Virtualizing Transactional Memory

Ravi Rajwar
Microarchitecture Research Lab
Intel Corporation
ravi.rajwar@intel.com

Maurice Herlihy[1]
Computer Science Department
Brown University
mph@cs.brown.edu

Konrad Lai
Microarchitecture Research Lab
Intel Corporation
konrad.lai@intel.com

Abstract

Writing concurrent programs is difficult because of the complexity of ensuring proper synchronization. Conventional lock-based synchronization suffers from well-known limitations, so researchers have considered non-blocking transactions as an alternative. Recent hardware proposals have demonstrated how transactions can achieve high performance while not suffering limitations of lock-based mechanisms.

However, current hardware proposals require programmers to be aware of platform-specific resource limitations such as buffer sizes, scheduling quanta, as well as events such as page faults, and process migrations. If the transactional model is to gain wide acceptance, hardware support for transactions must be virtualized to hide these limitations in much the same way that virtual memory shields the programmer from platform-specific limitations of physical memory.

This paper proposes Virtual Transactional Memory (VTM), a user-transparent system that shields the programmer from various platform-specific resource limitations. VTM maintains the performance advantage of hardware transactions, incurs low overhead in time, and has modest costs in hardware support. While many system-level challenges remain, VTM takes a step toward making transactional models more widely acceptable.

1. Introduction

Multicore architectures present both an opportunity and challenge for multithreaded software. The opportunity is that threads will be available to an unprecedented degree, and the challenge is that more programmers will be exposed to concurrency-related synchronization problems that until now were of concern only to a select few.

The limitations of conventional synchronization techniques, based on locks and condition variables, are well-known [10]. Coarse-grained locks, which protect relatively large amounts of data, simply do not scale well. Threads block one another even when they do not really interfere, and the lock itself becomes a source of contention. Fine-grained locks are more scalable, but they are difficult to use effectively and correctly. In particular, they introduce substantial software engineering problems, as the conventions associating locks with objects become more complex and error-prone. Locks also cause vulnerability to thread failures and delays: if a thread holding a lock is delayed by a page fault, or context switch, other running threads may be blocked. A thread failure also leaves shared objects with inconsistent updates. Locks also inhibit concurrency because they must be used conservatively: a thread must acquire a lock whenever there is a possibility of synchronization conflict, even if such conflict is actually rare.

Transactional memory [10] addresses these limitations. A transaction [5] is a finite sequence of memory reads and writes executed by a single thread. Transactions are atomic: each transaction either completes and commits (instantaneously taking effect) or aborts (discarding its updates). Transactions are serializable: they appear to take effect in a one-at-a-time order. Each thread announces the start of a transaction, executes a sequence of operations on shared objects, and then tries to commit the transaction. The updates take place if the commit succeeds.

High performance transactional memory implementations [2, 7, 10, 13, 20, 26] exploit hardware mechanisms such as speculative execution and on-chip caching. Hardware optimistically executes a transaction and locally caches memory locations read or written on behalf of the transaction, marking them *transactional*. The hardware cache coherence mechanism communicates information regarding read and write operations to other processors. A data conflict occurs if multiple threads access a given memory location via simultaneous transactions and at least one thread's transaction writes the location. A transaction commits and atomically updates memory if it finishes without encountering a data conflict.

Most prior hardware transactional memory proposals require programmers to be aware of platform-specific resource limitations such as cache and buffer sizes, scheduling quanta, and the effects of context switches and process migrations. Transactions that exceed these resources or repeatedly encounter such interruptions cannot commit.

[1]Supported in part by NSF Award Number 0410042 and by gifts from Sun Microsystems and Intel Corporation.

If transactional synchronization is to gain wide acceptance, however, programmers must be shielded from such complex, platform-specific details. Instead, transactional memory systems must provide full guarantees even for transactions that cannot execute directly in hardware.

To ensure high performance, implementations must provide sufficient resources for the vast majority of transactions to execute directly and efficiently in hardware. Prior proposals have demonstrated that most transactions require only modest hardware resources [2, 7, 17, 20]. Nevertheless, repeatedly aborting the few transactions that exceed these limits would severely restrict the usability of the transactional model because it is unreasonable to expect programmers to circumvent such complex limitations by special-case, handcrafted code. The resource exhaustion problem therefore is not one of performance but one of completeness and guarantees.

This paper proposes Virtual Transactional Memory (VTM), a combined hardware/software system architecture that allows the programmer to obtain the benefits of transactional memory without having to provide explicit mechanisms to deal with those rare instances in which transactions encounter resource or scheduling limitations. The underlying VTM mechanism transparently hides resource exhaustion both in space (cache overflows) and time (scheduling and clock interrupts). When a transaction overflows its buffers, VTM remaps evicted entries to new locations in virtual memory. When a transaction exhausts its scheduling quantum (or is interrupted), VTM saves its state in virtual memory so that the transaction can be resumed later.

VTM virtualizes transactional memory in much the same way that virtual memory (VM) [12] virtualizes physical memory. Programmers write applications without concern for underlying hardware limitations. Even so, the analogy between VTM and VM is just an analogy: we will see that the technical problems are quite different.

VTM achieves virtualization by decoupling transactional state from the underlying hardware on which the transaction is executing and allowing that state to move seamlessly (and without involving the programmer) into the transaction's virtual address space. This virtual memory-based overflow space allows transactions to circumvent hardware resource limits when necessary. Since the virtual address space is independent of the underlying hardware, a transaction can be swapped out, or migrate without losing transactional properties.

Nevertheless, using a virtual memory-based virtualization scheme presents challenges. Most importantly, VTM must not slow down the common case where hardware resources are sufficient. By keeping the virtualization machinery off the critical path, we can continue to ensure that the overall performance of the system is determined by the hardware-only case. Virtualization provides essential functionality, but should have an insignificant effect on performance. VTM requires the ability to detect inter-thread synchronization conflicts, and the ability to commit or abort multiple updates to disjoint virtual memory locations in an atomic, non-blocking way. None of this functionality is provided by existing virtual memory systems.

To this end, VTM provides two operational modes: a hardware-only fast mode provides transactional execution for common case transactions that do not exceed hardware resources and are not interrupted. This mode is based on mechanisms proposed by prior work [2, 7, 10, 20], and this mode's performance effectively determines the performance of the overall scheme. A second mode, implemented by a combination of programmer-transparent software structures and hardware machinery, supports transactions that encounter buffer overflow, page faults, context switches, or thread migration.

