Two and a Half hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

M.Sc. in Advanced Computer Science

Future Multi-Core Computing

Date: Thursday 21st January 2010
Time: 09.45 – 12.15

Please answer Question ONE and ONE section from Question TWO

Two academic papers are attached for use with Question 2(a) and Question 2(b) otherwise, this is a CLOSED book examination

The use of electronic calculators is NOT permitted
1. **Compulsory**

a) Discuss the reasons behind the move to multi-core processors which has been adopted by most major processor manufacturers.  

b) Describe what is meant by a ‘cache replacement algorithm’. Give examples of three different possible replacement algorithms and discuss the advantages and disadvantages of each.  

c) Explain the need for cache coherence in a multi-core processor and outline a protocol which can be used to maintain coherence in a bus based multi-core system. You do not need to provide a detailed state transition diagram or the exact detail of all state changes.  

d) Define the terms ‘bandwidth’ and ‘latency’ in a communication network and discuss the differences between them using ring network implementations as an example.  

e) Using an airline seat reservation system as an example, discuss the need for synchronisation between concurrent accesses to shared data. How do the options for implementing synchronisation affect the complexity and performance of such a system.
2. **Answer either Section (a) or Section (b) from this Question**

Provide an analysis of one of the attached two papers in the following form:

i) What is the problem being addressed? (5 marks)
ii) What is the proposed solution? (5 marks)
iii) What are the assumptions? (5 marks)
iv) How is the work evaluated? (5 marks)
v) What are the limitations? (5 marks)
vi) Overall assessment of paper? (5 marks)

a) The attached paper “Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory” describes a hardware system which takes a novel approach to the implementation of Transactional Memory. Provide an analysis of the paper as outlined above.

   (30 marks)

b) The attached paper “Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization” describes a novel software mechanism to support synchronization for parallel programming. Provide an analysis of the paper as outlined above.

   (30 marks)
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 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.

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], Mon-driaan 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. 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.

1Software 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 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. For example, both multi-word compare-and-swap and TM systems have exposed non-intuitive semantics, two examples of which are shown in Figure 2.

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 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

---

2Privatization 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).
side-effecting operations [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 vectoried 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.

<table>
<thead>
<tr>
<th>btm_begin imm32</th>
<th>Begin a BTM transaction with abort address given in immediate &lt;imm32&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>btm_end</td>
<td>End a BTM transaction</td>
</tr>
<tr>
<td>btm_abort</td>
<td>Abort a BTM transaction</td>
</tr>
<tr>
<td>btm_mov reg, txr</td>
<td>Copy transactional status register &lt;txr&gt; to register &lt;reg&gt;</td>
</tr>
</tbody>
</table>

Table 1. The BTM ISA extension
set the UFO bits for the memory line containing <addr> to <bits>

add_ufo_bits addr, bits OR <bits> into 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ustm_begin()</td>
<td>Begin a USTM transaction (checkpoint regs, clear log, get seq number, set transaction state)</td>
</tr>
<tr>
<td>ustm_end()</td>
<td>End a USTM transaction (release ownership, discard checkpoint)</td>
</tr>
<tr>
<td>ustm_abort()</td>
<td>Abort a USTM transaction (undo writes, release ownership, restore chkpt)</td>
</tr>
<tr>
<td>ustm_read_barrier</td>
<td>Acquire read permission for &lt;addr&gt;</td>
</tr>
<tr>
<td>ustm_write_barrier</td>
<td>Acquire write permission for &lt;addr&gt;</td>
</tr>
</tbody>
</table>

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 achieve 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 enabled: 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 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

The use of the term “barrier” derives from the garbage collection literature and not the instructions required by weak memory consistency models.
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-enable them at commit.

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, 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...
1: procedure USTM_WRITE_BARRIER(trans, addr)
2:   index ← GET_INDEX(addr); tag ← GET_TAG(addr)
3:   entry_state ← MAKE_ENTRY_STATE(trans, addr, WRITE)
4:   o ← otable[index];
5:   if o = 0 then
6:     COMPARE&SWAP(htable[index], 0, entry_state) return
7:   else if LOCKED(o) then BACKOFF(); goto 4
8:   else if TAG(o) = tag then
9:     LOCK_ROW(index, o); HANDLE_CHAIN(index, tag, trans, entry_state); UNLOCK_ROW(index)
10:  else if OWNERS(o) = trans then
11:     if STATE(o) = WRITE then
12:       COMPARE&SWAP(htable[index], o, entry_state)
13:   else RESOLVE_CONFLICT(index, trans, o); goto 4
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.

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();
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.\textsuperscript{5} 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.

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...
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 lock’s current owner, the lock 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 outstanding 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, 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

---

7If 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.

8While 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 the same otable rows previously read by HTM transactions. 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 excluding 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 STMs. When this second counter reaches zero, PhTM starts the shift back to an HTM phase by stalling transactions rather than starting them in the STMs. 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.

<table>
<thead>
<tr>
<th>Table 4. Simulation parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor frequency</td>
</tr>
<tr>
<td>Fetch/Decode width</td>
</tr>
<tr>
<td>Rename/Issue/Retire width</td>
</tr>
<tr>
<td>Instruction window size</td>
</tr>
<tr>
<td>Branch predictor</td>
</tr>
<tr>
<td>Indirect target predictor</td>
</tr>
<tr>
<td>L1 Data Cache</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
</tr>
<tr>
<td>L1 Cache Line size</td>
</tr>
<tr>
<td>L2 Cache Line size</td>
</tr>
<tr>
<td>Physical memory</td>
</tr>
<tr>
<td>Coherence protocol</td>
</tr>
<tr>
<td>Operating system</td>
</tr>
<tr>
<td>Linux kernel</td>
</tr>
<tr>
<td>USTM otable size</td>
</tr>
</tbody>
</table>

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.

