building TMs, but that existing proposals have significant weaknesses. A hybrid TM executes most transactions using a simple hardware TM (HTM), but handles large, long-running, or otherwise uncommon transactions using a software TM (STM) [9]. This approach promises the performance of an HTM in the common case and permits arbitrary transactions without the complex hardware required by unbounded HTMs. It also permits much of the system's transactional semantics to be defined in software rather than hardware. However, existing hybrid TM proposals suffer from two notable drawbacks: they are subject to the non-intuitive semantics resulting from not detecting conflicts between transactional and non-transactional code, and the performance of their common-case hardware transactions is sacrificed to ensure that hardware transactions do not violate the atomicity of software transactions. In this work, we propose a new hybrid TM design that addresses these shortcomings of the previous work.

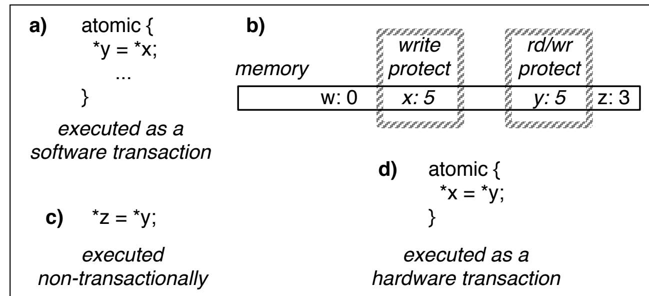


Figure 1. Using fine-grained memory protection to achieve a strongly-atomic, efficient hybrid TM: Software transactions (a) use hardware memory protection to *write protect* values that they read and *read/write protect* values that they write, as shown in (b). Accesses to these values by non-transactional execution (c) will raise a protection violation, preventing the transaction's atomicity from being violated. This same protection mechanism also prevents conflicting hardware transactions (d) from violating the atomicity of software transactions, *without the hardware transactions having to check STM metadata*; they can execute without software overhead.

In our proposal, the STM transactions protect the memory locations that they are reading and writing using a hardware fine-grained memory protection mechanism (*e.g.*, Blizzard [32], Mondriaan Memory Protection [37], iWatcher [40]), as shown in Figure 1. This prevents accesses from non-transactional code from violating the atomicity of software transactions, because a hardware fault will be raised before a conflicting non-transactional read or write completes.[1] In this way, we provide the STM with *strong atomicity* [4], which, as we discuss in Section 2.2, is a property that provides clean transactional semantics in a straightforward way.

[1] Software transactions disable these faults upon beginning and re-enable them upon commit.

This technique, however, also permits hardware transactions to detect conflicts with concurrently-executing software transactions **without slowing hardware transactions that do not conflict**. When a hardware transaction attempts to perform an access that conflicts with an in-flight software transaction, it receives a protection fault, permitting the hardware transaction to avoid the conflict by backing off or aborting. Since software transactions are required to protect their transactional data with hardware memory protection, hardware transactions can run at full speed because no software checks are required to detect conflicts with STM transactions. In this way, our proposal adopts a *pay-per-use* philosophy, where the costs of uncommon cases (such as cache overflows or I/O) only affect overall performance in proportion to their frequency.

Our approach also follows the precept that *hardware should provide primitives and not solutions* [38]. Our hybrid TM uses a minimum of special-purpose hardware, instead consisting of two hardware primitives — a best-effort hardware TM (Section 3.1) and fine-grained memory protection (Section 3.2) — that have many compelling applications unrelated to TM. We also avoid architecting in hardware the hybrid TM's semantics; instead, the STM is free to define (and evolve) its full feature set, of which the hardware TM accelerates a (likely common) subset. This approach is particularly compelling as it is in the dark corners of TM (*e.g.*, system calls, I/O, waiting, open nesting) where the desired semantics have yet to be determined, but where programs are likely to spend only a fraction of their time.

This paper makes the following contributions:

- We show how hardware memory protection enables low-overhead strong atomicity for STMs (Section 4.1).
- We demonstrate that this strong atomicity enables a hybrid TM system with zero-overhead hardware transactions that can execute concurrently with potentially-conflicting software transactions (Section 4.3) and characterize the performance of this hybrid TM (Section 5).
- We identify policies for managing contention between STM and HTM threads and switching transactions to software that achieve robust performance (Section 4.4).

We conclude (in Section 6) with a brief discussion of how this architecture naturally permits the introduction of richer TM semantics like transactional waiting and support for transactions that perform system calls and I/O.

2. Motivation

In this section, we discuss the three attributes of an ideal TM system mentioned above — high performance, a clean and complete programming model, and little dependence upon special-purpose hardware — and pitfalls encountered by prior TM proposals.

2.1. High performance

As the goal of multi-core programming is to increase program performance, many programmers will ignore transactions if they significantly underperform locks, even if they offer a cleaner programming model. For example, there has been little adoption of STMs because their significant single-thread overhead causes them to underperform lock-based code in any circumstance where lock contention is not the only bottleneck.

We strongly believe that, to achieve widespread use, uncontended transactions will need to incur little – if any – performance overhead relative to non-transactional execution. If transactions incur negligible single-thread overhead, programmers will be free to use them for concurrency control in almost any piece of code, without having to consider whether that code represents a scalability bottleneck.

2.2. A clean and complete programming model

To be accepted by programmers, TM should expose a programming model that is no more complex than locks. This poses a problem for bounded HTMs, for programmers should not have to reason about how their transactions fit into the cache geometry and the time quantum of all machines upon which they might run. Furthermore, the TM's semantics should be intuitive — however, some proposed TM systems have exposed non-intuitive semantics, two examples of which are shown in Figure 2.

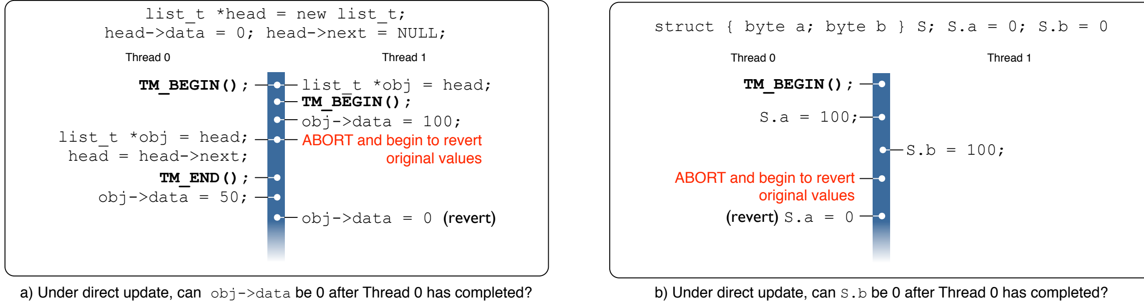
First, a common practice with locks is to *privatize* a shared object to one thread, typically by making all pointers to it inaccessible from other threads. This allows the newly-private object to be accessed outside of a critical section. Intuitively, privatization should work with transactions, but in some TM implementations it does not. Figure 2a, demonstrates how in some TM implementations, aborted transactions may still have accessed data that has been privatized by another transaction resulting in lost updates.

Second, there is the potential for lost writes due to *false conflicts* whenever the granularity for handling writes is larger than the minimum-sized write (a common property of TM systems) and conflicts between transactions and non-transactional code are not detected. Figure 2b demonstrates this problem in the context of neighboring accesses to a byte array, where a non-transactional write is lost when a transaction accessing a byte in the same transactional word aborts.

While these unintuitive behaviors can be addressed in a piecemeal fashion[2] they can also be addressed by making the TM system *strongly atomic*. Strong atomicity — requiring transactions to serialize with conflicting non-transactional (nonT) accesses — has been identified as a sufficient condition to avoid these and other problems [4, 14, 35, 24]. While most HTMs, which detect conflicts via coherence, are strongly atomic, most STM proposals are not, because doing so has required instrumenting non-transactional code, the software overheads of which can be significant, even with aggressive whole program analysis [35].