Overview: In VTM, transactional state is split into two parts: *locally-cached* state resides in processor-local buffers, and *overflowed* state resides in data structures in the application's virtual memory.

The VTM system architecture uses several data structures to track overflowed state. The VTM implementation manages these data structures and ensures that concurrent accesses are properly synchronized.

Each transaction has a *Transaction Status Word* (XSW) that tracks a transaction's status at all times. Since a thread executes one transaction at a time, the status word is associated with a single thread. The VTM implementation commits or aborts a transaction by atomically updating its XSW.

The *Transaction Address Data Table* (XADT) keeps track of transactional state that has overflowed from processors to memory. The XADT is common to all transactions sharing the address space. Transactional state can overflow into the XADT in two ways: a running transaction may evict individual transactional cache lines, or an entire transaction may be swapped out, evicting all its transactional cache lines. An XADT entry records the overflowed block's virtual address, its clean and tentative value (uncommitted state), and a pointer to the XSW of the transaction to which the entry belongs.

Each time a transaction issues a memory operation that causes a cache miss, it must check whether that operation's target conflicts with an overflowed address. VTM could detect such conflicts by "walking" the XADT, but doing so would defeat our goal of making the common case fast. Instead, VTM provides two "fast-path" mechanisms for the common case. First, the *XADT overflow counter* records the number of overflowed entries. Normally, this counter is zero, and is cached locally at each processor, avoiding the need for any traffic. Second, an *XADT filter* (XF) provides a quick way to detect the absence of conflict. A miss in the filter guarantees the address does not conflict, and a hit triggers an XADT walk.

VTM can instantaneously change the status of a transaction's XADT entries by atomically updating its XSW. As discussed in more detail below, this instantaneous

logical commit must be followed by a multi-step *physical* commit to move updated values from the XADT to memory.

We do not focus on mechanisms to make the common hardware-only case faster. Instead, we assume a standard high-performance hardware-only transactional memory implementation, and are agnostic about specific policies and implementations in the hardware-only common case.

Section 2 discusses the necessity and challenges of such virtualization, and the goals of our architecture. Section 3 describes our baseline model. Section 4 describes the VTM system architecture and its components, and Section 5 describes the VTM operations. Section 6 discusses remaining transactional memory challenges, Section 7 presents related work, and Section 8 concludes.

2. VTM: Necessity, challenges, and goals

Necessity: Hardware transactional memory implementations buffer state on a per-processor basis, since successful, uninterrupted transactions use only processor-local resources. Virtualization, however, allows transactions to be suspended, to migrate, or to overflow state from local buffers. Such abilities require decoupling transactional state from processor state for the following reasons:

- Making the hardware buffer sizes part of the architecture and exposing them to the programmer limits implementation flexibility and portability, while not exposing them makes it impossible for the programmer to ensure that transactions can run in hardware across a variety of platforms and applications.
- Hardware buffers lack persistence. A processor is a shared resource typically managed by an operating system. Multiple independent processes run over time, reusing local hardware buffers. A transaction can survive an interrupt only if its transactional state can be moved to persistent space before giving up the processor.

Further, maintaining overflow state on a *per-process* instead of a *per-processor* basis has additional benefits.

- Per-process state maintenance allows processes to be isolated from one another if necessary. An incorrectly or improperly executing application cannot interfere with another application since their address spaces are different.
- The overhead of detecting conflicts among multiple transactions typically depends on how many transactions there are. If potential conflicts are limited to a single process, then we can limit the cost of conflict detection. Moreover, we can limit the interference caused by malicious or erroneously written program.
- Overflowing to virtual memory also allows state to be visible to support software such as debuggers and other profiling libraries, a difficult task if overflow were only in physical space.

VTM tracks overflow state using virtual, not physical addresses. Using physical addresses would require pages (and any pages pointed to by these addresses) to be pinned and marked non-swappable during the transaction's execution, which would require significant operating system involvement.

Virtual memory is universally available, and already handles many complex resource-related problems. Nevertheless, transactional memory virtualization based on virtual memory introduces certain additional challenges.

Challenges: Transactional memory requires the ability to detect synchronization conflicts between transactions. Conflict detection is relatively easy when transactions run entirely in hardware (by exploiting native cache-coherence mechanisms), but additional mechanisms are needed to detect conflicts between active transactions and transactions whose state has partially or completely overflowed to virtual memory.

Transactional memory also requires the ability to commit or abort multiple memory accesses atomically. Here too, atomic commits and aborts are relatively easy for transactions that run in hardware, but we will need to invent new mechanisms to support atomic commit and abort for transactions with partially or completely overflowed state.

Goals: Virtualized transactional memory must satisfy the following requirements:

- The performance of the common-case hardware-only transactional mode must be unaffected.
- Conflict detection between active transactions and transactions with overflowed state should be efficient, and should not impede unrelated transactions.
- Committing or aborting a transaction should not delay transactions that do not conflict.
- Context switches and page faults may impede transaction progress, but must not prevent transactions from eventually committing.
- Non-transactional operations may cause transactions to abort but must never compromise any transaction's atomicity.
- Finally, VTM must be transparent to application programmers.

3. Baseline software and hardware model

An application consists of multiple concurrently executing software threads operating in a single shared virtual address space. Each thread serially executes transactions explicitly delimited by the instructions `begin_xaction` and `end_xaction` (see Figure 1). A fault occurs if a transaction executes an operation that cannot execute optimistically (such as input or output to a device). Nested transactions are allowed and are handled by flattening all inner transactions into the outermost transaction. The hardware tracks the nesting depth to determine when to commit the flattened transaction.

We assume a high-performance hardware transactional memory implementation [2, 7, 10, 20, 26] for the common case where local resources are sufficient. Such processor hardware support includes an architectural register state checkpoint for recovery, ability to execute and speculatively retire instructions in the transaction, to buffer memory updates locally, to track addresses for cached loads and stores to detect memory conflicts, and perform atomic commits and aborts.

When two transactions conflict, a conflict resolution policy decides which one is aborted. Conflict resolution policies might take into account a transaction's age, its operating system thread priority, and a variety of other properties. A complete discussion of this subject is beyond the scope of this paper (see, however, [10, 20]), so we will simply assume that a uniform policy exists.