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. 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-

---

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

10STAMP benchmarks feature no explicit aborts, so any aborts seen in Figure 6 are due to the operation of PhTM and HyTM.
actions that overflow the cache. 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 table 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 table entries competing with the transaction’s data for space in the transactional cache. These table 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.

genoexhibits 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 transition...
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 vacuum 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 "stable 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-
olutely 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 near TM 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


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.

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]_ufu_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.
Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization

J. Shirako  D. M. Peixotto  V. Sarkar  W. N. Scherer III
Department of Computer Science, Rice University, 6100 Main Street, Houston TX 77025
{shirako, dmp, vsarkar, scherer}@rice.edu

ABSTRACT
Coordination and synchronization of parallel tasks is a major source of complexity in parallel programming. These constructs take many forms in practice including mutual exclusion in accesses to shared resources, termination detection of child tasks, collective barrier synchronization, and point-to-point synchronization. In this paper, we introduce phasers, a new coordination construct that unifies collective and point-to-point synchronizations. We establish two safety properties for phasers: deadlock-freedom and phase-ordering. Performance results obtained from a portable implementation of phasers on three different SMP platforms demonstrate that phasers can deliver superior performance to existing barrier implementations, in addition to the productivity benefits that result from their generality and safety properties.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming

General Terms
Languages

1. INTRODUCTION
The computer industry is entering a new era of mainstream parallel processing in which the need for improved productivity in parallel programming has taken on a new urgency. One of the major obstacles to improved productivity is the complexity of coordination and synchronization of parallel tasks that is inherent in current parallel programming models. Coordination and synchronization constructs take many forms in practice such as mutual exclusion in accesses to shared resources using locks, termination detection of child threads using join operations, collective synchronization using barriers, and point-to-point synchronization using semaphores. Recent efforts on productivity improvements in parallel programming include transactional memory systems for mutual exclusion [7], work stealing schedulers to support dynamic nested parallel execution model with lightweight task creation and termination detection as in Cilk [11], and deadlock-free barrier synchronizations as in X10’s clocks [2]. These approaches have motivated new research on delivering efficient parallel performance, while retaining safety guarantees that are important for productivity.

In this paper, we focus on reducing the complexity of collective and point-to-point synchronization, both of which are used extensively in parallel algorithms. We introduce phasers, a new coordination construct that unifies collective and point-to-point synchronization. We establish two safety properties for phasers: deadlock-freedom and phase-ordering. Performance results obtained from a portable implementation of phasers on three different SMP platforms demonstrate that they can deliver superior performance to existing barrier implementations, in addition to the productivity benefits that result from their generality and safety properties.

The rest of the paper is organized as follows. The phaser construct is introduced in Section 2, and its phase-ordering and deadlock-avoidance properties are established in Section 3. Section 4 presents experimental results to compare the performance of phasers with other synchronization constructs, using an SMP implementation of phasers developed in the Habanero multicore software research project at Rice University [6]. Section 5 discusses related work, and Section 6 contains our conclusions.

2. OVERVIEW OF PHASERS
In this section, we introduce phasers, a new coordination construct that unifies collective and point-to-point synchronizations and embodies the following contributions:

1. Integration of producer-consumer with barrier synchronization. An activity has the option of registering with a phaser in signal-only mode or wait-only mode for producer-consumer synchronization, in addition to signal-wait mode for barrier synchronization.

2. Support for single statements. A next statement for phasers can optionally include a single statement which is guaranteed to be executed exactly once during a phase transition [16].

3. Scalable implementation on multicore SMPs. By design, phasers are amenable to scalable implementation on multicore SMPs, as demonstrated in Section 4.
int iters = 0; delta = epsilon+1;
while ( delta > epsilon ) {
    finish {
        for ( jj = 1 ; jj <= n ; jj++ ) {
            final int j = jj;
            async {
            } // async
        } // for
    } // finish
    delta = diff.sum(); iters++;
    temp = newA; newA = oldA; oldA = temp;
} System.out.println("Iterations: "+ iters);

Figure 1: One-Dimensional Iterative Averaging in X10 using Async and Finish

These properties, along with the generality of dynamic parallelism and the phaser-ordering and deadlock-freedom safety properties, distinguish phasers from synchronization constructs in past work including barriers [5, 10], counting semaphores [12] and X10’s clocks [2].

Section 2.1 briefly summarizes the async, finish and clock constructs from X10 [2], which provide the context for our work on collective and point-to-point synchronization. Section 2.2 introduces the phaser construct and the operations that can be performed on phasers. Though the description of phasers in this paper may appear to be specific to X10, they are a general unification of point-to-point and collective synchronizations that can be incorporated in any parallel programming model with a shared address space such as OpenMP, Intel’s Thread Building Blocks, Microsoft’s Task Parallel Library, Java Concurrency Utilities, Unified Parallel C, Co-Array Fortran and Titanium.

2.1 Async, Finish, Clocks

This section provides a brief summary of the async, finish, and clock constructs introduced in v0.41 of the X10 programming language [2].

2.1.1 async (stmt)

Async is the X10 construct for creating or forking a new asynchronous activity. The statement, async (stmt), causes the parent activity to create a new child activity to execute (stmt). Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to its next statement.

2.1.2 finish (stmt)

The X10 statement, finish (stmt), causes the parent activity to execute (stmt) and then wait till all sub-activities created within (stmt) have terminated (including transitively spawned activities). There is an implicit finish statement surrounding the main program in an X10 application.