In addition, the boundary of the TM programming model is still a very active area of research, with many compelling opportunities to extend the TM paradigm beyond the basic multi-word compare-and-swap. Two recent examples are transactional waiting and side-effecting transactions. Transactional waiting (introduced as the `retry` primitive by Harris *et al.* [15]) can eliminate *lost wakeup* bugs, but poses serious challenges for HTMs. Side-effecting operations such as I/O are generally unsupported within hardware or software transactions, requiring critical sections that perform such operations to use locks. Transactions which support

[2] Privatization can be supported by stalling commit until all conflicting transactions complete the abort process [33], and false conflicts are avoided by preventing data within the same transactional word from being accessed both inside and outside of transactions (but that is easier promised than guaranteed).



a) Under direct update, can `obj->data` be 0 after Thread 0 has completed?

b) Under direct update, can `S.b` be 0 after Thread 0 has completed?

Figure 2. Two transactional examples that can have surprising outcomes: (a) Privatization: the transaction in thread 0 successfully commits, privatizing `obj` and aborting the transaction in thread 1, but delayed rollback of thread 1's transaction can discard non-transactional writes by thread 0. (b) False Conflict: The transaction in a word-based STM logs the word containing both `S.a` *and* `S.b` before writing to `S.a`. During the transaction, thread 1 updates `S.b`; a later abort of the transaction in thread 0 will discard the update to `S.b`. These examples affect direct-updating TMs, but similar examples affect deferred-updating systems.

side-effecting operations [3, 23, 26] could permit transactions to be the single synchronization mechanism to be used for all isolated data accesses. As rich transactional programming models are still an active area of research, it is desirable that they are not precluded by a TM system – as would be the case if a TM's semantics were architected into hardware.

2.3. The Role of Hardware

Given the limitations of pure STMs, most of the TM community have accepted that some sort of hardware support is necessary to make TM viable. To this end, many *unbounded* HTM architectures, which handle transactions completely in hardware, have been proposed [1, 8, 29, 25].

We see three problems with these approaches. First, much of the hardware complexity (relative to a *bounded* TM proposal) is dedicated to handling cases that are expected to be relatively infrequent or not performance-critical. Second, these systems violate the precept that hardware should provide primitives and not solutions [38]; it seems particularly reckless to architect the TM semantics into hardware when it is not yet clear what the desired semantics are and while potentially different language environments may require slightly different semantics. Third, these approaches introduce significant hardware additions that are specific to transactional programming, whose importance has yet to be quantified.

We believe that (at least for the foreseeable future) hardware support for TMs should avoid fixing TM policy in hardware and minimize the amount of special-purpose TM hardware. For this reason, we explore a *primitives*-based approach to TM hardware, in which the bulk of the hardware required by the TM is also useful in contexts beyond transactional memory.

3. Hardware Primitives

In this section, we briefly introduce the two hardware components used in our proposed system, both of which are derived from previous work: a "best effort" hardware TM and fine-grained memory protection hardware.

3.1. BTM: a "best-effort" Hardware TM

BTM is a hardware-based *best-effort* TM system that provides functionality similar to the original TM proposal by Herlihy and Moss [16]. It supports transactions that fit entirely in the transactional cache (L1 in our case), do not raise exceptions or receive interrupts (including timer interrupts), require only flattened nesting, and perform no I/O. These limitations notwithstanding, we find that a significant majority of the dynamic transactions seen in our benchmarks are able to execute completely in BTM.

BTM extends a write-back L1 cache to support speculatively committed loads and stores in much the same way that has been proposed for speculative multithreading [11], speculative lock elision [28], and many other hardware TM proposals (*e.g.*, [1, 13]). BTM operates at cache-block granularity, extending every L1 cache block to include speculatively-read (SR) and speculatively-written (SW) bits. As usual, appropriate coherence permission must be acquired before completing transactional memory operations. When a transactional load commits, it sets its block's SR bit. Before a transactional store commits, the cache makes sure that the to-be-written cache block is clean (writing a dirty block to the next lower level of the cache hierarchy) before completing the store and setting the block's SW bit. If a block with an SR or SW bit set is evicted from the cache, the transaction is aborted; all transactionally-written lines are invalidated, and all SR/SW bits are flash-cleared. As we discuss in Section 4.4, BTM uses an age-based contention management policy.

This speculative execution hardware is exposed to software through a simple interface (Table 1) which permits high-performance implementations. Software specifies the beginning of a transaction with a `btm_begin` instruction, which specifies an abort PC. When a (non-nested) `btm_begin` is executed, a register checkpoint is taken; if the transaction is aborted, this checkpoint is restored and control is vectored to the abort PC. Software can specify that a transaction can be committed or aborted with the `btm_end` and `btm_abort` instructions. A (non-nested) transaction is committed by flash clearing all SR and SW bits and discarding the register checkpoint.

`btm_begin imm32`	Begin a BTM transaction with abort address given in immediate `<imm32>`
`btm_end`	End a BTM transaction
`btm_abort`	Abort a BTM transaction
`btm_mov reg, txr`	Copy transactional status register `<txr>` to register `<reg>`

Table 1. The BTM ISA extension

`set_ufo_bits addr, bits`	set the UFO bits for the memory line containing <addr> to <bits>
`add_ufo_bits addr, bits`	OR <bits> into the UFO bits for the memory line containing <addr>
`read_ufo_bits reg, addr`	return the UFO bits for the memory line containing <addr> in register <reg>
`enable_ufo/disable_ufo`	turn on/off UFO faults on this processor

Table 2. UFO ISA extensions for manipulating UFO bits and enabling/disabling UFO faults.

`ustm_begin()`	Begin a USTM transaction (checkpoint regs, clear log, get seq number, set transaction state)
`ustm_end()`	End a USTM transaction (release ownership, discard checkpoint)
`ustm_abort()`	Abort a USTM transaction (undo writes, release ownership, restore chkpt)
`ustm_read_barrier(void *addr)`	Acquire read permission for <addr>
`ustm_write_barrier(void *addr)`	Acquire write permission for <addr>

Table 3. The USTM API: All functions return `void`.

In addition, BTM provides access to status registers that record whether a transaction is executing, its current nesting depth and the reason for the last transaction abort. Abort reasons include cache set overflow, explicit abort, interrupt, illegal operation, conflict, exception, system call, uncacheable access, page fault, and hardware nesting depth overflow. When an address is associated with the event (such as the PC of an explicit abort or the page fault address) it is also recorded so it can be made available to software.

The hardware atomicity provided by BTM is useful not only for implementing TM systems. The same hardware can be used for implementing speculative lock elision (SLE) [28, 31], where lock-based critical sections are speculatively executed as hardware transactions. More recently, hardware atomicity has been proposed as a means to facilitate speculative software optimizations [27].

3.2. UFO: User-mode Fine-grained Hardware Memory Protection

Fine-grained memory protection mechanisms refine the page-granularity protection supported by virtual memory to granularities smaller than a page [32, 37, 40]. For this work, we consider an iWatcher-style [40] mechanism that permits applications to install access permissions at a cache block granularity and achieves zero execution overhead checking of these access permissions in the common case when no faults occur. In Appendix A, we describe an enhanced implementation of iWatcher, called UFO, which retains iWatcher's modest hardware complexity, but also provides multiprocessor-safety and supports arbitrarily large regions, context switching, and swapping.

The implementation provides two *User Fault-On (UFO) bits* per cache line: a fault-on-read bit and a fault-on-write bit. The protection bits and the data with which they are associated move together throughout the whole memory hierarchy: caches, main memory, and the swap file. User-mode instructions are provided for setting and reading the UFO bits (Table 2), meaning that protection can be added/removed with low overhead but that this protection is not suitable for security applications. Setting UFO bits requires exclusive coherence permission to ensure that all copies of a block have consistent permissions.

When a processor performs an access not permitted by the block's current UFO bit settings, a UFO fault is raised. The processor's exception handling mechanism invokes a software handler registered by the application. As with page faults, the faulting address can be read from a special-purpose register, so that the UFO fault handler can take the appropriate action. UFO provides the ability to disable UFO faults on a per-thread basis, in the same way that interrupts can be disabled; `enable_ufo`, `disable_ufo` respectively set and clear a *UFO enable* bit, which is part of the thread's context.