4. VTM system architecture and design

VTM supports data overflows, conflict resolution among transactions, and atomically committing transaction state. While the typical programmer is not concerned with these components, they are part of the VTM architecture specification. We outline our architectural structures in Section 4.1, where we also discuss how these structures interact with the software. Sections 4.2, 4.3, and 4.4 present details of the new structures, and Section 4.5 describes the overall VTM system.

4.1 VTM Architecture

Hardware transactional memory monitors accesses and updates at the hardware level. VTM complements the hardware by monitoring memory accesses and updates at the application level. In this way, VTM provides application-centric atomicity, instead of hardware-centric atomicity. VTM allows events such as context switches or page faults to occur within a transaction, as long as they do not jeopardize atomicity. A transaction can be temporarily paused, unrelated operations can execute, and the transaction subsequently resumes. However, these capabilities require transaction virtualization to be architecturally visible.

VTM architecturally defines a Transaction Status Word (XSW) for each transaction. The XSW tracks its transaction's state at all times. A transaction is associated with a unique thread (although a thread serially executes multiple transactions), so each XSW is part of a thread's state. An XSW resides in the application's virtual address space and any thread can operate directly on any XSW. The XSW is the ultimate authority on a transaction's status, and a transaction can be committed or aborted by modifying its XSW using an atomic compare-and-swap operation.

VTM also architecturally defines two data structures: the Transaction Address Data Table (XADT), and a filter for this table (XF). The XADT is the central repository for buffering overflowed transaction state and for resolving conflicts involving such state. The XF is a compact representation of the XADT that allows a quick test whether an address has overflowed into the XADT. The XADT and XF are software data structures that reside in the application's virtual address space. All transactions in an application share the XADT and XF.

While these structures reside in addressable memory, access to them is controlled. To the typical programmer, the address space where these structures reside is unavailable. The VTM system, implemented in either hardware or microcode, manages these structures, and performs overflow and conflict detection operations on behalf of the programmer. For example, the programmer performs a series of reads and writes demarcated by the `begin_xaction` and `end_xaction` instructions. If any address accessed within the transaction during execution overflows local buffers, VTM automatically detects the overflow, and performs the necessary adjustments to the XADT. The programmer typically does not observe VTM intervention. Each processor has its own VTM system implementation which acts like a coprocessor (but with state), and operates at the user's privilege level on these structures using cacheable load and store operations. These operations are not part of a transaction. The application automatically initializes the XADT and XF structures and passes their location and bounds to VTM. The VTM system might need to perform adjustments and operations to these structures, beyond loads and stores. For example, if the XADT needs to grow because of very high overflows. In such an event, VTM signals the application, which responds by calling user-level libraries. The application then communicates any adjustments to the VTM system. Because these structures reside in virtual address space, the operating system can swap their pages out to disk. If the VTM system encounters such an access fault, it again signals the application to request page fault servicing on its behalf.

These operations are typically transparent to the application programmer, because VTM, not the programmer, executes synchronized operations on the XADT and XF. However, software such as debuggers, runtime garbage collectors, and so on might need controlled access to the XADT and XF structures, which is why these structures are part of the VTM architecture. The VTM system mechanisms that operate on the XADT and XF, and perform commits and aborts are implementation dependent.

4.2 XSW

A transaction's XSW summarizes the transaction's current execution status, along the following three dimensions.

In the first dimension, a transaction may be running (R), it may have committed but not made its updates visible (C), or it may have aborted (B). Since the XADT resides in virtual memory, physical commits and aborts are a multi-step process. For the second dimension, a transac-

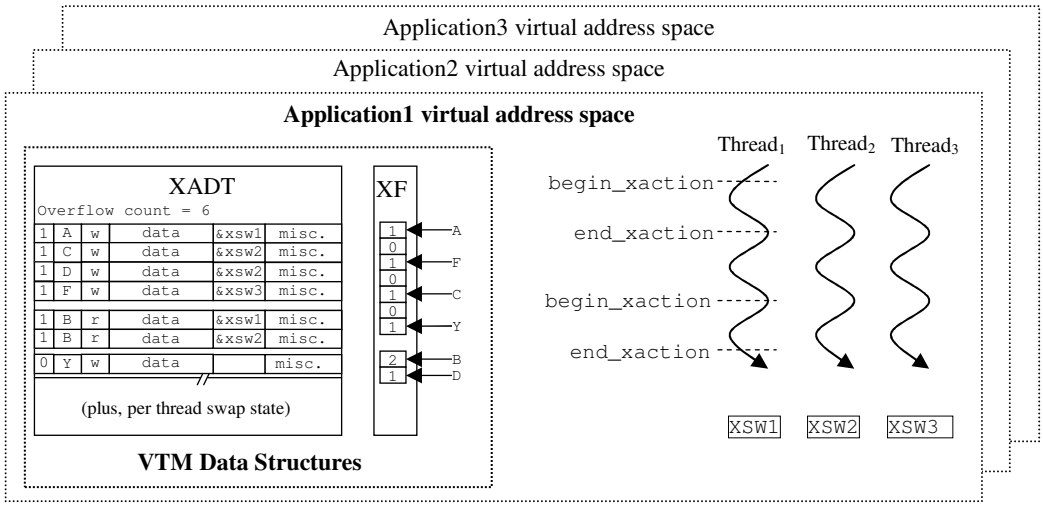

Figure 1 VTM software data structures. A conceptual snapshot of the address space is shown. Threads execute series of transactions. XADT records overflow information, and any swap information. XF summarizes the XADT like a bloom filter. For example, XF has "Y" marked, but "Y" is invalid in XADT.

tion is either actively executing (A) or has been swapped out (S). For the third dimension, a transaction's state is either completely cached locally (L), or has partially overflowed (O).

Only a few combinations are valid. For example, a swapped transaction cannot have a completely locally cached state. Aborted and committed transactions cannot have locally-cached states since aborts and commits for hardware-only transactions are instantaneous. A transaction in the process of updating memory completes updates before swapping out. In the end, only the following state combinations are valid: RAL: running, actively executing, with locally-cached state, RAO: running, actively executing, overflowed, RSO: running, swapped out, overflowed, BAO: aborting, actively executing, overflowed, BSO: aborting, swapped, overflowed, and CAO: committing, actively executing, and overflowed. A thread not executing a transaction is in NonT (non-transactional) state.

The transaction is globally ordered when its XSW status successfully transitions to a committing state. All operations in the transaction are globally ordered atomically at this point. The committing state captures a *logical commit* of the transaction. The physical commit, where updates become visible, is implementation dependent, but must ensure the logical atomicity of updates.

