Tradeoffs in Transactional Memory Virtualization

JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom,
Christos Kozyrakis and Kunle Olukotun

Computer Systems Laboratory Stanford University
{jwchung, caominh, austenmc, travissk, hchafi, bdc, kozyraki, kunle}@stanford.edu

Abstract

For transactional memory (TM) to achieve widespread acceptance, transactions should not be limited to the physical resources of any specific hardware implementation. TM systems should guarantee correct execution even when transactions exceed scheduling quanta, overflow the capacity of hardware caches and physical memory, or include more independent nesting levels than what is supported in hardware. Existing proposals for TM virtualization are either incomplete or rely on complex hardware implementations, which are an overkill if virtualization is invoked infrequently in the common case.

We present *eXtended Transactional Memory (XTM)*, the first TM virtualization system that virtualizes all aspects of transactional execution (time, space, and nesting depth). XTM is implemented in software using virtual memory support. It operates at page granularity, using private copies of overflowed pages to buffer memory updates until the transaction commits and snapshots of pages to detect interference between transactions. We also describe two enhancements to XTM that use limited hardware support to address key performance bottlenecks. We compare XTM to hardware-based virtualization using both real applications and synthetic microbenchmarks. We show that despite being software-based, XTM and its enhancements are competitive with hardware-based alternatives. Overall, we demonstrate that XTM provides a complete, flexible, and low-cost mechanism for practical TM virtualization.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures

General Terms Design, Performance

Keywords Chip Multi-processor, OS Support, Transactional Memory, Virtualization

1. Introduction

As multi-core chips become ubiquitous, it is critical to provide architectural support for practical parallel programming. Transactional Memory (TM) simplifies concurrency management by supporting parallel tasks (transactions) that appear to execute atomically and in isolation [10]. By virtue of speculation, TM allows programmers to achieve good parallel performance using easy-to-write, coarse-grain transactions. Transactions also address other challenges of lock-based code such as deadlock avoidance and failure isolation.

There are several proposed architectures that implement TM using hardware caches for data versioning and coherence protocols for conflict detection [18, 6, 2, 17]. Nevertheless, for TM to become useful to programmers and achieve widespread acceptance, it is important that transactions are not limited to the physical resources of any specific hardware implementation. TM systems should guarantee correct execution even when transactions exceed scheduling quanta, overflow the capacity of hardware caches and physical memory, or include more independent nesting levels than what the hardware supports. In other words, TM systems should transparently virtualize *time*, *space*, and *nesting depth*. While recent application studies have shown that the majority of transactions will be short-lived and will execute quickly with reasonable hardware resources [2, 3], the infrequent long-lived transactions with large data sets must also be handled *correctly* and *transparently*.

Existing TM proposals are incomplete with respect to virtualization. None of them supports nesting depth virtualization, and most do not allow context switches or paging within a transaction (TCC [6], LTM [2], LogTM [17]). UTM [2] and VTM [19] provide time and space virtualization but require complex hardware and firmware to manage overflow data structures in memory and to facilitate safe sharing among multiple processors. Since long-lived transactions are not expected to be the common case [3], such a complex and inflexible approach is not optimal.

This paper presents the *first comprehensive study of TM virtualization* that covers all three virtualization aspects: time, space, and nesting depth. We propose *eXtended Transactional Memory (XTM)*, a software-based system that builds upon virtual memory to provide *complete TM virtualization without complex hardware*. When a transaction exceeds hardware resources, XTM evicts data to virtual memory at the granularity of pages. XTM uses private copies of overflowed pages to buffer memory updates until the transaction commits and snapshots to detect interference between transactions. On interrupts, XTM first attempts to abort a young transaction, swapping out transactional state only when unavoidable. We demonstrate that XTM allows transactions to survive cache overflows, virtual memory paging, context switches, thread migration, and extended nesting depths. XTM can be implemented on top of *any* of the hardware transactional memory architectures. The combination is a *hybrid TM system* that provides the performance advantages of a hardware implementation without resource limitations.

XTM supports transactional execution at page granularity in the same manner that page-based DSM systems provide cache coherence at page granularity [25, 21]. Unlike page-based DSM, XTM is a backup mechanism utilized only in the uncommon case that hardware resources are exhausted. Hence, the overheads of software-based virtualization can be tolerated without a performance impact on the common case behavior. Compared to hardware-based virtualization, XTM provides flexibility of implementation and lower cost.