In addition to the debugging applications for which iWatcher was proposed [40], low-overhead fine-grained memory protection enables a broad array of applications including speculative value specialization optimizations [34], concurrent garbage collection [2], and efficiently supporting self-modifying code in binary translators/optimizers. Like BTM, UFO is a multi-purpose mechanism.

4. System Organization

In this section, we introduce a novel hybrid TM implementation that addresses the concerns raised in Section 2. This hybrid TM — comprised of the two general-purpose hardware mechanisms described in the previous section — executes most transactions directly in BTM with no instrumentation overhead, provides strong atomicity to yield clean TM semantics, which can be extended (in software) to support a rich transactional programming model. The section is organized around our three key contributions: we show how fine-grained memory protection can be used to build a strongly-atomic STM (Section 4.1), we demonstrate how using such an STM enables a hybrid with zero-overhead HTM transactions (Section 4.3), and we highlight the policies necessary to achieve performance comparable to an unbounded HTM using such a hybrid (Section 4.4).

4.1. USTM: Building a Strongly-Atomic STM

We show how fine-grained memory protection can be used by an STM to provide strong atomicity at low overhead. As background, we first describe the design of the eager-versioning, eager-conflict detection cache-block granularity STM that we are using in this work. We then show how we extend it to be strongly atomic.

The UFO STM, or USTM, is an STM library for C/C++. It implements transactions using the API shown in Table 3. Notably, calls to `ustm_read_barrier`[3] and `ustm_write_barrier` should be inserted (by a compiler) before read and write accesses to shared variables, allowing the STM to acquire permission and perform logging as necessary *before* the operation is performed: like BTM, USTM detects conflicts eagerly.

USTM relies internally on two data structures: an ownership table (`otable`) shared between all transactional threads and a per-thread transactional status structure (which includes a transaction log). The shared `otable`, as shown in Figure 3, contains a record for each cache line currently read or written by a USTM transaction (making USTM, like BTM, cache-line granularity.) The `otable` is logically organized as a chained hash table, with each entry containing: 1) a tag to identify which cache line is specified, 2) the permissions held for the line, and 3) the set of transactions that have

[3] The use of the term "barrier" derives from the garbage collection literature and not the instructions required by weak memory consistency models.

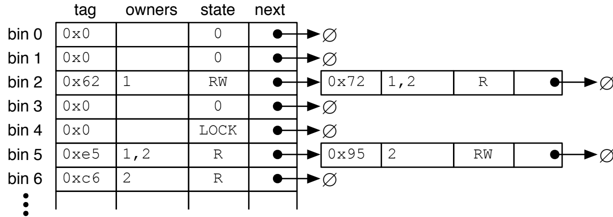


Figure 3. The USTM Shared Ownership Table (otable) is organized as a chained hash. Entries contain a tag, owner list and a state. Two special entries exist: *null* entries, which are not in use, and *locked* entries, which can only be examined or altered by the thread that holds the lock.

access to it. When it cannot be updated with a single atomic operation, the head entry of a chain can be put into *locked* state to provide isolated access to the chain to handle races between threads updating the otable. In Figure 3, hash bins 0, 1, and 3 are unused, hash bin 4 is *locked*, and bins 2 and 5 have chains; realistic implementations generally have at least tens of thousands of entries to minimize aliasing.

The use of the otable is demonstrated by the ustm_write_barrier pseudocode given in Algorithm 1. This code optimizes for the common cases when chain length is zero or one. If the requester has an existing entry for the block with write permission, no action is necessary. Inserting an entry when none is present or upgrading an existing entry when the requester has sole read access can be done with a compare&swap. If one or more other transactions currently own this entry of the otable, conflict resolution is invoked as described below. If an entry is present, but its tag doesn't match, the whole chain is locked while it is searched for a matching entry; if none is found, an entry is inserted at the head of the chain. Once ownership has been obtained, the cache block's address and current values are logged (eager versioning) and the write is performed to memory. The code for ustm_read_barrier is similar in structure, except multiple readers are permitted and only the block's address is logged.

When a USTM transaction T aborts or commits, it removes its entries from the otable. For each entry in T's log, the corresponding entry in the otable is removed through a process similar to Algorithm 1: entries owned solely by T are removed, and T is removed from the *owners* sets of any shared read-only entries. If T is aborting, it restores to memory the logged values of cache blocks to which it wrote.

The conflict resolution policy we employ for USTM is age-based. An STM transaction that conflicts with another STM transaction stalls if it is younger than one of the transactions it conflicts with, otherwise it aborts the conflicting transactions. Our current STM implementation is blocking, relying on transactions to unwind themselves on an abort. As a result, after a transaction notifies a conflictor that it should abort, it waits, monitoring the transaction status, until the abort process is complete before it continues. In this way, USTM avoids contention on the otable and possible livelock. Likewise, when a transaction is aborted, it waits until the transaction that aborted it has retired before reissuing, also to avoid otable contention and livelock.

4.2. Making USTM Strongly Atomic

USTM, as described in Section 4.1, is not strongly atomic. The key to making USTM strongly atomic is to install memory protection for transactionally-accessed cache blocks whenever otable entries are created or upgraded. Specifically, fault-on-write protection is installed by ustm_read_barriers, and both fault-on-read **and** fault-on-write protection is installed by ustm_write_barriers. To prevent USTM transactions from receiving protection faults for their transactional data, threads disable UFO faults at the beginning of USTM transactions and re-eanble them at commit.[4]

An example of the changes necessary to extend USTM to use UFO is given in Algorithm 2, which replaces lines 5 and 6 of Algorithm 1. Note that we ensure the atomicity of inserting the otable entry and setting the UFO bits by locking the otable chain during the insertion. Doing so prevents races between threads inserting and removing an otable entry from leaving the UFO bits in a state inconsistent with the otable. Other places in the STM code that insert, upgrade (from read to write permission) or remove otable entries must set, upgrade, and clear UFO bits, respectively, in the same way.

The virtue of this approach to strong atomicity is that there is minimal additional execution overhead for the STM and no overhead for non-transactional code in the common case of no conflicts. When a conflict does occur, the faulting non-transactional thread vectors to a fault handler — registered by the STM before the first transaction executes — which can stall the non-transactional access or abort the conflicting transaction, based on a software-defined contention management policy.

4.3. Bringing It Together: The UFO Hybrid

The goal of a hybrid TM [9], which composes a bounded HTM with an STM, is to achieve HTM performance for most transactions, but to support large transactions without the hardware complexity of an unbounded HTM. Hybrid TMs can be implemented by organizing the transactional code as shown in Figure 4. The new component in this structure is the *abort handler*, which decides after a transaction is tried and fails in the HTM whether to retry the transaction again in hardware or to failover in software. Our abort handler is described in Section 4.3.1.

The other additional requirement and the traditional challenge of hybrid TMs is to prevent HTM transactions from violating STM atomicity,[5] which previous hybrid designs solve in ways that negatively impact hardware transactions (as discussed and demonstrated in Section 5). This challenge, however, is easily surmounted in a hybrid TM built on the strongly-atomic USTM. Because USTM provides strong atomicity via fine-grained memory protection, accesses by hardware transactions are prevented from violating STM

[4] This does not affect correct operation of the STM, as conflict detection between USTM transactions occurs as described in Section 4.1.

[5] Conflicts between HTM threads are detected by the HTM; those between STM threads are detected by the STM. The HTM's isolation cannot be broken by the STM because the HTM is strongly atomic; if, for example, a STM transaction tries to write a variable read by the HTM, the request will stall (if *nack*ed) or the HTM transaction will be aborted. Without a strongly atomic STM, however, nothing prevents an HTM transaction from observing the writes of, or writing to a variable previously read by, an uncommitted STM transaction.