4.3 XADT

The XADT is the central structure responsible for managing data overflow. All transactions executing within a virtual address space share the same XADT. A hardware (or firmware) structure, the XADT *walker*, manages the XADT. XADT load and store operations executed by the XADT walker are cache coherent. Key XADT operations include adding an entry on overflows, entry lookup using an address, commit and abort operations via actions on the XSW, and saving state on context switches. The XADT also records the nesting depth (maintained by the hardware) of a swapped transaction.

The XADT also records an *overflow count* of overflowed data blocks at any time. This count provides a fast way to determine whether any overflows have occurred, which is none in the common case. An XADT entry includes at least the following fields: a) Status bits marking whether the entry is valid and whether a transaction read or wrote the address, b) virtual address of the data block overflowed to this entry, c) data field for buffering updates to the overflowed location, and d) pointer to the overflowing transaction's status word (XSW). The address of the data field serves as the remapped address for the overflowed data. Miscellaneous fields for conflict prioritization may use timestamps and information such as thread priority from the operating system. XADT entries belonging to the same transaction are linked together to allow efficient cleanup on aborts and commits. An address that is concurrently being read by multiple transactions would have multiple XADT entries.

The transactional state also includes state saved at context switches, such as temporary register state (since the transaction is executing speculatively), any temporary updates performed in the local caches, and the virtual address and data value of any cache blocks read during the transactional execution, even if the blocks have not been written. As discussed in detail below, recording the clean values of the data blocks allows a rescheduled transaction to ensure that there were no conflicting non-transactional operations while it was swapped out.

We expect the combination of the overflow count and the conflict filter (discussed below) to minimize the actual number of accesses to the XADT.

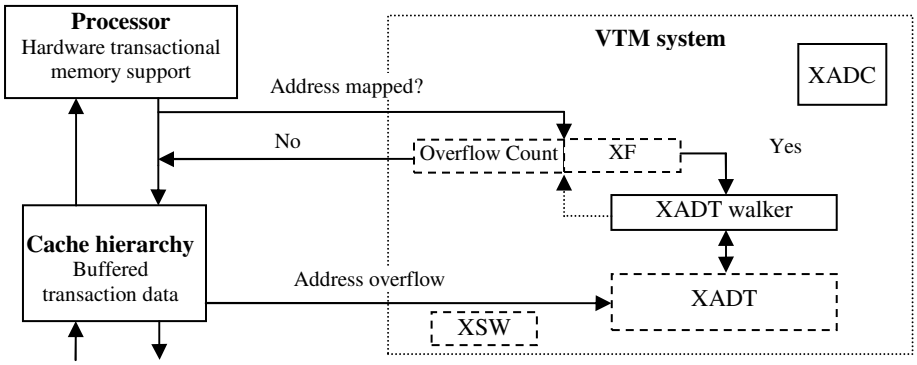

Figure 2 VTM overview. Each processor has its own VTM system machinery. The software-resident XADT and XF data structures are shown with dashed boxes, and the hardware structures are shown with solid boxes. The VTM machinery operates on the XF and XADT using cacheable operations. The XADC caches remapping translation information for overflowed blocks.

4.4 XF

When a transaction issues a read or write request that causes a cache miss, VTM must quickly determine whether the request conflicts with a request already issued by another transaction. The existing cache coherence protocol will detect conflicts involving locally-cached state, but it cannot detect conflicts involving overflowed state. For example, a transaction that has been swapped out does not participate in the cache coherence protocol. (Requests that hit in the cache do not require conflict resolution.)

One way to detect conflicts involving overflowed state is to call the XADT walker. Because such conflicts are uncommon, however, the XF conflict filter provides a fast "out-of-band" way to detect the absence of conflict in most cases, thus avoiding the need to walk the XADT. A miss in the XF means that an address does not conflict with any overflowed block, while a hit means that a conflict probably exists. On a hit, the requestor's VTM walks the XADT and determines whether the conflict actually exists. Since the XF is a conservative representation of the XADT, updates to the XF can be performed lazily.

Bloom filters as conflict detectors: A *Bloom filter* [3] is an efficient set data structure that provides two operations: *add(x)* inserts x into the set and *member(x)* queries whether x is in the set. *Member* is allowed to produce (infrequent) false positives. The filter itself is an n-element Boolean array B, initially all *false*, and a set of independent hash functions, $h_1,\ldots h_k$. To add x to the set, set $B[h_i(x)]$ to *true*, for all i. A query testing whether a particular y is in the set returns *true* if $B[h_i(y)]$ is *true*, for all i. It is easy to see that if the query returns *false*, then y is not in the set, while if it returns *true*, then it might be. Classical Bloom filters do not permit elements to be removed. A *counting Bloom filter* [6] replaces the Boolean array B with an array of counters C. Adding an element x (atomically) increments each $C[h_i(x)]$, and removing the element (atomically) decrements each such counter. A *member(x)* query returns *true* if every $C[h_i(x)]$ is non-zero.

The XADT filter (XF) is a counting Bloom filter that summarizes the XADT. The probability of a false positive can be made arbitrarily small by choosing enough counters. Analysis of the trade-offs among filter size, counter size, and the number of hash functions can be found in the literature (see, for example, [6]), Experimental work [21] suggests that a family of linear congruences work well for hash functions. Others [1, 23] have discussed hardware filter implementations.

XF representation and design: A concrete representation of a counting Bloom filter must address several questions: how many hash functions and counters to use, how large are the counters, and how are they represented in memory? If m entries are placed in an n-element Bloom filter using k hash functions, then the likelihood of a false positive is bounded by $(1-e^{-kn/m})^k$. The probability that some counter in the table will exceed i is less than $m(c \ln 2/i)^i$. Assume we are willing to tolerate a 1% probability of false positives when there are one million (2^{20}) overflowed blocks. (Fewer blocks means lower probabilities, while more means higher.) If the filter uses 2 hash functions, then it requires at least 21.0M counters, while if it uses 4, it requires 12.5M counters. If each counter has 4 bits, then the probability that 1M blocks will cause any counter to overflow in either case is less than 1.44×10^{-09}. (Overflow is a performance, not a correctness issue, because impending overflows can be detected and redirected to an overflow table.) In practice, of course, hash functions will not be perfectly random, plus we will keep adding and removing blocks, but 4 bits per counter seems more than adequate.