Each dynamic instance of a finish statement can be viewed as being bracketed between matching instances of start-finish and end-finish instructions. Operationally, each instruction executed in an X10 activity has a unique Immediately Enclosing Finish (IEF) dynamic statement instance. In the X10 computation DAG introduced in [1], a dependence edge

is introduced from the last instruction of an activity to the end-finish node corresponding to the activity’s IEF.

We use the pedagogical One-Dimensional Iterative Averaging program from [3] as a running example in this paper. The goal of this program is to perform iterative averaging on a one-dimensional array, A[0:n+1], initialized with A[0:n] = 0 and A[n+1] = n+1. A[0] = 0 and A[n+1] = n+1 are fixed boundary conditions, and a 2-point stencil is used to iteratively replace A[j] by the average of A[j-1] and A[j+1] for 1 ≤ j ≤ n. The algorithm terminates when the sum of element changes from one iteration to the next is less than a given threshold. Figure 1 contains a simple X10 version of this example using finish and async constructs. Note that the async activities created in iterations of the j loop can all run in parallel, and that the finish statement ensures that all the async’s have terminated before execution proceeds to the diff.sum() computation.

2.1.3 Clocks

While the code in Figure 1 is correct, the overhead of repeatedly spawning and terminating activities in each iteration of the while loop and the potential accompanying loss of locality can be a major performance bottleneck. Instead, it would be preferable to perform a barrier-like coordination within each iteration of the while loop, without terminating the activities involved. This serves as one of the motivations for X10’s clocks.

An X10 activity, Ai, can allocate a clock, Ci, and then create a child activity Aj registered with Ci by using the clocked async construct, “async clocked(C) <stmt>”. Ai starts executing in the same clock phase as its parent, Aj. Note that, unlike barriers in standard SPMD models, X10’s clock construct permits the set of activities registered with a clock to vary dynamically.

When multiple activities (such as Ai and Aj) registered with the same clock need to perform a collective barrier synchronization, they do so by executing a next; statement which forces the activity to suspend until all clocks with
which it is registered can advance. A clock can advance only when all activities that are registered with it execute a `next` statement. X10 also permits an activity to `drop` a clock, which implicitly unregisters it with C. A terminating activity implicitly drops all clocks with which it is registered.

Figure 2 shows an X10 version of the Iterative Averaging example that uses clocks. A key difference from the previous version is that the while loop is now pushed inside the body of each async, so activities are not created and terminated in every iteration of the while loop.

## 2.2 Phasers

A `phaser` is a synchronization object that supports the operations described below. At any point in time, an activity can be registered in one of four modes with respect to a phaser: `signal-wait-next`, `signal-wait`, `signal-only`, or `wait-only`. The mode defines the capabilities of the activity with respect to the phaser. There is a natural lattice ordering of the capabilities as shown in Figure 3.

The phaser operations that can be performed by an activity, $A_i$, are defined as follows:

1. `new`: When $A_i$ performs a `new phaser(MODE)` operation, it results in the creation of a new phaser, $ph$, such that $A_i$ is registered with $ph$ according to $MODE$. Phaser creation also initializes the phaser to its first phase (phase 0).

2. `phased async`: `async phased (mode1(ph), \ldots) A_j`  
   When activity $A_i$ creates an async child activity $A_j$, it has the option of registering $A_j$ with any subset of phaser capabilities possessed by $A_i$. The following constraints are imposed on the transmission of phasers:
   
   (a) **Capability rule**: $A_i$ can only transmit a capability on phaser $ph$ to $A_j$ if $A_i$ itself has that capability or higher (Figure 3). The capability rule is imposed to avoid race conditions on phaser operations.

   (b) **IEF Scope rule**: $A_i$ can only transmit a capability on phaser $ph$ to $A_j$ if the creation (`new`) instruction for $ph$ has the same Immediately Enclosing Finish (c.f. Section 2.1) as the async statement for $A_j$; i.e., both $ph$ and $A_j$ were created in the scope of the same dynamic finish statement. The IEF rule is imposed to avoid a potential deadlock between finish operations and wait/next operations on phasers.

An attempt to transmit a phaser that does not obey the above two rules will result in a PhaserException being thrown at runtime. We allow the following default syntax to indicate that $A_i$ is transmitting all its capabilities on all phasers that it is registered with to $A_j$: “$A_i : \text{async phased } A_j$”.

3. `drop`: There are two ways in which $A_i$ can drop a registration that it holds with a phaser:
   
   (a) **Activity termination**: When $A_i$ terminates execution, it performs an implicit `next` operation, and then completely de-registers from each phaser that it was registered with.

   (b) **End-finish**: When $A_i$ executes an end-finish instruction for finish statement $F$, it completely de-registers from each phaser $ph$ created by $A_i$ in the scope of $F$; i.e., for which $F$ is the IEF for the creation statement of $ph$. Thus, the lifetimes of phasers naturally follow the lexical scoping of finish constructs.

4. `next`: The `next` operation has the effect of advancing each phaser on which $A_i$ is registered to its next phase, thereby synchronizing all activities registered on the same phaser. As indicated in Table 1, the semantics of `next` depends on the registration mode that $A_i$ has with a specific phaser, $ph$.

5. `next with single statement`: The `next (stmt)` operation has the semantics of a `next` statement as defined above, with the extension of executing `stmt` as a single statement. This operation is only permitted if $A_i$ is registered in `signal-wait-next` mode on the phaser (see Table 1). Further, we require all other activities registered with the phaser in `signal-wait-next` mode and executing a `next` with single statement must execute the same static `next (stmt)` statements. These constraints are imposed to ensure the integrity of the single statement [16].