1: **procedure** USTM_WRITE_BARRIER($trans$, $addr$)
2: $index \leftarrow$ GET_INDEX($addr$) ; $tag \leftarrow$ GET_TAG($addr$)
3: $entry_state \leftarrow$ MAKE_ENTRY_STATE($trans$, $addr$, WRITE) ▷ we hope to place this new entry in the `otable`
4: $o \leftarrow otable[index]$; ▷ read the head of the appropriate `otable` chain
5: **if** o = 0 **then** ▷ ownership table entry is unowned
6: COMPARE&SWAP($otable[index]$, 0, $entry_state$) **return** ▷ ownership acquired
7: **else if** LOCKED(o) **then** BACKOFF( ); **goto** 4 ▷ ownership table entry is locked, backoff and try again
8: **else if** TAG(o) != tag **then** ▷ is there a tag mismatch?
9: LOCK_ROW($index$,o); HANDLE_CHAIN($index$, tag, $trans$, $entry_state$); UNLOCK_ROW($index$) ▷ repeat process on chain
10: **else if** OWNERS(o) = $trans$ **then** ▷ $trans$ is the sole owner of this line
11: **if** STATE(o) != WRITE **then** ▷ if $trans$ already the only writer, do nothing
12: COMPARE&SWAP($otable[index]$, o, $entry_state$) ▷ if $trans$ the only reader, become the only writer
13: **else** RESOLVE_CONFLICT($index$, $trans$, o); **goto** 4 ▷ either stall or kill $owners$ and wait until $owners$ = 0
14: **return**

Algorithm 1: USTM_WRITE_BARRIER($trans$, $addr$) checks the `otable` for conflicts, then, if none is found, acquires write ownership for $trans$ on $addr$. If any COMPARE&SWAP or LOCK_ROW() fails, we repeat the algorithm from line 4.

5: **if** o = 0 **then** ▷ ownership table entry is unowned
6: LOCK_ROW($index$, 0) ▷ lock this `otable` row
7: SET_UFO_BITS($addr$, READ | WRITE) ▷ other threads should Fault-on Read and Fault-on Write
8: $otable[index]$ = $entry_state$ ▷ acquires ownership and releases lock
9: **return** ▷ ownership acquired

Algorithm 2: Fragment of USTM_WRITE_BARRIER($trans$, $addr$) showing how strong atomicity is conferred on USTM writes. Appropriate UFO bits are set when `otable` entries are inserted.

```
// A hybrid transaction which first attempts to execute
// in BTM, but can fail over to USTM.
xact:
  BTM_BEGIN( &&fail );
    // transaction body
  BTM_END();
  goto cont;
fail:
  // Resolve faults if possible.  If faults seem
  // resolved, goto xact; otherwise flow through
  BTM_ABORT_HANDLER( xact );
  USTM_BEGIN();
    // transaction body with ustm_reads and ustm_writes
  USTM_END();
cont:;
```

Figure 4. Hybrid TM code. If the transaction cannot succeed in BTM, even after BTM's abort handler is invoked, then it *fails over* to USTM.

atomicity the same way as non-transactional code is.[6] Notably, this approach adds no execution overhead to hardware transactions — the common case — even when concurrently executing with STM transactions.

One undesirable interaction between UFO and BTM, resulting from their mutual reliance on the underlying coherence protocol, bears mentioning. As exclusive coherence permission is required to set UFO bits (in order to keep them coherent), actions by the STM can cause inflight BTM transactions to abort. As we discuss in Section 4.4, we are not concerned when such an abort results from a true conflict, because it is reasonable to prioritize the STM transactions over the HTM transactions. Rather, the concern comes from the potential for false conflicts between two transactions reading the same line. USTM read barriers set the fault-on-write bit for the line in question, which will kill BTM transactions that have that block in their read set. Our results in Section 5.4 suggest that this is not a substantial problem. Were this interaction to lead to a significant loss of performance, it could be addressed by changing the coherence protocol to permit setting UFO bits in the `owner` state (as previously proposed [5]) or by lazily clearing UFO bits for read-mostly data.

[6]Unlike STM transactions, HTM transactions do *not* disable UFO faults.

4.3.1. The BTM Abort Handler

When a BTM transaction fails, control is transferred to the *abort PC* provided to the `btm_begin` instruction. In our hybrid TM implementations, this address vectors to an *abort handler* that decides whether the transaction should be re-tried in BTM or should *fail over* to the STM. This abort handler (Algorithm 3) tries to complete as many transactions as possible in the HTM, while quickly failing over to software for transactions that will end up completing there. It manages these conflicting goals by using the reason that the transaction aborted to categorize it into one of three classes: transactions likely to fail if tried again in hardware, transactions that should be retried in hardware, and transactions that can be retried in hardware after performing a software action.

Four conditions nearly guarantee that a transaction will abort again if it is re-tried in hardware: cache overflow, system call invocation, performing I/O, and incurring non-page fault exceptions. These transactions are immediately failed over to software — that is, re-tried as software transactions — as they represent the uncommon cases for which we are relying on the increased capabilities of the STM.

Some transactions are aborted due to conditions that are unlikely to repeat. For example, most transactions aborted by interrupts will complete when retried in hardware after the thread is re-scheduled. Similarly, when a transaction is aborted due to contention with another transaction, we generally want to retry the transaction in hardware because the slower execution of software transactions will tend to aggravate the contention. To mitigate contention, we implement an exponential back-off scheme in the abort handler. As indicated in Algorithm 3, our BTM implementation keeps track of the number of aborts by interrupts and conflicts (counting up to 7 of

```
1:  procedure BTM_ABORT_HANDLER(xact_label)
2:      abort_reason ← GET_ABORT_REASON( )
3:      if abort_reason = HW_CONFLICT then
4:          if MULTIPLE_CONFLICTS( ) then EXPONENTIAL_BACKOFF( )
5:          goto xact_label
6:      if abort_reason = INTERRUPT then
7:          if MULTIPLE_INTERRUPTS( ) then goto fallthrough
8:          else goto xact_label
9:      if abort_reason = SET_OVERFLOW then goto fallthrough
10:     if abort_reason = PAGE_FAULT then
11:         FIX_PAGE_FAULT( ) ; goto xact_label
12:     if abort_reason = EXCEPTION then
13:         if FIX_EXCEPTION( ) = EXCEPTION_FIXED then
14:             goto xact_label
15: fallthrough:
```

Algorithm 3: BTM_ABORT_HANDLER(*xact_label*), inlined at every transaction, attempts to resolve any problems preventing successful transaction completion in the HTM. Only if it unlikely that the transaction will succeed will this routine fallthrough to execute the STM version of the transaction.

each) since the last committed transaction, so that we can fail over to software and invoke backoff appropriately.

Finally, two abort conditions cause repeatable failure of hardware transactions, but can be addressed by the abort handler itself before restarting the transaction: missing register exceptions and page faults. The x86 ISA permits operating systems to lazily swap in the floating point registers by tracking the presence of the FP/SSE registers and raising an exception on their use when they are not present. If the transaction was aborted due to an `SSE device not available` exception (as is observed in the benchmark `kmeans`), the abort handler executes an SSE `nop`, provoking the fault, before retrying the transaction in hardware. In Algorithm 3, this is performed in FIX_EXCEPTION().

Similarly, when a transaction performs an access that generates a page fault, the transaction is immediately aborted, leaving the fault unhandled. The faulting address and access type, however, are stored in a pair of transactional status registers. The abort handler can perform an appropriate access to the faulting address,[7] forcing the fault to be handled outside of a transaction, before restarting the transaction in hardware.

4.4. Contention Management in a Hybrid TM

As demonstrated by previous work [6, 17], how a TM system responds to contention can have a first-order impact on how it performs. While those works studied contention management in pure HTM and pure STM contexts, respectively, our development of the UFO hybrid TM has led us to study contention management in the context of hybrid TMs. Following experimentation with several different policies, we have identified the following principles for handling conflicts and deciding when to retry a transaction in the HTM or the STM. We provide sensitivity results in Section 5.4 to support these assertions.