XF implementation options: The most straightforward way to implement the XF is as an array of 4-bit counters. At 2 counters per byte, 21.0M counters occupy 10.5Mbytes (6Mbytes with 4 hash functions). On many platforms, such an array is small enough to reside in memory, so paging is unlikely to be an issue. False sharing is unlikely to be an issue as long as updates (i.e., overflows) are rare. Atomic increments and decrements can

be implemented using compare-and-swap instructions or the equivalent.

A more compact representation is to use a hash table mapping counters to non-zero values. (A missing mapping is interpreted as zero.) The advantage is that the hash table size is largely determined by the actual number of overflows, not the total number of counters. If we use 4 bytes per entry (3 to index the counter and 1 for its value), then 1M blocks need roughly 4Mbytes. The disadvantage is that *lookup*, *add*, and *delete* operations are more complex.

Perhaps the most attractive alternative is to use a hybrid structure. Split the XF into a bitmap that identifies which values are non-zero, with a hash table holding the actual values as above. For 20.5M counters, the bitmap occupies less than 1Mbyte, and commonly occurring *lookup* operations need never visit the hash table. *Add* and *delete* operations must still access the hash table, and care must be taken that combined updates to the bitmap and hash table are properly synchronized

The amount of traffic to the XF depends on the locality of the transaction. A transaction references the XF only when it takes a cache miss and each such miss produces k independent references to the XF, where k is the number of hash functions used (2 in the example given above).

4.5 VTM system overview

Figure 2 shows the VTM system architecture. The principal architectural data structures, XSW, XF, and XADT, are shown as dashed boxes. The solid boxes show the VTM implementation-dependent hardware components, the XADT walker and the XATC. The hardware components act like coprocessors.

When a processor executing a transaction misses in its cache, VTM checks whether that address was overflowed by another transaction. It first tests the XADT overflow count. Most often, this count is zero, cached, and accessed with the latency of a cache hit.

If the overflow count is non-zero, then VTM consults the XF. If the XF hits, then the VTM calls the XADT walker to identify the conflict, if it exists. This sequence is similar to a hardware page-miss handler determining a missing translation.

In VTM, the requesting processor's VTM system performs conflict detection for overflowed blocks prior to generating the request, and operates on the XADT and XF. This localizes conflict detection, allows conflicts with swapped transactions to be detected, and avoids unnecessary interference with other processors, all of which otherwise would have had to perform XADT and XF operations to determine conflicts based on an incoming request. The non-overflowed blocks are handled conventionally by the other processors (e.g., as in TLR [20]).

5. VTM system operational details

We now discuss VTM in more detail. Figure 3 shows transactional state transitions, which occur when VTM updates the transaction's XSW. First, we describe VTM operations for the hardware-only mode to demonstrate how VTM virtualization does not slow down the common case (Section 5.1). Next, we describe how VTM in virtualization mode provides four basic functions: managing data blocks that overflow from local hardware buffers, suspending and swapping interrupted transactions, detecting conflicts for data that has overflowed, and atomically committing and aborting overflowed transactions, discussed in Sections 5.2 through 5.5. Finally, we discuss how VTM interacts with page faults in Section 5.6.

5.1 VTM hardware-only operational mode

All threads begin in a transaction state NonT, as shown in Figure 3. The RAL state is the hardware transactional memory mode where the transaction executes and commits using only processor-local resources. The critical performance path is the NonT to RAL to NonT transition (begin and commit/abort). As discussed, VTM avoids slowing down this critical path by testing the XADT overflow count, which in the common case (of no overflows) takes the same latency as a local cache hit. The VTM overflow management and conflict detection machinery is invoked only if an overflow has occurred.

5.2 Managing data overflow

A transaction that evicts a transactional cache line transitions from the RAL to the RAO state, shown in Figure 3. The cache's LRU policy determines which data block to overflow, thus maintaining locality. The VTM machinery allocates a new XADT entry as necessary. The VTM machinery locally caches this information (e.g., the new address for the data field) in the XADC (the XADT cache), to speed subsequent accesses to this overflowed block. At this point, non-overflowed blocks reside in local buffering and overflowed updates in the XADT. The state flow between local hardware and the XADT is transparent to the programmer.

When a processor overflows a transaction block, another processor may already have locally cached the block as part of its hardware-only transactional execution. In such an event, the XADT (and the XF) would not have an entry for such a block. To ensure that any remote processor detects this conflict, the overflowing transactions' VTM system updates the XF, and sends coherence invalidation for the overflowed block address. This step forces any remote processors' VTM system concerned with the block to re-read the XF for that block, and detect a potential overflow.

5.3 Detecting conflicts with overflowed data

If a memory access results in a cache miss, and if the XADT overflow count is non-zero, then VTM consults the XF. If the XF returns a miss, no conflict exists. Since the XF is mostly read-only, and if this specific address did not overflow, the test will be quick and typically hit in the