6. **Phaser-specific signal**: We permit $A_i$ to perform a phaser-specific signal operation, `ph.signal()`, for any phaser $ph$ on which it is registered with a signal capability. The operation performed depends on $A_i$’s registration mode for $ph$. If the registration mode is `signal-wait-next` or `signal-wait`, then `ph.signal()` effectively converts $ph$ into a fuzzy barrier [5] for $A_i$ by allowing local work to be performed between the `ph.signal()` and `next` operation.

7. `signal`: A `signal` operation performed by $A_i$ is a shorthand for a `ph.signal()` operation performed on each phaser $ph$ with which $A_i$ is registered with a signal capability.

Figure 4 shows the X10 iterative averaging example from Figure 2 rewritten to use phasers. There are two major differences between the phaser version and X10-clock version. First, the two `next` statements and their intervening conditional statement that computes `diff.sum()` in the X10 version have been combined into one `next`-with-single statement in the phaser version. Not only does this reduce overhead, but it also lets the runtime take the responsibility for determining which activity should execute the `single` statement i.e., which activity should be the `master` (using the terminology from Section 4). The second major difference is that the programmer is not required to insert an additional `async phased` at the outer level, as in the X10 version. X10 prohibits a
Table 1: Semantics of phaser operations as a function of registration modes

<table>
<thead>
<tr>
<th>Operation</th>
<th>Registration Mode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>signal-wait-next or signal-wait</td>
<td>signal + wait (or just wait if previous operation was signal)</td>
</tr>
<tr>
<td>next (stmt) (next w/ single stmt)</td>
<td>signal-wait-next</td>
<td>signal + wait + single execution of stmt (or just wait + single execution of stmt if previous op was signal)</td>
</tr>
<tr>
<td>ph.signal()</td>
<td>signal-wait-next or signal-wait</td>
<td>signal on ph (or error if previous op by activity on ph was signal)</td>
</tr>
<tr>
<td>signal</td>
<td>*</td>
<td>Perform ph.signal() on each phaser that activity has a signal capability on</td>
</tr>
</tbody>
</table>

```
finish {
    delta.f = epsilon+1; iters.i = 0;
    phaser ph = new phaser();
    for ( jj = 1 ; jj <= n ; jj++ ) {
        final int j = jj;
        async phased { // will be registered with ph
            while ( delta.f > epsilon ) {
                next { // Single statement
                    delta.f = diff.sum(); iters.i++;
                    } // while
                } // async
        } // for
    } // finish
    System.out.println("Iterations: " + iters.i);
}
```

Figure 4: One-Dimensional Iterative Averaging using Phasers for Collective Synchronization

clocked async from being created immediately in the scope of a finish to avoid a potential deadlock with the end-finish operation. Phasers instead avoid deadlock by enforcing the IEF scoping rule on phasers, which ensures that all activities have de-registered from phaser ph after the end-finish instruction.

Figure 5 shows an alternate version of the iterative averaging example as an illustration of the use of phasers for point-to-point synchronization instead of barrier synchronization. For simplicity, this version uses a for loop with a fixed number of averaging iterations in each async activity, so it will not produce the same output as the version in Figure 4 which used a while loop to control termination. In this example, each async is registered with three phasers — one in signal-only mode and two in wait-only mode. Note that phasers gracefully handle boundary conditions that often arise in point-to-point synchronization. For example, it is not necessary to provide a signal operation to match the wait on ph[j-1] in the async for the j = 1 iteration or on ph[j+1] in the async for the j = n iteration.

```
finish {
    phaser[] ph= new phaser[n+2]; // array of phasers
    for ( jj = 1 ; jj <= n ; jj++ ) {
        final int j = jj;
        async phased(signalOnly(ph[j]), waitOnly(ph[j-1]), waitOnly(ph[j+1]) ) {
            for ( int iter=0 ; iter<NUM_ITERS ; iter++ ) {
                signal; // Signals ph[j]
                // Activity can do local work here
                next; // Await signals from iters j-1 & j+1
                temp = newA; newA = oldA; oldA = temp;
            } // for
        } // async
    } // for
} // finish
```

Figure 5: Alternative version of One-Dimensional Iterative Averaging using Phasers for Point-to-Point Synchronization

3. PHASE-ORDERING AND DEADLOCK-AVOIDANCE IN PHASERS

In this section we precisely define the guarantees phasers provide for phase ordering and deadlock avoidance by building on the notion of an X10 computation DAG introduced in [1] for finish and async constructs. Due to space limitations, these safety properties are established by informal proof sketches rather than formal proofs.

3.1 Phase ordering

We model execution constraints using a Dynamic Computation DAG (DCD). The DCD has a node for each instance of an instruction executed during a computation. The edges in the DCD encode constraints in the execution order. For the computation to be correct, an instruction may not execute until all of its predecessors have executed. In addition to the normal instruction nodes, two special accumulator nodes are added to model phasers. The begin-accum and end-accum nodes are used to enforce constraints between phases so that instructions following a wait will not execute until all signals for that phase have executed.
The different types of edges in the DCD are discussed below.

1. **continue**: A continue edge represents sequential control flow. There is an edge from node \(A\) to \(B\) in the DCD if the computation executes the instruction sequence \(A;B\) in the same activity.

2. **async**: An async edge is added from node \(A\) to \(B\) if \(A\) is an async instruction and \(B\) is the first instruction in the new activity.

3. **finish**: A finish edge is added from node \(A\) to \(B\) if \(A\) is the last instruction of an activity and \(B\) is the end-finish instruction for the IEF of that activity.