First, *there appears to be no substitute for having a good contention management policy in hardware*. As the need to implement contention management as part of an HTM introduces hardware complexity, we explored naïve hardware contention management policies which guaranteed forward progress by eventually failing over to the STM (where implementing contention management is straightforward). We found, however, that in regions of high contention such an approach often performed worse than the STM by itself. In fact, we found that any significant simplification in the HTM contention management policy yielded a first-order drop in performance. The policy that we implemented, like LogTM [25], uses transaction age in contention management. Unlike LogTM, which implements "requester stalls" and uses transaction age to detect deadlock causing cycles, we perform age-ordered conflict resolution for every request in the HTM: if the requester is older than a block's current owner, the block is taken and the current owner is aborted. If the current owner is older, the requester is nacked and requests again after 20 cycles.

Second, *it is important to only execute a transaction in the STM if doing so is required*. In particular, contention should not be a reason to fail over to software, because the STM's overhead will increase the transaction's duration, thereby holding contended variables longer, increasing contention. Policies that retry conflict-aborted HTM transactions as STM transactions are metastable; the slightest bit of contention can cause a chain reaction that throws all contending transactions into software.

Third, *there is little potential benefit to dynamically prioritizing STM transactions with respect to HTM transactions*. As the STM primarily runs long-running, large-footprint transactions (that have already failed to execute in the HTM), they are generally older than any hardware transactions they conflict with. In our experiments, the STM transaction is older in more than 99% of such conflicts. As a result, we statically prioritize STM transactions over HTM transactions, which is also the simplest policy to implement.

5. Performance Analysis

In this section, we characterize the performance of the UFO hybrid TM, demonstrating that it is a compelling alternative to previously proposed hybrid TMs and that it achieves performance comparable to pure (unbounded) HTM systems, which must guarantee the forward progress of all transactions in hardware. Specifically, our experiments compare the UFO hybrid to an unbounded HTM, three STMs, and two previously proposed hybrid TMs which we describe below: HyTM and Phased TM (PhTM). To facilitate comparison between these TM schemes, wherever possible we use the same building blocks: USTM (Section 4.1) and BTM (Section 3.1).

We give results from three STMs: USTM without strong atomicity, USTM with UFO-based strong atomicity to show the overhead of using memory protection for strong atomicity, and TL2, to link our performance with previously published results [10, 24]. For all but the unbounded HTM configurations, hardware transactions are limited to those that can fit in the L1 data cache.

For the unbounded HTM system that we model, we use the BTM model,[8] except the memory footprint of a transaction is not limited. In this way, our unbounded HTM is idealized with respect to actual pure HTM proposals (*e.g.*, it can flash clear on an abort, where

[7] If a write, we first read the value then atomically update it to itself using a compare-and-swap operation, repeating this process until the CAS succeeds.

[8] While the code for the unbounded HTM experiments does not include STM-targeted versions of the transactions, it does include a simplified abort handler necessary to make forward progress in the presence of page faults and `SSE device not available` exceptions (as described in Section 4.3.1).

LogTM uses a software rollback mechanism), meaning our results for the unbounded case may be optimistic with respect to what is implementable.

HyTM [9] addresses the challenge of detecting HTM/STM conflicts by burdening hardware transactions with the responsibility of checking the STM metadata to ensure that they are not violating the atomicity of STM transactions. The code of HyTM's hardware transactions are instrumented with read and write barriers, much like its software transactions, but the code is substantially simpler. The barriers perform the same `otable` lookup, but merely inspect whether a conflicting record is present. If a conflicting record is present, the transaction explicitly aborts and retries again in hardware. The primary drawback of adding these checks is the execution overhead they introduce into the hardware transactions. In addition, our implementation, like the original [9], reads `otable` entries transactionally, creating the potential for false conflicts when unrelated STM accesses alias to the same `otable` rows previously read by HTM transactions.[9] Furthermore, these transactional reads of the `otable` inflate HyTM's transactional footprint, sometimes yielding extra cache set overflows.

PhTM [19] avoids instrumenting hardware transactions by precluding HTM and STM transactions from executing concurrently. The system maintains a counter of the number of STM transactions currently executing, which is read at the beginning of each HTM transaction. If the counter is non-zero when read or is updated during the HTM transaction's execution, the HTM transaction aborts. The major drawback of this approach is that if one hardware transaction has to fail over to software, it takes the rest of the concurrent hardware transactions — even those which could have completed in hardware — with it. To prevent an STM phase from lasting perpetually, PhTM maintains a second counter which tracks the number of running transactions that failed over to software due to a condition the HTM does not support (*e.g.*, cache overflow, exception). As long as this second counter is non-zero, any new transaction will commence in the STM. When this second counter reaches zero, PhTM starts the shift back to an HTM phase by stalling transactions rather than starting them in the STM. When the first counter reaches zero, the last STM transaction has completed, and the waiting transactions can commence as HTM transactions.

5.1. Experimental Method

In our experiments, we model the hardware in an x86 full-system, timing-first [22], execution-driven simulator, built using Virtutech Simics [20] and incorporating the x86 instruction decoder from PTLsim [39] and the Ruby MOESI-directory memory system [21]. The simulated system includes a modified Linux kernel that provides support for saving and restoring UFO bits when physical memory pages are swapped to and from disk. The details of the simulated system are provided in Table 4.

We used the STAMP benchmark suite [24], which consists of three programs that exhibit a diversity of transaction construction and interaction. `kmeans` implements a clustering algorithm and consists largely of small transactions. `vacation` is a reservation-scheduling system that includes large, long-running transactions that sometimes overflow the cache. `genome` is a gene sequencing application whose transactions periodically overflow the cache and exhibit significant contention at a central bottleneck. As in previous work [24], we show data for both high- and low-contention configurations of `kmeans` and `vacation`.

Processor frequency	4.0 GHz
Fetch/Decode width	3 x86-instructions
Rename/Issue/Retire width	4/4/4
Instruction window size	128
Branch predictor	4K gshare, 8-bit history
Indirect target predictor	256-entry BTB, 32-entry RAS
L1 Data Cache	64 KB, 8-way, 3 cycle hit
L2 Unified Cache	512 KB, 8-way, 11 cycle hit
L1 Cache Line size	64-bytes
L2 Cache Line size	64-bytes
Physical memory	512 MB, 100 cycle latency
Coherence protocol	MOESI directory
Operating system	Red Hat Enterprise Linux 4
Linux kernel	Modified 2.6.23.9 kernel
USTM `otable` size	65,536 entries

Table 4. Simulation parameters

5.2. Results

Our performance results are shown in Figure 5, plotted as speedups relative to sequential execution. We find that even when some transactions overflow to software, the UFO hybrid can achieve performance close to that of an unbounded HTM solution while retaining the ability to fall back to an STM for cases that hardware designers choose not to handle. In addition, we observe that the UFO hybrid consistently outperforms (or performs equally well as) both the HyTM and PhTM hybrids. Figure 6 shows the reasons that hardware transactions aborted in the benchmarks.[10] Finally, it can be seen that making USTM strongly atomic (via adding UFO bit operations) adds little overhead to the baseline USTM, which performs similarly to TL2 [10] in all but `kmeans`. In `kmeans` we observe that the *winner* of a transactional conflict frequently has to stall a non-trivial amount of time because our current USTM implementation relies on aborted transactions to recognize (as part of STM barriers) that they have been killed and release their `otable` entries and that in `kmeans` transactions execute for long periods without STM barriers. We believe that our ongoing work to make USTM non-blocking will address this effect.

As noted previously, `kmeans` gives hybrids few reasons for transactions to fail over to software. Almost all of the aborts in kmeans are due to contention or other recoverable reasons. As a result, performance of all of the hybrids closely parallels that of unbounded hardware in both the high- and low-contention runs because almost all transactions commit in hardware. Specifically, there is less than a 1% difference in performance between unbounded HTM, the UFO hybrid, and PhTM. The barriers in HyTM cause its performance to lag the rest by 10-20%; in `kmeans` the barrier overhead is small because of its low density of STM barriers.

In contrast, `vacation` presents a significant challenge for hybrid TMs, since it consists of long-running, large-memory-footprint transactions. The hybrid TMs actually perform better in the high-contention case because the low-contention version has more trans-

[9] Such false conflicts could be eliminated by extending BTM to support non-transactional loads for use by the barrier code.