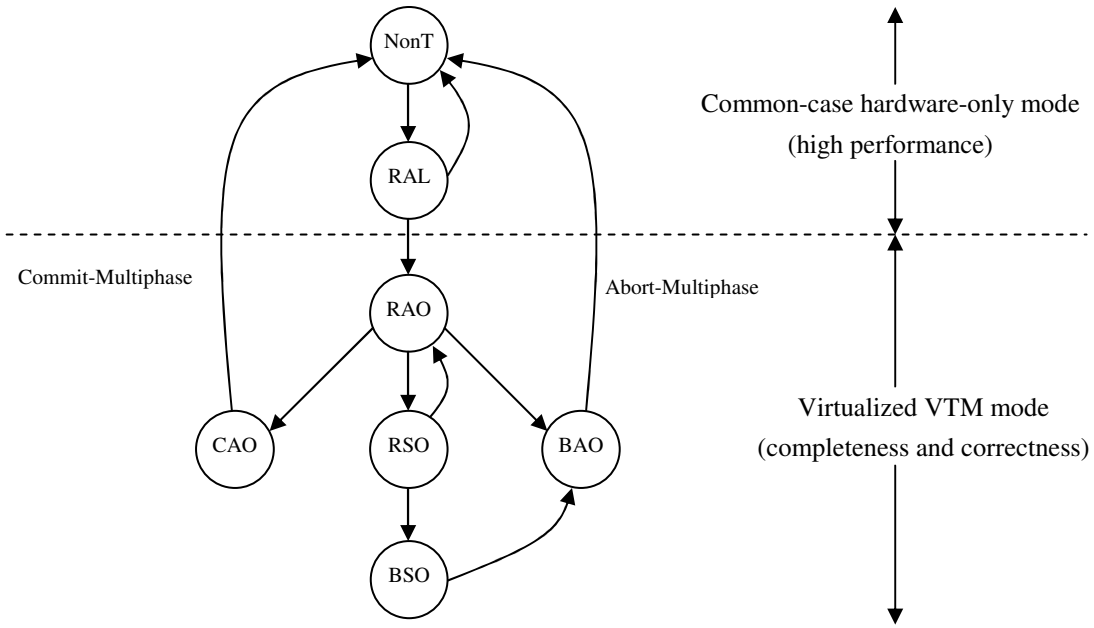

Figure 3 VTM state transition diagram. NonT: Not executing a transaction, R: running, C: committing, B: aborting, A: actively executing, S: swapped out, L: all local hardware, O: overflowed state.

local cache hierarchy. An XADT walk occurs only if the XF returns a miss, and it becomes necessary to determine if the conflict exists.

We have described how VTM detects conflicts among transactions. We would also like to guarantee that synchronization conflicts between transactional and non-transactional operations do not threaten transactions' atomicity. Existing proposals (for example, TLR) provide this guarantee for transactions that that run entirely in hardware. For transactions that do overflow, it would be relatively easy to ensure atomicity by forcing each non-transactional operation to consult the XF and XADT. Nevertheless, we consider such an approach to have unnecessarily low performance. Instead, let us consider what kind of conflicts might occur. A non-transactional operation reads or writes a single memory location, so it is enough to ensure that any such operation can be ordered either before or after any concurrent transaction. A transaction never releases uncommitted data, so a non-transactional operation cannot read a value that is later aborted. The following scenario illustrates how serializability can still be violated. Initially, the address hold the value *v*. A transaction reads *v*, and then a non-transactional operation writes *v''* to that address. The transaction computes *v'* from *v*, writes *v'*, and commits. The problem is that writing *v''* cannot be serialized either before or after the transaction. As discussed below, the fix is to ensure that when an overflowed transaction commits, the values it read are still correct.

When a processor overflows a transactional block, it poisons that block's set in its cache. Only external non-transactional operations to that set will alert the processor of a possible conflict, causing it to check the XF to determine whether the conflict is real. If so, it aborts the conficting transaction.

5.4 Suspending and swapping transactions

On a context switch, VTM overflows all locally buffered transactional state (memory and processor) to the XADT. To facilitate forced overflows, the VTM machinery also records the virtual addresses for locally cached transactional blocks. The clean value for locally buffered but temporarily updated cache blocks is available in memory. After the forced overflow, the hardware buffers have no transactional state: it is all in the virtual memory-based XADT. When a transaction is suspended, the transaction transitions from the RAL state to the intermediate RAO state and finally to the RSO state.

When the transaction re-schedules, it re-populates its cache hierarchy on demand. When a suspended non-aborted transaction is re-scheduled, it transitions from the RSO state to the RAO state; else, it transitions from the BSO state to the BAO state (an active transaction might have aborted the transaction while it was swapped out). If the transaction did not abort, the processor-architected state is restored and execution resumes. VTM re-caches data blocks as necessary and updates the XADT and XF to reflect the transition to hardware mode for those blocks.

As noted, because non-transactional operations do not consult the XF and XATT, a non-transactional write may have overwritten a value read by a swapped-out transaction. To detect such conflicts, when VTM re-schedules a transaction it must check that the values the transaction

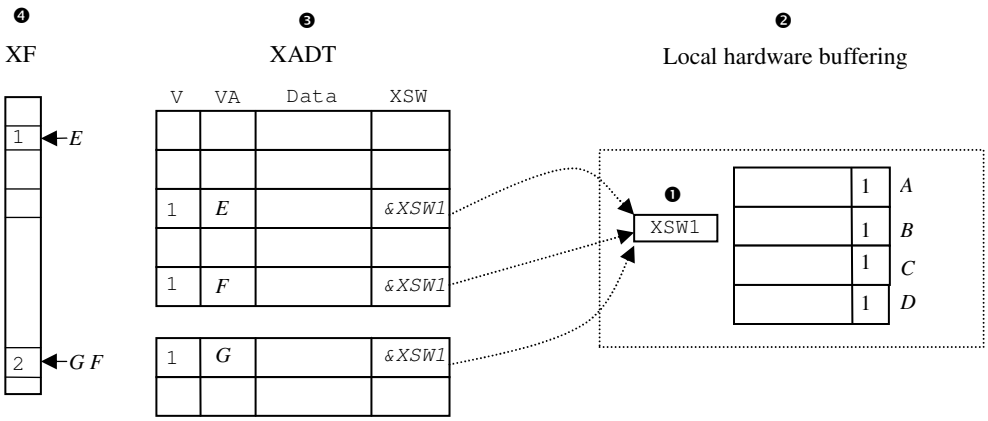

Figure 4 Commit sequence for virtualized transactions in VTM. Here, locations G and F map to the same XF entry.

read agree with the current memory values. A similar value-based validation was proposed by Martin et al. [16].

5.5 Committing and aborting transactions

Commits and aborts require atomic updates to both locally cached and overflowed state. As noted, VTM logically commits a transaction by atomically updating its XSW status, and then physically commits its state by marking local hardware state committed and copying the overflowed updates one-at-a-time from the XADT to memory. In a similar way, VTM logically aborts by updating its XSW status, and physically aborts by marking local hardware state invalid and discarding overflowed updates from the XADT. Logical aborts and commits atomically update the XSW, indirectly marking all associated XADT entries (which have pointers to the XSW) as either aborted or committed. If another transaction detects a conflict with a transaction that has physically but not logically committed, then it stalls until the physical commit completes. This approach is similar to commit protocols used by software-only transactional memory proposals [9]. Because aborting or committing a transaction requires access to the XADT entries belonging to the transaction, these entries may be linked together (by extending the XADT entry fields) to speed traversal. Detecting and resolving conflicts with non-transactional operations during the physical commit requires ensuring these operations do not observe stale memory values (locations that have not yet been updated from the XADT), and we discuss this below.