In the base design, XTM executes a transaction either fully in hardware (no virtualization) or fully in software through page-based virtual memory[1]. Conflicts for overflowed transactions are tracked at page granularity. If virtualization is frequently invoked, these characteristics can lead to large overheads for virtualized transactions. To reduce the performance impact, we also present two enhancements to the base XTM system. *XTM-g* allows an overflowed transaction to store data both in hardware caches and in virtual memory in order to reduce the overhead of creating private page copies. *XTM-e* allows conflict detection at cache line granularity, even for overflowed data in virtual memory, in order to reduce the frequency of rollbacks due to false sharing. XTM-g and XTM-e require limited hardware support, which is significantly simpler than the support necessary for hardware-based virtualization [17, 2]. XTM-g and XTM-e perform similar to hardware-based schemes like VTM, even for the most demanding applications.

Overall, this paper describes and analyzes the major tradeoffs in virtualization for transactional memory. Its major contributions are:

- We propose XTM, a software-based system that is the first to virtualize time, space, and nesting depth for transactional memory. XTM builds upon virtual memory and provides transactional execution at page granularity.
- We develop two enhancements to XTM that reduce the overheads of page-based virtualization: XTM-g that allows gradual overflow of data to virtual memory and XTM-e that supports conflict detection at cache line granularity.
- We provide the first quantitative evaluation of TM virtualization schemes for a wide range of application scenarios. We demonstrate that XTM and its enhancements can match the performance of hardware virtualization schemes like VTM [19] or TM systems that use serialization to handle resource limitation [6].

Overall, this work establishes that a software, page-based approach provides an attractive solution for transparent TM virtualization.

The rest of this paper is organized as follows. Section 2 reviews the requirements and design space for TM virtualization. Section 3 summarizes existing hardware-based approaches. Section 4 describes the base XTM design, while Section 5 presents the two enhanced XTM systems. Sections 6 and 7 present qualitative and quantitative comparisons between XTM and hardware virtualization schemes. Finally, Section 8 presents related work, and Section 9 concludes the paper.

2. Design Considerations for TM Virtualization

While the various TM architectures differ in the way they operate, their hardware structure is similar. They all track the transaction read-set and write-set in the private caches (L1 and potentially L2) of the processor executing the transaction [18, 6, 2, 17]. Membership in either set is indicated using additional state bits (metadata) associated with each cache line. The data for the write-set are also stored in the caches. Conflicts between concurrently executing transactions are detected during coherence actions for cache lines that belong to the write-set of one transaction and the read-set of another. More recent proposals support nested transactions that can rollback independently [14]. Tracking the read-set and write-set for nested transactions requires an additional tag per cache line to identify the nesting depth. This hardware support for transactional memory is sufficient if transactions do not exceed the capacity of caches, never exceed the supported nesting depths, are not interrupted, nor are migrated between processors. However, this case cannot be guaranteed because of multiprogrammed operating systems and the desire for software that is portable across different hardware implementations.

TM virtualization allows transactions to survive cache overflows, virtual memory paging, context switches, thread migration, and extended nesting depths. Virtualization is achieved by placing transactional state (read-sets and write-sets) in virtual memory, which provides processor-independent, location-independent, and practically infinite storage. Depending on the case, we may place some of the transactional state in virtual memory (e.g., on a cache overflow) or all of it (e.g., on a context switch).

A good virtualization scheme should satisfy the following requirements with respect to correctness and performance. First, it should be completely *transparent to the user*. Second, it should *preserve transactional atomicity and isolation* under all circumstances. Third, it should *not affect the performance of the common case* when virtualization is not needed. Finally, virtualized transactions should have *no significant effect on the performance of non-virtualized transactions* executing concurrently.

While the data for virtualized transactions are always stored in virtual memory, there are several design options to consider for the mechanisms that implement data versioning, conflict detection, and commit for virtualized transactions[2]. Table 1 summarizes the advantages of the major alternatives for each mechanism. The basic choices are between a) hardware vs. software implementation (performance vs. cost and flexibility), b) cache line vs. page granularity (storage efficiency and performance vs. complexity), and c) eager vs. lazy operations (performance vs. isolation). While it is difficult to quantitatively evaluate all reasonable combinations of the above options, this paper aims at characterizing the design space for TM virtualization sufficiently so that major conclusions can be drawn.