[10] STAMP benchmarks feature no explicit aborts, so any aborts seen in Figure 6 are due to the operation of PhTM and HyTM.

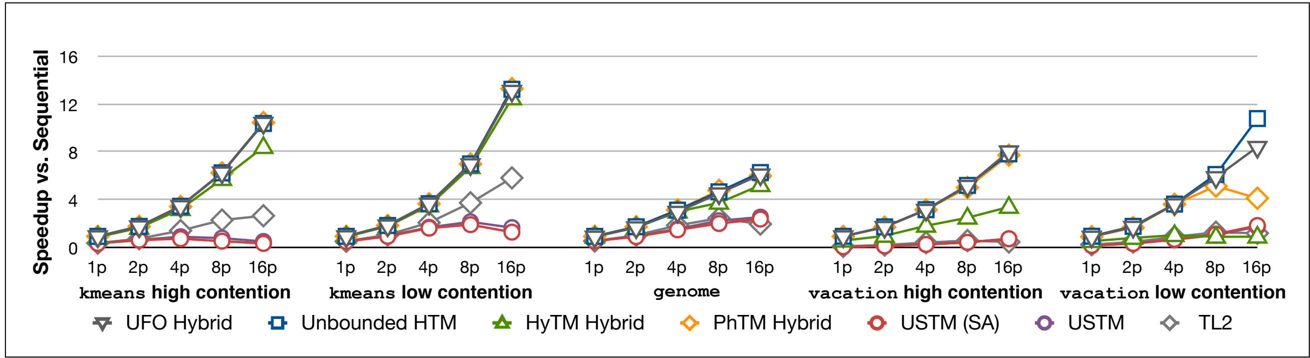


Figure 5. STAMP Performance of several TM schemes. Performance is normalized to that of sequential execution. USTM (SA) is USTM with strong atomicity via UFO enabled.

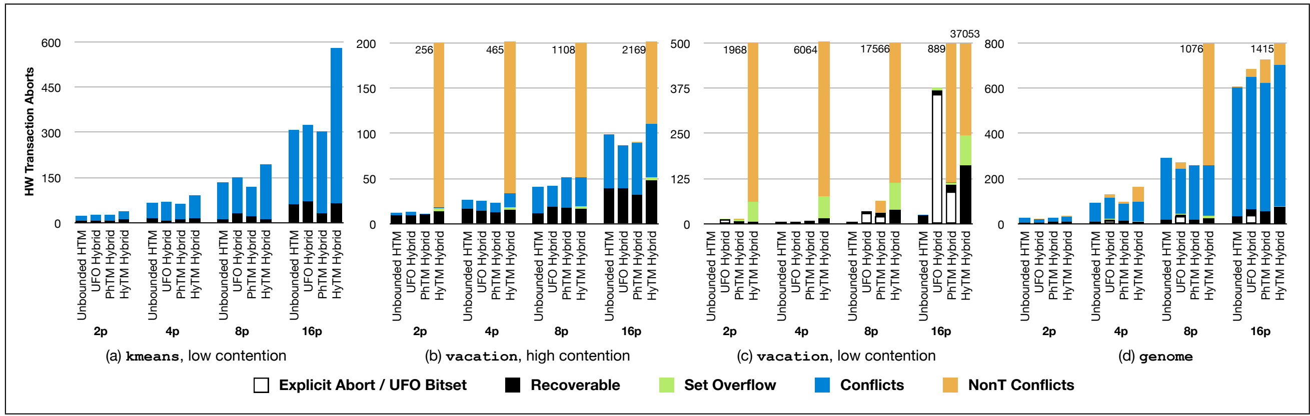


Figure 6. Hardware Abort Reasons. Explicit aborts only occur in PhTM and HyTM, and UFO bit sets only occur in the UFO hybrid, so they are shown with the same bar. Recoverable aborts include interrupts, page faults, and `SSE device not available` exceptions. **NonT**ransactional conflicts result from STM writes to data held by hardware transactions. `kmeans` high contention (not shown) is qualitatively similar to `kmeans` low contention.

actions that overflow the cache.[11] The effects of a non-trivial number of large transactions failing over to software can be seen in Figure 6c: the UFO hybrid incurs more transactions killed by *UFO bit sets* (re-tried in hardware); HyTM receives more *nonT conflicts* on previously-read `otable` entries, as described earlier in this section; and PhTM generates more *explicit aborts* (due to trying to start hardware transactions while software transactions are in-flight) and *nonT conflicts* (on the software-transactions-in-flight counter, due to software transactions starting while hardware transactions are in-flight). This propensity to overflow the cache, combined with the long-running nature of `vacation`'s transactions (which run still longer in software) particularly affects PhTM, whose performance actually begins to degrade as the number of threads (and therefore the chance that one of them is running a software transaction) increases, as shown in Figure 5.

The performance variation seen among the hybrids in `vacation` low contention is largely due to the set overflows; when the transactional cache is made sufficiently large to hold all `vacation` low contention's transactions, the hybrids perform (relative to the unbounded HTM) almost exactly as they do for `vacation` high contention. However, only the UFO hybrid is capable of simultaneously deploying no-overhead hardware transactions while permitting only those transactions that really need to fail over to software to do so. This ability gives it a marked advantage.

The larger working set of `vacation` also reveals another weakness of HyTM: in Figure 6c, we see that HyTM suffers a notably greater number of cache *set overflows* than the other hybrids. This effect is due to accessed `otable` entries competing with the transaction's data for space in the transactional cache. These `otable` accesses do not only result in additional cache overflows, but also drastically increase the number of *nonT conflicts* that abort hardware transactions, both of which negatively impact performance (as seen in Figure 5.) Furthermore, the longer-running HyTM hardware transactions tend to run into timer interrupts more often, yielding a greater number of recoverable aborts as well.

`genome` exhibits a high-contention initialization phase, in which elements are inserted in sorted order into a shared linked list — a data structure not well suited for concurrent writes by transactions. While the code could be modified to alleviate this contention, it serves as a challenging test case for TM implementations. When a transaction writes the list, it kills any transactions (which are necessarily younger because we resolve conflicts with age ordering) that have read the written part of the list. It is this type of code

[11] The benchmark configuration dictates what fraction of a fixed-sized database is accessed, with low contention accessing more of the database, leading to larger working sets.

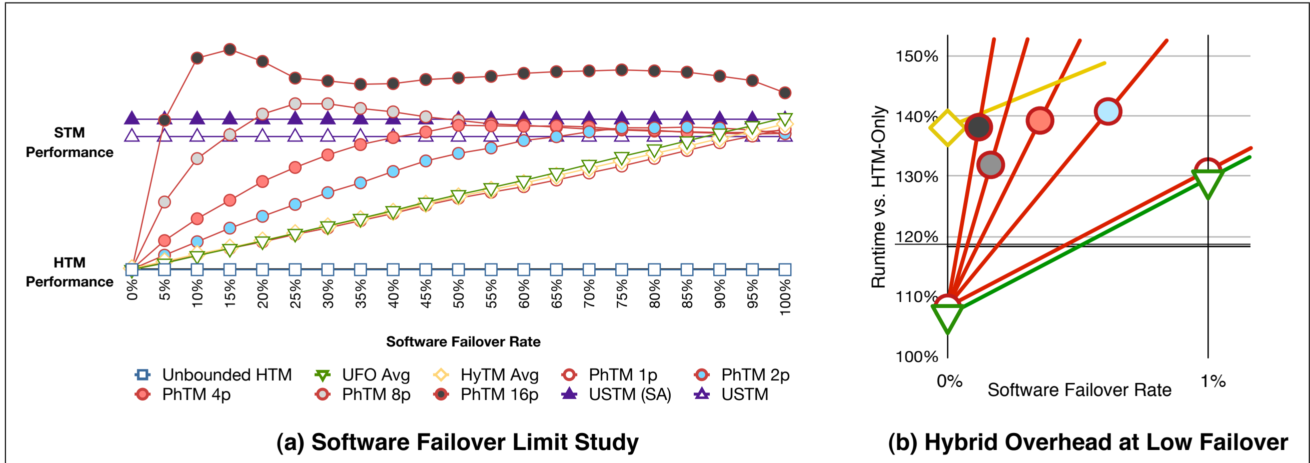


Figure 7. Software Failover Limit, in (a), compares the hybrid proposals listed in Section 5. PhTM, whose performance varies with the number of processors, has a curve for each of 1p, 2p, 4p, 8p, and 16p; for HyTM and the UFO hybrid we plot only the 8p runs as their behavior doesn't vary with processor count. Parallel runtime (y-axis) is normalized to HTM-only execution as software failover rate (x-axis) is varied from 0-100%. STM data are provided for reference. (b) zooms into (a) at low software failover rates.

that requires that a TM has robust contention management so that forward progress and scalability are ensured, as we discuss in Section 5.4. As relatively few transactions fail over to software, PhTM and the UFO hybrid are able to give performance comparable to the unbounded HTM.