5.5.1 Committing transactions

Figure 4 describe the commit operation for an overflowed transaction. A completed transaction is about to execute the `end_xaction` instruction. The hardware implementation ensures appropriate local cache transactional state is writable. First, the transaction atomically updates its XSW (❶). If the status transitions successfully to CAO, the transaction has started the commit sequence, and cannot abort. The XADT entries are automatically marked committed through the committing transaction's XSW(❸).

Since the transaction will not abort, local hardware state is atomically committed (❷). Incoming requests can observe the updated hardware state (A, B, C, D). As shown in the figure, the overflowed state E, F, and G, is in the XADT, and access to these locations is controlled by VTM during the commit—any access to these blocks may wait (or return the latest value from the XADT) but cannot return the old value in the original memory location. This ensures logical atomicity of the local and overflow updates. Since access to overflowed data is controlled at all times in the commit sequence, even if the commit sequence is interrupted, atomicity is maintained.

The committing transaction's VTM then updates the original location of the overflowed blocks with corresponding data from the XADT and frees the XADT entry. When an XADT entry is committed to the original locations, the XF entry for that location is updated to reflect XADT changes. This update can occur at any time as long as it is after the update of the XADT (❹).

Even though non-transactional operations typically do not consult the XF and XADT, they need to access these structures only during the commit sequence to ensure the commit itself is atomic (a committing transaction cannot abort). To ensure non-transactional operations do not read inconsistent state during commit (since updating memory locations is a multi-step process), the committing transaction needs to inform other threads to access the XADT and XF during the commit sequence. One way to achieve this is for the VTM system to maintain a count tracking currently committing transactions. Operations on this counter involve atomic increments and decrements, and the counter is non-zero only if an overflowed transaction is committing. When this happens, the VTM system of a processor can ensure its non-transactional operations consult the XF and XADT. Alternative implementations can be hardware oriented, employing broadcast messages.

5.5.2 Aborting transactions

A transaction can abort another transaction by atomically updating the other transactions' XSW. Since the XSW is cached by the local VTM machinery at all times, a running transaction will detect the abort. If the aborted transaction was not running, it will detect the abort when it re-schedules. An aborting transaction discards its local speculative state, and restarts execution. Entries in the XADT corresponding to the aborted transaction are invalidated. This cleanup can occur lazily since those entries are marked aborted and their XSW pointer cleaned up. However, to allow the aborting transaction to restart right away and execute, the XSW should be re-usable. Since the XSW is thread-specific, a local pool of XSWs can be used to allow an aborting transaction to restart execution in parallel with the XADT cleanup. The programming model might allow programmers to abort transactions explicitly, causing a similar sequence of events.

5.6 VTM and page faults

Page faults can occur during the execution of a transaction. If an address accessed by the transaction is unmapped, the operating system's page fault handler executes. This would be legal even if the transaction later aborted because even though the transaction is executing optimistically, it is following a valid execution path (unlike say, in an out-of-order processor where the instruction must first become non-speculative because the data inputs itself to the execution path may be incorrect and the path invalid). To handle the page fault, VTM can either suspend the transaction (similar to the pause-and-resume sequence) and request fault handling, or may request fault handling and then restart the transaction. If the page of a previously accessed address in a transaction is unmapped, the address would have overflowed and would also reside in the XADT. Thus, un-mapping an address would not necessarily result in a transaction abort.

The VTM system itself may generate page faults because its data structures, XADT and XF, reside in virtual memory. A processor's VTM machinery would signal a page fault request to the application if access to an XADT or XF results in a page fault, and user-level libraries would trigger the handling. The program counter would correspond to the instruction that resulted in the access to the XADT and XF. Such faults can occur either during a forced overflow (of cached state) of a transaction because of a context switch, or during the commit sequence when data is copied from the XADT to the original memory location. All faults can be incrementally handled without requiring the transaction itself to abort. In the context switch case, the transaction restarts in an explicit overflow mode, forcing all state to be overflowed, and then incrementally handling page faults. During the commit sequence (which cannot be aborted), the operating system must ensure that when the VTM system commits an XADT entry, both the original address and the XADT entry for that address are mapped at the same time for the duration of committing that entry's update. This allows the data to be copied from the XADT to the original location. The only implication in the worst case of all accesses sequentially experiencing page fault is performance. When the page fault occurs during a commit, the transaction does not abort, and requests fault handling

Note that it is always possible to write a transaction that accesses so many pages that overwhelms the paging machinery itself. Our goal is to ensure that the paging behavior of a transaction is not substantially worse than a non-transactional computation with the same footprint.

6. Open challenges in transactional memory

We have focused on identifying requirements and providing mechanisms for key system-level virtualization mechanisms for transactional memory and have avoided dictating implementation or the user-level API details. However, key challenges remain, some with VTM, and others with the transactional memory API itself.

To virtualize transactions, VTM assumes that multiple *threads*, each executing a transaction, share a single virtual address space associated with the *process* under which they are executing. However, some virtual memory implementations use *virtual address aliasing* to allow sharing between two *processes* executing under different virtual address spaces. The operating system in such situations explicitly maps different virtual addresses from different address spaces to the same physical address. VTM would require additional mechanisms to support interactions among processes from different virtual address spaces. The respective XADT structures would need to communicate, and we leave this as future work.

VTM currently does not define the effects of operating system calls performed within a transaction. A straightforward approach is to provide the operating system with its own XADT structure, and the ability to undo privileged changes. While no fundamental obstacles exist, the operating system would need to be aware of the support, and we leave this as future work.

The role of I/O within a transaction is unresolved. What should it mean for a transaction to write to a memory-mapped device or to use DMA to move data from part of memory to another? For some I/O operations, a log could be introduced as a main memory structure that is written by transactions and spooled to disk, as occurs in databases. The behavior for other I/O operations needs to be driven by the transactional memory usage model.

The behavior of an exception, such as divide-by-zero, thrown inside transactions has to be defined based on the usage model. Exception behavior is also influenced by how nested transactions are handled. VTM currently flattens nested transactions into the top-level transaction: an abort restores state to the beginning of the outermost transaction. However, software engineering and pro-

gramming methods may require finer nested recovery capability. VTM can be adapted to allow such behavior. The challenge here is not so much implementing the desired behavior as deciding what that behavior should be.

These are unresolved questions about the user-level API and the underlying transactional memory model, and researchers must address them to make transactional execution a reality. The VTM design will have to evolve to accommodate behaviors deemed necessary.