4. **next**: A next instruction decomposes into a signal followed by a wait instruction, with edges added for both instructions as described below.

5. **signal**: A signal edge is added from node \(A\) to \(B\) if \(A\) is a signal instruction and \(B\) is the begin-accum node for the phase of the phaser to which the signal is sent.

6. **wait**: A wait edge is added from node \(A\) to \(B\) if \(A\) is the end-accum node for the phase of the phaser and \(B\) is a wait instruction performed on that phaser.

7. **next with single**: An edge is added from the signal node to the begin-accum node as normal. The nodes in the single sequence are connected with continue edges starting from the begin-accum node and ending at the end-accum node. Finally, edges are added from the end-accum node to all wait nodes in the same manner as for a standard wait edge.

We annotate each node in the DCD with the current signal and wait phase for each phaser the activity is registered with. For each phaser \(ph\) with which the activity is registered, define the functions

- \(S(ph, i): P \times \text{Insts} \rightarrow \mathbb{N} \cup \{\infty\}\) is the number of signal operations that have been performed on phaser \(ph\) by the current activity (or ancestors) before the activity executes instruction \(i\).
- \(W(ph, i): P \times \text{Insts} \rightarrow \mathbb{N}\) is the number of wait operations that have been performed on phaser \(ph\) by the current activity (or ancestors) before the activity executes instruction \(i\).

Here, \(\text{Insts}\) is the set of all instruction instances, \(P\) is the set of all phasers, and \(\mathbb{N}\) is the set of non-negative integers. The ancestors of an activity are defined recursively to be the parent of an activity (i.e. the activity that executed the async starting the current activity) and any ancestors of the parent. The main activity has no ancestors.

In addition to the \(S\) and \(W\) values associated with each node in the DCD, there are \(S\) and \(W\) values associated with each phaser that are independent of any activity. The values for the \(S\) and \(W\) functions are defined in Table 2.

Figure 6 shows an example of a DCD and the \(S\) and \(W\) values for two activities registered as signal-wait-next on the phaser \(ph\). The example illustrates several features including a fuzzy barrier and next with single.

![Figure 6: Example of a Dynamic Computation DAG and its associated \(S\) (SP) and \(W\) (WP) values for the phaser \(ph\). The figure shows two signal nodes \((S)\), two wait nodes \((W)\), a begin-accum \((A_b)\), and end-accum \((A_e)\) node.](image)

We can now precisely define the phase-ordering property, which states how the signal phase and wait phase relate to the execution order of instructions.

**Phase-ordering property**: Given two instruction instances \(i_1\) and \(i_2\) in the DCD, if \(\exists\) a phaser \(ph. S(ph, i_1) < W(ph, i_2)\) then \(i_1 \prec i_2\) where \(i_1 \prec i_2\) is shorthand for “\(i_1\) precedes \(i_2\)”. We now establish that the definitions of \(S\) and \(W\) given in Table 2 satisfy the phase-ordering property. Consider four cases:

- The activity executing \(i_1\) is registered on phaser \(ph\) as wait-only. Then \(S(ph, i_1) = \infty\) and the \(S(ph, i_1) < W(ph, i_2)\) precondition will be false.
- The activity executing \(i_1\) is registered on phaser \(ph\) as signal-only. If \(S(ph, i_1) < W(ph, i_2)\) then \(i_2\) is being executed by an activity that has waited on the phaser more times than the activity executing \(i_1\) has signaled. It must be that \(i_1\) is executed before \(i_2\), because \(i_2\) could not be executed without at least one more signal from the activity executing \(i_1\).
- The activity executing \(i_1\) is registered on phaser \(ph\) as signal-wait. If \(S(ph, i_1) < W(ph, i_2)\) then the same argument follows as in the signal case above. If \(i_1\) occurs in a split phase then we cannot establish any order on the activities executing in the next wait phase \(W(ph, i_1) + 1\) because we have signaled that they may continue on to phase \(W(ph, i_1) + 1\). The inequality will only hold for instructions two wait phases ahead, \(W(ph, i_1) + 2\).
- \(i_2\) occurs in a next with single section. When \(i_1\) occurs in the next with single section, it is executed with \(S\)
3.2 Deadlock avoidance

We use the same DCD to provide guarantees about the deadlock-avoidance property of phasers. An instruction instance in the DCD will not execute until all its predecessors have executed. Logical deadlock can only occur if there is a cycle in the DCD. This result does not address deadlock due to limited physical resources [1].

Deadlock results when the instruction $i_1$ has a predecessor in the DCD, $i_2$, such that there is a path from $i_1$ to $i_2$. In other words, if there is a cycle in the DCD then there is a deadlock situation because the instruction $i_1$ cannot execute until $i_2$ executes, but $i_2$ cannot execute until $i_1$ executes.

Deadlock-avoidance property: Phasers introduce no cycles into the DCD.

The semantics of phaser operations ensure that no cycle will be created in the DCD. There are two cases to consider where phasers might possibly introduce a cycle, but we will show that the semantics of phasers prevent any such cycles from being created.

First, a cycle could be created if an activity $A_1$ is waiting on a phaser that another activity $A_2$ is signaling on, and $A_2$ is waiting on a phaser for a signal from $A_1$. For this to happen, it must be the case that $A_1$ is waiting on a phaser but has not yet signaled on all of its phasers which it is registered with, preventing $A_2$ from proceeding. We can see from Table 1 that there is no way for an activity to wait on a phaser without first signaling on all of its phasers. There is no explicit wait in the programming model. The only way for an activity to perform a wait operation is by using next which explicitly signals all of an activity’s phasers before waiting on them. Thus all signals are performed before all waits and no cycle will be introduced into the DCD.