5.3. Software Failover

We find that the performance of the hybrid TMs is a complex interaction involving contention and the fraction of transactions that must execute in software. In real programs, these variables are related, as having a transaction fail over to software typically increases the amount of contention. In an attempt to isolate the impact of *software failover rate*, we constructed a microbenchmark where the transactions result in no conflicts, but randomly fail over to software at a prescribed rate. We show the results of this microbenchmark in Figure 7 as a function of the failover rate, compared to pure HTM and pure STM approaches.

Clearly, for every hybrid, an increasing software failover rate results in decreasing performance – or, more precisely, as more transactions are forced to software, performance becomes more like a pure STM. However, the *rate at which performance decreases* differs between the hybrid proposals. UFO Hybrid and HyTM vary almost linearly between pure HTM and pure STM performance, executing only as many transactions in software as were randomly failed over, but PhTM must execute not only those transactions in software, *but also any other concurrent transactions* in software, even if they could have completed in hardware. This behavior worsens with increased processor count, with increased likelihood that at least one transaction needs to execute in the STM, as was seen in `vacation` in Figure 5.

While all of the hybrids approach pure HTM performance at low failover rates, Figure 7b shows that there are performance differences. At 0% failover, the UFO hybrid performs equivalently with the pure HTM; the observed 6% overhead is due to additional code in all of the hybrid TMs that force the software failovers. PhTM incurs an additional 2% overhead from checking the counter of STM transactions. HyTM's overhead is even higher, due to the `otable`

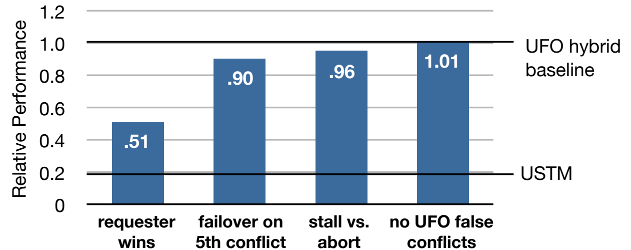


Figure 8. Sensitivity Analysis Average results shown for 16 processor runs.

lookups it must perform. In this way, the UFO hybrid exhibits our *pay-per-use* principle, where the overhead of supporting STM transactions is only paid by those transactions that execute in the STM. The UFO hybrid, however, introduces overhead (not present in HyTM and PhTM) into its software transactions to set and clear the UFO bits. As a result the UFO hybrid's curve has a greater slope than HyTM, making it underperform HyTM for software failover rates exceeding 45%, but it is unclear whether the performance of workloads with such high failover rates is important.

5.4. Sensitivity Analysis

In the development of the policies described in Section 4.4, ran a number of experiments exploring alternative policies, which we attempt to summarize here. Our most important result is how performance tanked whenever we used a low quality hardware contention management policy. For example, the performance of a simple "requestor wins" policy (1st bar in Figure 8) is result of STM-like performance in high contention regions. Also, such policies required failing over to the STM after a number (*e.g.*, 5) of contention-induced aborts to avoid live-lock.

Second, presuming we had a strong hardware contention management policy, we found that performance was better if conflicts never caused a fail-over to software compared to failing-over on the nth abort (2nd bar). We found this effect could be partially mitigated by preventing hardware transactions from aborting unless ab-

solutely necessary (3rd bar) — for example, we tried having BTM transactions stall instead of abort on UFO faults resulting from conflicts with STM transactions — but this is not necessary when contention never causes a fail-over. Finally, we include results for a limit study where UFO bit sets only abort BTM transactions when they represent a true conflict that shows that there is little lost performance due to false conflicts (4th bar).

6. Concluding Remarks

We have described an implementation of transactional memory which we believe is compelling. Our proposed system, the UFO hybrid TM, is comprised of two hardware primitives of modest complexity which have broad applicability beyond TM, provides a clean TM programming model with strong atomicity, and achieves performance rivaling an unbounded HTM across a broad set of workloads, without the challenges of having to guarantee completion of all transactions in hardware. In addition, since the semantics of the TM are not architected in hardware, this approach is viable even in the presence of the current TM bootstrapping problem: without significant transactional application development, it is difficult to know what features a useful TM programming model requires, but few – if any – programmers will be willing to do significant development until TM performs acceptably (that is, better than STMs). The hybrid approach permits evolution of the TM semantics via the STM's extensibility while preserving near-hardware performance.

As part of the development of the UFO hybrid, we have been extending its programming model to support system calls, I/O, and transactional waiting. Idempotent system calls (*e.g.*, `sbrk`, `gettimeofday`) are already trivially supported by failing over to the STM. We use this support to handle `malloc` in transactions; a feat achieved much less gracefully in our unbounded HTM by introducing complexity into its abort handler. By further adding support for deferring, "going non-speculative", and compensation code the vast majority of side-effecting operations exhibited in real code can be supported [3]. Such extensions are straightforward because they only require modification of the STM.

Adding support for the transactional waiting primitive `retry` [15] is more challenging because of its required interaction with HTM transactions. A transaction which determines, based on transactionally-read data, that conditions prevent its forward progress may issue the `retry` command. This action undoes the transaction's speculative writes, converts all its held `otable` entries to transactionally-read, and changes its state to *retrying*. The transaction may then deschedule itself. When a later transaction, while committing, updates a value that the *retrying* transaction had read, it awakens the *retrying* transaction, which releases its remaining `otable` entries and restarts as if after an abort. Such support is included in the STM in our nascent implementation of `retry` in the UFO hybrid.

The HTM component of `retry` can be implemented using existing features of BTM and UFO. First, when the compiler generates the HTM version of the code, it translates `retry` into an explicit abort, causing transactions reaching that point to failover to software. Second, an HTM can detect that it is conflicting with a *retrying* transaction by inspecting the `otable` in the user-mode UFO fault handler (executed while in BTM), and the ID of the other transaction can be recorded so that it can be awakened after the BTM commit. The transaction can then clear the UFO bit, relying on the fact that this update will not be visible until its commit and will be discarded if it aborts. Seeing how naturally our primitives enabled integrating `retry` into the UFO hybrid makes us cautiously optimistic that the UFO hybrid approach provides the necessary extensibility to support a broad range of TM features.

7. Acknowledgments

This research was supported in part by NSF CAREER award CCR-03047260, NSF CNS-0615372, and a gift from the Intel corporation. We would like to thank Ravi Rajwar and Konrad Lai for productive discussions relating to this work.