7. Related work

Lamport introduced lock-free synchronization to allow multiple threads to work on a data structure concurrently without a lock [14]. Knight investigated architectural support for multi-word synchronization and proposed the use of cache coherence protocols and hardware to add parallelism to mostly-functional LISP programs [13]. The load-linked/store conditional instructions allow for an optimistic atomic read-modify-write on a single word [11].

The IBM 801 storage architecture [4] provided implicit hardware transaction functions, using transaction mechanisms for locking and logging, on virtual storage access to files. The architecture focused on database systems and provided durability.

Transactional Memory [10] and the Oklahoma Update [26] were hybrid hardware/software schemes and provided optimistic read-modify-write on multiple locations. They allowed the programmer to write explicitly transactional code using extensions to the instruction set and cache coherence protocols. These proposals did not provide a solution to handling overflows other than requiring the programmer to handle them.

Software transactional memory [8, 9, 24] uses software primitives to implement transactions. They require careful programming methodologies and do not provide atomicity of a transaction with respect to other operations that do not occur within transactions. Further, they suffer from poor common-case performance.

Speculative Lock Elision [19] and Transactional Lock Removal [20] are hardware proposals that take existing lock-based programs, and execute them in a lock-free manner to attain transactional behavior. These schemes explicitly acquire the lock if the lock-free transactions experience resource overflow. In such an event, the execution can no longer abort. In Transactional Coherence and Consistency [7] all computations occur within a transaction. All transactions execute speculatively in the cache, and on commit, broadcast their updates to all other processes, who then detect conflicts. If transactions experience resource overflows, the execution becomes non-speculative and the execution cannot abort. Since the above schemes cannot abort execution in the presence of insufficient local buffering, they do not provide transactional memory behavior in the presence of overflow and rely on the programmer to ensure this does not happen.

Thread-Level Transactional Memory [17] proposes the use of a thread-level log to allow the software to perform recovery in the event of aborts for overflowed transactions. They show that overflowed transactions are rare.

Unbounded Transactional Memory (UTM) [2] is an alternative scheme for freeing transactions from dependence on hardware resources. One important difference between VTM and UTM is that conflict detection in UTM is considerably less efficient in the normal case. In UTM, each transaction maintains an *xstate* data structure roughly comparable to our XADT. Each memory block has an associated *log pointer* to information about that block in the xstate. When a transaction encounters a cache miss on a load or store, it must check that the location accessed does not conflict with an overflowed entry by reading that location's associated log pointer. (If non-transactional operations are not to jeopardize transactional atomicity, each non-transactional loads and stores must do the same.) In the normal case, where there are no actual conflicts, reading a log pointer on each memory access is slower than reading the locally cached XADT counter. Moreover, keeping one log pointer per memory block takes up substantially more space than our XF filter, possibly affecting the cache hit rate. LTM [2] is a version of UTM that only handles buffer overflow. A special overflow area in processor-local physical memory is used as an extension of the cache, and overflowed blocks are chained together to facilitate lookups. The duration of transactions must be less than a time-slice and transactions cannot migrate. A similar overflow scheme was used by Prvulovic et al. for use in speculative thread-level parallelization [18].

Thread-level speculation (TLS) techniques use hardware support to speculatively parallelize sequential programs [13, 25]. While such speculative multithreading techniques use some of the hardware mechanisms required for transactional memory, critical differences exist. These techniques do not automatically provide transactional memory semantics because in such techniques, one thread is always non-speculative and cannot abort.

Transactional memory is concerned with providing multiprocessor synchronization but not with ensuring that updates survive crashes. By contrast, “lightweight” transaction systems such as RVM [22] and Rio [15] are concerned with the complementary problem of providing durability but not synchronization.

8. Concluding remarks

Transactional memory avoids software engineering and reliability problems associated with lock-based synchronization when developing multithreaded programs. Hardware implementations of transactional memory allow transactional memory models to achieve high performance with respect to other lock-based schemes, but expose programmers to low level hardware implementation, since hardware-resident transactions will always be limited in size and scope. This paper’s premise is that transactional

memory can realize its promise only if programmers are shielded from low-level hardware constraints of high performance transactional memory implementations.

Virtual memory simplified memory management where programmers no longer had to worry about overlays when dealing with physical memory. Supporting virtual memory was not simple, but the benefits far outweighed virtual memory's initial cost and complexity. VTM adopts this approach and virtualizes hardware transactional memory implementations. VTM operations, and accesses to its own data structures, though subtle, are hidden from the programmer. This paper demonstrates that transactional memory virtualization is possible in a way that does not slow down the hardware-only transactional memory operations.

Significant work remains in the software model development for transactional memory in large-scale applications. By demonstrating transactional memory virtualization in this paper, we hope that software developers can reason with transactional memory without worrying about the underlying implementation or constraints, thus making transactional memory more attractive and compelling.

Acknowledgements

We especially thank Jim Smith for discussions and comments on the paper. We thank Haitham Akkary, Iris Bahar, Jim Goodman, and Eric Rotenberg for comments on earlier drafts, and Galen Hunt, Jim Larus, and David Tarditi for discussions regarding the ideas in the paper.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

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

[3] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), 1970.

[4] A. Chang and M. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[5] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11), November 1976.

[6] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networks*, 8(3), 2000.

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

[8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.

[9] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, July 2003.

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

[11] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Lawrence Livermore National Laboratory, Technical Report UCRL-97663, November 1987.

[12] T. Kilburn, D. B. J. Edwards, M. J. Lanigan, and F. H. Sumner. One-Level Storage System. *IRE Trans. Electronic Computers*, 11(2), April 1962.

[13] T. F. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of ACM Lisp and Functional Programming Conference*, August 1986.

[14] L. Lamport. Concurrent Reading and Writing. *Communications of the ACM*, 20(11), November 1977.

[15] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[16] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly Implementing Value Prediction in Microprocessors That Support Multithreading or Multiprocessing. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[17] K. E. Moore, Thread-Level Transactional Memory. presented at *Wisconsin Industrial Affiliates Meeting*, October 2004 http://www.cs.wisc.edu/multifacet/papers/affiliates04_tltm.pdf

[18] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[19] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[20] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[21] M. V. Ramakrishna. Practical Performance of Bloom Filters and Parallel Free-Text Searching. *Communications of the ACM*, 32(10), 1989.

[22] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1), 1994.

[23] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[24] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, August 1995.

[25] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[26] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel & Distributed Technology*, 1(4), November 1993.