If performance was the only optimization metric, it is obvious that a virtualization system should be hardware-based and should handle data at cache line granularity. We discuss the proposed systems that follow this approach in Section 3. However, a virtualization system is by nature a backup mechanism, only invoked when the hardware mechanisms are no longer sufficient. Recent studies show that the majority of transactions will not exceed the hardware capabilities [2, 3]. Chung et al. [3] showed that, when transactions are used for non-blocking synchronization, 98% of them require less than 22 Kbytes for read-set and write-set buffering. About 95% of transactions include less than 5,000 instructions and are unlikely to be interrupted by external events (context switches, interrupts, paging, etc.). When transactions are used for speculative parallelization, they showed that read- and write-sets get significantly larger, but that the capacity of an L2 cache (e.g., 128 Kbytes) is rarely exceeded. The rare occurrence of transactions requiring virtualization implies that one's choices in architecting a virtualization system should better balance performance and cost. We propose such systems in Sections 4 and 5.

3. Hardware-based Virtualization

In this section, we review proposals for hardware-based TM virtualization.

UTM: The UTM system was the first to recognize the importance of virtualizing transactional memory [2]. It uses cache line

[1] When one transaction overflows, non-overflowing transactions continue executing in the hardware mode.

[2] There are similar design options for hardware support for TM. However, this paper focuses exclusively on the design tradeoffs in TM virtualization.

		Data Versioning	Conflict Detection	Commit
Implementation	**Hardware**	Low per access overhead	Overlap with other work	Low overhead
	Software	No ISA/HW changes needed	Flexibility in conflict resolution	Supports transactional I/O
Granularity	**Cache line**	Low memory/BW requirements	Less false sharing	Low overhead
	Page	Reuse paging mechanisms	No ISA/HW changes needed	Amortize overheads better
Timing	**Eager**	Fast commits	Early detection	N/A
	Lazy	Fast aborts	Guaranteed forward progress	N/A

Table 1. TM virtualization options for data versioning, conflict detection, and transaction commit. Each cell summarizes the advantage of the corresponding implementation, granularity, or timing option.

granularity, eager versioning, and conflict detection. UTM supports space and time virtualization, but does not virtualize nesting depth.

Unlike most other proposals that start with a limited hardware TM system and add virtualization support, UTM starts with a virtualized system and provides some acceleration through caching. UTM relies on the XSTATE data structure in virtual memory, which is a log with information on the read- and write-set of executing transactions. Portions of the XSTATE can be cached by each processor for faster access. The UTM implementation is rather idealized as it assumes all memory locations are appended with a pointer to XSTATE. On cache misses, a processor must always follow the pointer to detect conflicts. It also relies on the availability of global virtual addresses, which are not available in most popular architectures. Overall, UTM is space inefficient and incurs significant overheads even in some common cases (e.g., cache miss, no conflict). The same paper introduces LTM, a more practical TM system that allows transactions to overflow caches, but does not allow them to survive virtual memory paging, context switches, or migration [2].

VTM: VTM uses hardware and firmware mechanisms to provide virtualization at cache line granularity with eager conflict detection and lazy versioning [19]. It supports space and time virtualization, but does not virtualize nesting depth. For each process, VTM defines the XADT data structure in virtual memory. The XADT is organized as a hash table and contains an overflow count, the overflowed data (including metadata), and a bloom filter (called the XF) that describes which addresses have overflowed to the XADT. When a hardware cache overflows, VTM evicts a cache line into the XADT and appropriately updates the overflow count and the filter. On a context switch, VTM evicts the entire read- and write-set for the transaction to the XADT. Conflict detection and refills for evicted data occur on demand when transactions experience cache misses. However, the XADT is only searched if the overflow count is non-zero and the XF filter returns a hit. Commits or aborts for data in the XADT happen lazily: VTM atomically sets the status of transactions to committed or aborted and does the transfer to memory or XADT clean up at a later point.

VTM provides fast execution when virtualization is not needed by caching the overflow count and XF in an additional hardware cache. It also provides for fast execution when virtualizing, as it uses hardware to evict cache lines to the XADT and search the XADT for conflicts or refills. Nevertheless, the performance advantages of VTM comes at a significant complexity cost. First, the hardware must be aware of the organization of the XADT, so the XADT must be defined in the instruction set, similar to how the page table organization is defined in ISAs if hardware TLB refills are desired. Second, in order to allow overflows to the XADT without trapping into the OS for reverse translation, VTM must append each cache line with its virtual page number. For 32-byte cache lines, this implies a 10% area increase for data caches. Third, the hardware must provide coherence for the cached copies of the overflow count and the XF in each processor. Also, these cached copies must be consistent with updates to the XADT. For example, a processor incurring a miss must receive the answer to its coherence miss requests before checking the overflow counter. Otherwise, one can construct cases where a conflict is missed due to a race between an XADT eviction and a counter update. Overall, updating the counter and the XF must be done carefully and, in several cases, accesses to these structures should act like memory barriers or acquire/release instructions for relaxed consistency models.