References

[1] C. S. Ananian *et al.*, "Unbounded Transactional Memory," in *HPCA-XI*, Feb. 2005.
[2] A. W. Appel, J. R. Ellis, and K. Li, "Real-time concurrent collection on stock multiprocessors," in *ACM SIGPLAN*, June 1988.
[3] L. Baugh and C. Zilles, "An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory," in *TRANSACT-II*, Aug. 2007.
[4] C. Blundell *et al.*, "Subtleties of Transactional Memory Atomicity Semantics," *Computer Architecture Letters*, vol. 5, Nov. 2006.
[5] C. Blundell *et al.*, "Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, 2007.
[6] J. Bobba *et al.*, "Performance Pathologies in Hardware Transactional Memory," in *ISCA-34*, Jun 2007.
[7] L. Ceze *et al.*, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA-34*, June 2007.
[8] W. Chuang *et al.*, "Unbounded Page-Based Transactional Memory," in *ASPLOS-XII*, Oct. 2006.
[9] P. Damron *et al.*, "Hybrid Transactional Memory," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, 2006.
[10] D. Dice, O. Shalev, , and N. Shavit, "Transactional Locking II," in *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, 2006.
[11] M. Franklin and G. S. Sohi, "The expandable split window paradigm for exploiting fine-grain parallelism," in *ISCA-19*, May 1992.
[12] L. Gwennap, "Alpha 21364 to Ease Memory Bottleneck," in *Microprocessor Report*, Oct. 1998.
[13] L. Hammond *et al.*, "Transactional Memory Coherence and Consistency," in *ISCA-31*, June 2004.
[14] T. Harris, "What does 'atomic' mean?." Presentation.
[15] T. Harris *et al.*, "Composable Memory Transactions," in *PPOPP*, 2005.
[16] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *ISCA-20*, May 1993.
[17] W. N. S. III and M. L. Scott, "Advanced Contention Management for Dynamic Software Transactional Memory," in *PODC-XXIV*, 2005.
[18] J. R. Larus and R. Rajwar, *Transactional Memory*. Dec. 2006.
[19] Y. Lev, M. Moir, and D. Nussbaum, "PhTM: Phased Transactional Memory," in *TRANSACT-II*, Aug. 2007.
[20] P. S. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, Feb. 2002.
[21] M. M. Martin *et al.*, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News (CAN)*, Sept. 2005.
[22] C. J. Mauer *et al.*, "Full system timing-first simulation," in *SIGMETRICS-02*, June 2002.
[23] A. McDonald *et al.*, "Architectural Semantics for Practical Transactional Memory," in *ISCA-33*, June 2006.
[24] C. C. Minh *et al.*, "An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees," in *ISCA-34*, June 2007.
[25] K. E. Moore *et al.*, "LogTM: Log-based Transactional Memory," in *HPCA-XII*, Feb. 2006.

[26] M. J. Moravan *et al.*, "Supporting nested transactional memory in LogTM," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, 2006.

[27] N. Neelakantam *et al.*, "Hardware Atomicity for Reliable Software Speculation," in *ISCA-34*, June 2007.

[28] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *MICRO-34*, Dec. 2001.

[29] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *ISCA-32*, June 2005.

[30] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," in *SPAA-IX*, June 1997.

[31] C. J. Rossbach *et al.*, "TxLinux: Using and Managing Transactional Memory in an Operating System," in *SOSP-XXI*, Oct. 2007.

[32] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," in *ASPLOS-VI*, Oct. 1994.

[33] M. L. Scott *et al.*, "Transactions and Privatization in Delaunay Triangulation," in *PODC-XVI*, 2007.

[34] A. Shankar et al., "Runtime Specialization with Optimistic Heap Analysis," in *OOPSLA 2005*, 2005.

[35] T. Shpeisman *et al.*, "Enforcing Isolation and Ordering in STM," in *PLDI '07*, June 2007.

[36] T. F. Wenisch *et al.*, "Mechanisms for Store-Wait-Free Multiprocessors," in *ISCA-34*, June 2007.

[37] E. Witchel et al., "Mondrian Memory Protection," in *ASPLOS-X*, Oct 2002.

[38] W. A. Wulf, "Compilers and Computer Architecture," *IEEE Computer*, vol. 14, no. 7, 1981.

[39] M. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *ISPASS '07*, Apr. 2007.

[40] P. Zhou et al., "iWatcher: Efficient Architectural Support for Software Debugging ," in *ASPLOS-VI*, Oct. 1994.

Appendix A. UFO Implementation Details

While the implementation of UFO is not the focus of this paper, understanding its implementation is potentially important to interpreting the paper's results. To this end, we provide a detailed description of the UFO implementation assumed in this work.

Fundamentally, the UFO implementation maintains two bits of protection information for every cache block of data, extending all levels of the virtual memory hierarchy, as shown in Figure 9. Our implementation of UFO augments the hardware in three places: the cache hierarchy, the memory controller, and the execution core.

The UFO bits travel with the data throughout the cache hierarchy. In the caches, each line is extended with one UFO read and one UFO write bit; even with SMT and shared caches, only a single copy of the bits is required per cache line. The existing cache coherence protocol is used to ensure that all threads or processors observe a consistent state of the UFO bits.

UFO bits are also present at a cache-line granularity in physical memory. These bits may be explicitly architected in DRAM chips or stored in a separate memory module, but we propose repurposing some ECC bits by encoding ECC at a larger granularity, as was done to provide storage for the Alpha 21364's directory [12]. Given the increasing susceptibility to single-event upsets with decreasing feature size, it is likely that ECC will be pervasive throughout future systems. In this case, the only hardware change required to provide storage for UFO bits throughout physical memory is an amendment to the memory controller, permitting it to re-encode ECC at a coarser granularity and to store and retrieve UFO bits in the reclaimed bits.

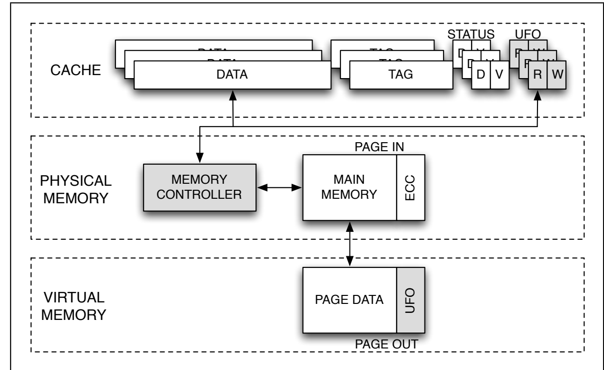


Figure 9. The Memory Hierarchy with UFO support. Shaded regions are added or altered to support UFO. In the cache, each line is augmented with Read and Write Protection bits. In the memory controller, ECC is encoded at a coarser granularity, and UFO bits are stored in ECC. In the virtual memory system, swapped-out pages are stored with (possibly compressed) UFO data.

As physical pages are swapped to and from disk, the operating system is responsible for saving and restoring the UFO bits. We modified the Linux 2.6.23.9 kernel to allocate an array with one 16-byte element per swap-file location (much like the `swap_map`), to save the UFO bits when a page is swapped to disk, to restore the UFO bits when a page is swapped from disk, and to clear the bits when a physical page is freed. Using real machine experiments, we found the overhead of these changes to be negligible in workloads where swapping normally occurs (*e.g.*, a parallel kernel build with 512MB memory). Minor overhead was observed with intensive page swapping (*e.g.*, 8% additional overhead when the same kernel build is thrashing from only having 64MB memory); the source of this overhead is additional swapping induced by accesses to the UFO-bit storage arrays. Much of this overhead is eliminated by optimizing the case in which no UFO bits have been set (and thus do not need to be saved or restored) by maintaining an additional array with a single bit per page indicating that the all the UFO bits in a page are clear.

When an instruction accesses the cache, the UFO bit for the type of the access is consulted (as part of the tag check) and recorded in the instruction's ROB entry, resulting in no additional overhead. Immediately prior to the instruction's retirement, this bit is checked; if it is set, then a UFO fault is raised. To support weak consistency models, stores can be speculatively retired into a store buffer before the cache block is available, using one of the many previously proposed techniques [7, 30, 36] to recover precise state if a UFO fault is required.

The `[set,add,read]_ufo_bits` instructions are treated by the pipeline like memory instructions. Both `set_ufo_bits` and `add_ufo_bits` behave similarly to stores: they require exclusive coherence permission to the cache line and must have write access and update the page's dirty bit in the TLB (to ensure that UFO bits are swapped properly and the semantics of operations like copy-on-write are maintained). The `read_ufo_bits` instruction is performed speculatively (like normal loads) and must be invalidated based on coherence events as per the memory consistency policy on the platform.