The second case to consider is the interaction between phasers and the finish construct. A cycle could be created if an activity $A_1$ is waiting at a finish node for a spawned activity $A_2$ to complete, and the activity $A_2$ is waiting for a signal from $A_1$ on a shared phaser. The IEF Scope rule described in Section 2 prevents this situation from occurring. An activity can only transmit a phaser to another activity with the same IEF, and upon reaching a finish node the activity automatically drops all phasers created in the scope of that finish. The above two rules ensure that no cycle is created in the DCD between finish nodes and phaser synchronization nodes.

A combination of static and dynamic properties is sufficient to establish the deadlock-freedom and phase-ordering safety properties. An example of a static property is that an activity is prohibited from performing a wait operation on a specific phaser. Examples of dynamic properties include the Capability and IEF Scope rules described in Section 2, and ensuring that an activity does not perform two consecutive signals on a phaser on which it is registered in signal-wait mode. The advantage of runtime checks is that an exception is thrown at the point at which an error occurs, which helps in diagnosing the source of the error (analogous to runtime checks for null pointers and index-out-of-bounds).

### 4. EXPERIMENTAL RESULTS

In this section, we present experimental results using an SMP implementation of phasers developed in the Habanero multicore software research project at Rice University [6].

Sections 4.1, 4.2 and 4.3 contain the results obtained on an 8-CPU AMD Opteron 8347 SMP with 2 quad-core processor chips, a 64-CPU IBM Power5+ SMP with 32 dual-core processor chips, and a 64-way Sun UltraSPARC T2 system with 8 eight-core chips and 8 threads per core.
The results were obtained for the following versions of the seven benchmarks shown in Table 3:

1. **Sequential Java.** This version was used as the baseline for all speedup results except BarrierBench. For the Java Grande Forum benchmarks, this version was taken from version v2.0 of the JGF benchmark release [9] with one exception — the sequential version of SOR was obtained by serializing the parallel version in threadv1.0 because the v2.0 version uses a different algorithm from the threadv1.0 version. For NAS parallel benchmarks, this version was obtained by using the "serial" option, and for BarrierJacobi, this version was obtained by removing all parallel constructs from the JUC version [4].

2. **Threaded Java.** For the Java Grande benchmarks, this version was taken from version threadv1.0 of the JGF benchmark release [9]; for NAS parallel benchmarks, this version was obtained by using the parallel option, and for BarrierJacobi; this version was obtained from the JUC version [4].

3. **X10 with clocks.** This version represents the performance that is obtained from an X10 version implemented with standard clocks. As in [13] we use a "lightweight" X10 version with regular Java arrays to avoid the large overheads incurred on X10 arrays in the current X10 implementation. However, all the other characteristics of X10 (e.g., non-null used as the default type declaration and forbidden use of non-final static fields) are preserved faithfully in all the X10 versions.

4. **X10 with tournament barriers.** This version replaces clock operations by tournament barriers [9] everywhere in the X10 program. Tournament barriers are not a standard construct in Java, but a hand implementation of a static synchronization pattern available with the JGF benchmarks — they do not provide any of the safety and dynamic features of phasers.

5. **X10 with phasers (unfixed master)** This version measures the performance of the SMP phaser implementation with the *unfixed master* option which enables the runtime to change the "master" activity for each phaser during any phase transition, thereby providing an opportunity for adapting to load imbalances.

6. **X10 with phasers (fixed master)** This version measures the performance of the SMP phaser implementation with the *fixed master* option which keeps the "master" activity fixed for each phaser i.e., the master can only be changed when the current master activity drops its registration with $p_h_j$.

For all runs, the main program was extended with a three-iteration loop within the same Java process, and the best of the three times was reported in each case. This configuration was deliberately chosen to reduce/eliminate the impact of JIT compilation time in the performance comparisons.

In an effort to reduce the number of variables that differ between the two platforms, we used the same X10 version in both cases (version 1.5 [15]) modified to optionally use tournament barriers or phasers in lieu of clocks. All X10 runs were performed with the following common X10 options:

- `-BAD_PLACE_RUNTIME_CHECK=false`
- `-NUMBER_OF_LOCAL_PLACES=1`
- `-PRELOAD_CLASSES=true -BIND_THREADS=true`

In addition, `-INIT_THREADS_PER_PLACE` was always set to the number of CPUs for which the measurement was being performed.

### 4.1 8-way Opteron SMP

All results in this subsection were obtained on a Quad-Core AMD Opteron Processor 8347 1.9 GHz SMP server with 4GB main memory running Fedora Linux release 8. The execution environment used for all Java runs is the IcedTea Runtime Environment (build 1.7.0-2b1) with IcedTea 64-Bit Server VM (build 1.7.0-2b1, mixed mode) and the -“Xmx1000M -Xmx1000M” options. In addition, Intel’s icc OpenMP compiler was used to measure the performance of the OpenMP version of BarrierBench.

Figure 7 contains two charts that use the BarrierBench microbenchmark introduced in [14] to calibrate the performance of the different versions of barriers and phasers listed at the start of this section. No results are provided for the serial or 1-CPU case because the only “useful” work done by this microbenchmark is to coordinate synchronization among multiple threads. As shown in the top chart, the barrier overhead associated with phasers is significantly lower than that of X10’s clocks (up to 30.7×), Java’s notify-wait as implemented in the SimpleBarrier class in [9] (up to 28.2×), and JUC’s cyclic barrier (up to 27.7×). In addition, the “fixed master” mode is a better choice than “unfixed master”, since there is little load imbalance among the threads in these benchmarks. Finally, even though the gap with the TournamentBarrier version (as implemented in [9]) is small, phasers with “fixed master” was still 1.4× faster than TournamentBarrier for 8 threads.