LogTM: LogTM operates at cache line granularity and uses eager versioning and conflict detection [17]. It allows transactions to survive cache overflows, but not paging, context switches, or excessive nesting. LogTM uses a cacheable, in-memory log to record undo information that is used if a transaction aborts. Hence, an overflowed cache line is written back to main memory directly without allocation additional storage in virtual memory. An overflow bit set in the cache to allow the processor to check for conflicts when requests for this line arrive from other processors. The overflow bit may lead to some false positives in conflict detection.

4. eXtendend Transactional Memory (XTM)

The XTM system provides space, time, and nesting depth virtualization while meeting all the requirements introduced in Section 2. XTM is software-based and operates at the OS or virtual machine level. The only hardware requirement for the base XTM is that an exception is generated when a transaction overflows hardware caches or exceeds the hardware-supported nesting depth. XTM handles transaction read-sets and write-sets at page granularity. It uses lazy versioning and conflict detection.

4.1 XTM Overview

With XTM, a transaction has two execution modes: all in hardware (no virtualization) or all in software (virtualized). When the hardware caches are filled, XTM catches the overflow exception and switches to virtualized mode, where it uses private pages from virtual memory as the exclusive buffer for read- and write-set. Switching first aborts the transaction in hardware mode, which clears all transactional data from hardware caches, and then restarts it in virtualized mode. While aborting introduces re-execution overhead, it eliminates the need for an expensive hardware mechanism to transfer the physically-addressed transactional data in caches to virtual memory. XTM also marks invalid the entries in the data TLB for the processor executing the overflowed transaction. No other transactions are affected by the switch.

In the virtualized mode, XTM catches the first access to each page through a page-fault and creates on-demand copies of the original page in virtual memory. By operating on copies, the virtualized transaction is isolated from any other transactional or non-transactional code. We create two copies of the original page: the *private page* is created on first access (load or store) and the *snapshot page* is created just before the first store by copying the private page. The private page buffers data updates by the transaction until it commits successfully (lazy versioning). The snapshot is a pristine copy of the original page in memory at the time the transaction started accessing it and is used for conflict detection. If a page is

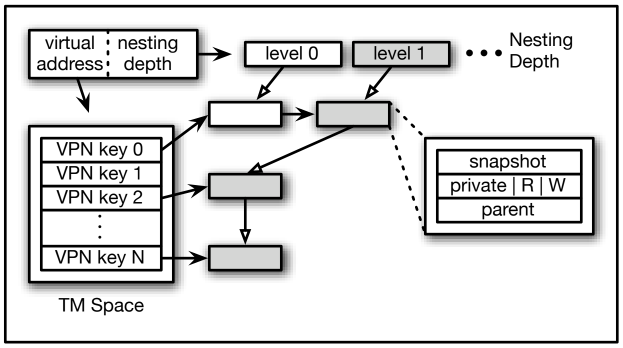

Figure 1. The Virtualization Information Table (VIT). The white box belongs to level 0, and the gray boxes belong to level 1.

never written, we avoid creating the snapshot as the private page is sufficient.

When creating the copies, XTM uses non-cached accesses for two reasons: to avoid recursive overflows and to avoid caching the metadata for overflowed transactions, which typically have low temporal locality. In fact, XTM works like a streaming program: it reads data, applies a simple operation, and writes them back in a sequential order. Most modern instruction sets support non-cached accesses by providing a bit in each page-table entry that the software can use to indicate if a page is cachable or not at this point. Once the necessary copies are created by XTM, the transaction can access data directly through loads/stores, with caching enabled, and without the need for XTM to intervene.

A virtualized transaction checks for conflicts when it is ready to commit (lazy conflict detection). Control is transferred to XTM using a commit handler [14]. XTM detects conflicts by comparing each snapshot page to the original page in virtual memory similar to backward-oriented validation developed in database literature [5]. If the contents of the two pages differ, a conflict is signaled and the virtualized transaction is rolled back by discarding all private pages. If all snapshots are validated, we commit the transaction by copying its private pages to their original locations.