The second chart in Figure 7 compares the performance of phasers with “fixed master” with that of OpenMP barriers using the Intel icc implementation, and shows that OpenMP was faster with 2, 4, and 8 threads and phasers were faster in the remaining configurations. The difference in performance is fairly small and it is nice to see that phasers can be competitive with the OpenMP native code implementation. Our initial experiments used gcc’s OpenMP implementation and found that phasers were up to 30.4× faster. The large difference in performance underscores the importance of using a proprietary vendor implementation of OpenMP as the baseline for experimental results.

Figure 8 shows the performance comparison for the six other benchmarks listed in Table 3 in the form of speedup relative to the serial Java version. We see that the phaser runtime (with or without a fixed master) delivers the best average speedup, and is also remarkable in its consistency (unlike the TournamentBarrier which gave good speedups in many cases, but led to significant degradation for MG). It is worth noting that we obtained larger speedups (closer to 8 for 8 CPUs) when running these benchmarks with smaller data sizes than those used for Figure 8.

### 4.2 64-way Power+ SMP

The results in this subsection were obtained on a p595+ 64-way Power5+ 2.3GHz SMP server with 512GB main memory running AIX5.3 TL5. All runs were performed with SMT turned off, and with a large page size of 256GB. The execution environment used for all Java runs was IBM’s J9
Figure 7: BarrierBench microbenchmark results on 8-CPU AMD Opteron 8347 SMP

Figure 8: Speedup relative to serial Java version on 8-CPU AMD Opteron 8347 SMP
### Figure 9: BarrierBench microbenchmark results on 64-CPU Power5+ SMP

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>X10 w/ clocks</th>
<th>Java notify&amp;wait</th>
<th>Java w/ Cyclic Barrier</th>
<th>Java w/ Tournament Barrier</th>
<th>Java w/ phasers (unfixed master)</th>
<th>Java w/ phasers (fixed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>19.89</td>
<td>8.27</td>
<td>8.49</td>
<td>0.61</td>
<td>1.45</td>
<td>1.04</td>
</tr>
<tr>
<td>4</td>
<td>63.37</td>
<td>43.42</td>
<td>26.22</td>
<td>1.35</td>
<td>2.03</td>
<td>1.69</td>
</tr>
<tr>
<td>8</td>
<td>145.65</td>
<td>91.37</td>
<td>64.44</td>
<td>4.63</td>
<td>4.63</td>
<td>2.38</td>
</tr>
<tr>
<td>16</td>
<td>315.02</td>
<td>226.24</td>
<td>142.04</td>
<td>13.75</td>
<td>13.75</td>
<td>6.84</td>
</tr>
<tr>
<td>32</td>
<td>667.60</td>
<td>473.63</td>
<td>310.78</td>
<td>27.39</td>
<td>27.39</td>
<td>11.21</td>
</tr>
<tr>
<td>64</td>
<td>1506.80</td>
<td>984.00</td>
<td>676.40</td>
<td>53.78</td>
<td>53.78</td>
<td>25.00</td>
</tr>
</tbody>
</table>

- Average time per barrier [micro sec]

### Figure 10: Speedup relative to serial Java version on 64-CPU Power5+ SMP

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LUFact (size C)</th>
<th>MolDyn (size B)</th>
<th>SOR (size C)</th>
<th>CG (class A) Benchmark</th>
<th>MG (class A)</th>
<th>BarrierJacobi</th>
<th>AVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup relative to Java serial</td>
<td>10.00</td>
<td>20.00</td>
<td>30.00</td>
<td>40.00</td>
<td>50.00</td>
<td>60.00</td>
<td>70.00</td>
</tr>
</tbody>
</table>

- Speedup relative to Java serial

**Legend:**
- X10 w/ clocks
- Java threads
- X10 w/ Tournament Barrier
- X10 w/ phasers (unfixed master)
- X10 w/ phasers (fixed master)
VM (build 2.4, J2RE 1.6.0) with the following options, \texttt{-Xjit:count=0, optLevel=veryHot, ignoreIEEE -Xms1000M -Xmx1000M"}. IBM's xlc OpenMP compiler (XL C/C++ Enterprise Edition V8.0 for AIX) was used to measure the performance of the OpenMP version of BarrierBench.

As shown in the top chart of Figure 9, the barrier overhead associated with phasers is significantly lower than that of X10's clocks (up to 134.5×), Java's notify-wait (up to 87.8×), and JUC's cyclic barrier (up to 60.4×). In addition, the gap with the TournamentBarrier version (as implemented in [9]) was 2× with 8 threads and 4.8× with 64 threads.

The second chart in Figure 9 compares the performance of phasers with “fixed master” with that of OpenMP barriers using the xlc implementation. It shows that the phaser implementation is still faster than OpenMP for 2, 4, 8, 16 and 32 threads, while a cross-over was observed at 64 threads.

Moving to Figure 10, we see that the phaser runtime (with or without a fixed master) obtains significantly better average speedup (45.7×) compared to parallel Java (22.0×) or current X10 (14.3×). The TournamentBarrier version yielded a speedup of 39.1×, which is closer to but still less than the speedup obtained by phasers. However, it’s possible that some of the lessons learned from the TournamentBarrier implementation will become relevant to implementation of phasers on larger-scale SMPs in the future.

### 4.3 64-way UltraSPARC T2 SMP

All results in this subsection were obtained on a 64-way (8-core × 8 threads/core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10. The execution environment used for all Java runs is the Java(TM) 2 Runtime Environment (build 1.5.0_12-b04) with Java HotSpot(TM) Server VM (build 1.5.0_12-b04, mixed mode) and the “-Xms1000M -Xmx1000M” options. Sun’s C compiler 5.9 was used for the OpenMP version of BarrierBench.

The results for the barrier micro-benchmark are shown in Figure 11. The results for the standard benchmark suite are shown in Figure 12. Figure 11 shows the barrier overhead of phasers is significantly lower than X10’s clocks (up to 85.36×), Java’s notify-wait (up to 81.89×) and cyclic barrier (up to 75.76×). Another observable point is that TournamentBarrier shows better performance than phasers on the T2 processor, suggesting that it may have an advantage on processors with simultaneous multithreading.

Figure 12 also shows that the performance of phasers is better than others and almost equal to that of TournamentBarrier. Though the overhead of TournamentBarrier is smaller than that of phasers, the single statements in phasers reduce the number of barriers for LUFact, MolDyna, CG, MG and Barrier-Jacobi. Also, phasers provide efficient support for point-to-point synchronization in SOR.

### 5. RELATED WORK

There is an extensive literature on barrier and one-way (point-to-point) synchronization. In this section, we focus on key comparable work. Like barriers, condition variables, and semaphores, phasers are used for coordinating the flow and execution of threads and processes; this is in contrast to mutual exclusion (locks) and transactional memories, which are more typically oriented towards preserving the consistency of data in (potentially) concurrent environments.

The JUC CyclicBarrier class [4] supports periodic barrier synchronization among a set of threads. Unlike Phasers, however, CyclicBarriers do not support the dynamic addition or removal of threads; nor do they support one-way synchronization or split-phase operation.

OpenMP provides directives for barrier synchronization and single-thread execution regions [10]. The single and master directives are used to mark regions that should only be executed by a single thread. The main difference between the two is that the single region may be executed by any one thread in a thread group, but the master region is always executed by the master thread. The single construct presented in this paper serves a purpose similar to the OpenMP single directive, but we require that the single section be executed only after all threads have reached the single statement. The master directive in OpenMP is used to force only the master thread to execute a block of code in a parallel region, and there are no barriers on entry to or exit from the master region. We do not differentiate between single and master regions. Our implementation of phasers allows the master thread (i.e. the thread executing the single statement) to either be fixed for the life of the phaser or to change at each encounter of a single section. Also, in OpenMP, the master and single directives cannot be directly nested under work-sharing constructs (e.g. parallel for), but no such restriction exists for phasers. Perhaps the most important difference between OpenMP single directive and the phaser single statements is that threads can be dynamically added to the group of threads participating in a phaser with a single statement, but the number of threads participating in an OpenMP single section is fixed on entry to a parallel region.

Titanium is a dialect of Java for SPMD parallelism [8]. The language has a notion of single values which are the values used to ensure coherence of synchronization points. The designers want to statically ensure that the sequence of global synchronizations is identical across all processors so they make a conservative check to ensure that the statements with global effects is coherent across all processors. They define specific rules for constructing expressions that may have global effects to ensure that the coherence condition can be checked statically. Our phasers do not require all activities to reach the same synchronization point except in the case of next-with-single barriers. The runtime performs a dynamic check that all activities execute the same next-with-single, rather than the conservative static check used in Titanium.

Gupta’s work on fuzzy barriers [5] introduces the concept of overlapping synchronization with “real” work in a model that is now widely known as a split-phase barrier. In this paradigm, a region of instructions is targeted for execution between the point when a processor enters the barrier and when is waits for all other processors to also enter. When processors enter the barrier synchronization region, they signal to the other processors that they are ready to synchronize. They may continue to execute any instructions in the synchronization region; but before a processor leaves the synchronization region, it must receive a notification from all processors indicating that they have entered the synchronization region. Unlike Gupta’s work, in which compiler analysis uses low-level code motion to select a region of code for execution after signaling the barrier, the signal
**Figure 11:** BarrierBench microbenchmark results on 64-way Sun UltraSPARC T2

**Figure 12:** Speedup relative to serial Java version on 64-way Sun UltraSPARC T2
and next operations in Phasers allow explicit programmer control over when code is executed.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced phasers, a new coordination construct that unifies collective and point-to-point synchronizations. We established two safety properties for phasers: deadlock freedom and phase-ordering. Performance results obtained from a portable implementation of phasers on two different SMP platforms demonstrate that they can deliver superior performance to existing barrier implementations, in addition to the productivity benefits that result from their generality and safety properties. Opportunities for future research related to phasers include extensions for reduction and collective operations, implementations on heterogeneous multicore processors (such as the Cell) and on distributed-memory clusters, and new compiler analyses and optimizations for phaser operations. We also believe that the ability for a thread to mark completion of multiple phases via a single signal statement could further reduce runtime overheads for certain workload scenarios.

Acknowledgments

We would like to thank Nathan Tallent for his feedback on this paper, and Rajkishore Barik for his contributions to early discussions related to phasers. We are grateful to all X10 team members for their contributions to the X10 software used in this paper. We would like to especially acknowledge Vijay Saraswat’s work on the design and implementation of clocks in the current X10 SMP implementation. We gratefully acknowledge support from an IBM Open Collaborative Faculty Award, and from Microsoft fund R62710-792. Finally, we would like to thank AMD for their donation of the 8-CPU Opteron SMP system, the IBM Poughkeepsie Benchmark Center for access to the 64-CPU Power5+ SMP system, and Doug Lea for access to the UltaSPARC T2 SMP system.

7. REFERENCES