XTM uses two private data structures to track transactional state. First, a per-transaction page table provides access to the private copies of pages in the read-set or write-set of the transaction. Assuming a hierarchical organization, this page-table is not fully populated. Instead, it is allocated on demand and consists only of the translation entries necessary for TM virtualization. For every virtual page, the page table points to the physical location of the private copy. The second structure is the *Virtualization Information Table (VIT)*, which is shown in Figure 1. The VIT is organized as a hash table and contains one entry per page accessed at each nesting level. An entry holds pointers to the private and snapshot pages, metadata indicating if it belongs in the read-set or write-set, and the pointers necessary for the hash table and to link related entries (see Section 4.2). The VIT is queried using a virtual address and a nesting depth. It can also be traversed to identify all private copies for a transaction at a specific depth.

4.2 XTM Space and Depth Virtualization

Figure 2 presents an example of space and depth virtualization using XTM. After an overflowing transaction is aborted, XTM allocates a per-transaction page-table and a VIT, both initially empty (❶). When the transaction restarts in virtualization mode and attempts to read its first page, XTM creates a private page and a VIT entry. The newly allocated private page is pointed to by both the VIT and the page table, and the R bit is also set in the VIT entry (❷). On the first transactional write to the page, a snapshot page is created, the VIT entry is updated, and the W bit is set (❸). If the first transactional access had been a write instead of a read, XTM would have executed steps (❷) and (❸) together.

When a nested transaction begins in virtualized mode, we need to independently track its read- and write-set. Hence, we allocate a new per-transaction page table independent from that of its parent transaction (❹). With the new table, XTM catches the first read/write to a page by the nested transaction without walking the parent's table to change access permissions. The new page table is also only partially populated. On the other hand, we do not allocate a new VIT. Nested reads (❺) and nested writes (❻) are handled like those of the parent transaction. The first nested read creates a new VIT entry that points to the parent's private page. If this is the first time this page is accessed at any depth, we create a new private page. On the first nested write, a new snapshot of the private page is created, and the modification goes to the private page. If multiple transactions in the nest access the same page, we have multiple linked VIT entries and multiple snapshots, but the private page is shared among the parent and all its nested children.

When the nested transaction needs to commit, it validates its read-set. The read-set is all snapshot pages and all read-only private pages. Validation involves comparing all pages in the read-set to the current state of the pages in memory. If validation is successful (no differences), the transaction commits by merging its transactional metadata with that of its parent. Finally, the per-transaction page table for the nested transaction is discarded. If validation fails, the transaction is rolled back by discarding all VIT entries at that level and its page table. Modified private pages are rolled back using snapshots. When the outermost transaction (nesting depth 0) commits, we copy all private pages to the original locations and merge the private page table's metadata bits into the master page table.

To make the outermost commit atomic, a transaction must gain exclusive access of all its virtualized pages. There are multiple ways to support such functionality. One way is to serialize commit while still allowing concurrent execution. In TCC [6], commit serialization is achieved by acquiring the commit token. For other systems, one can use TLB shootdown to give overflowed transactions exclusive access to validation pages. At the arrival of a shootdown message, non-XTM transactions will be rolled back only if they have a conflict with the committing transaction. An exception handler is immediately invoked and executed as an open-nested transaction [14]. The handler searches the cache for lines already accessed that belong to the evicted page. Alternatively, the TLB can maintain additional R/W bits in each entry that allow the TLB shootdown handler to quickly check for conflicts. The R/W bits are not part of the PTE entry. One can devise an adaptive protocol that selects between the two options for atomic commit, if both are available. The protocol will decide based on the number of pages committed and the expected impact of serialization or TLB shootdowns. The writes used to copy private pages into original locations must be snooped by hardware to check conflicts for pending hardware-mode transactions.

4.3 XTM Time Virtualization

XTM also handles events that require process swapping (e.g., context switches or process migration). Once a transaction is in virtualization mode, all its state is in virtual memory and can be paged in and out on demand.

Other events, like I/O interrupts, require interrupt handling, before resuming user code. Existing TM virtualization schemes [2, 19] propose swapping out transactional state on such interrupts. Since most transactions are short [3], XTM uses an alternate approach, shown in Figure 3, that avoids swapping overhead in most cases. First, XTM waits for one of the processors to finish its cur-

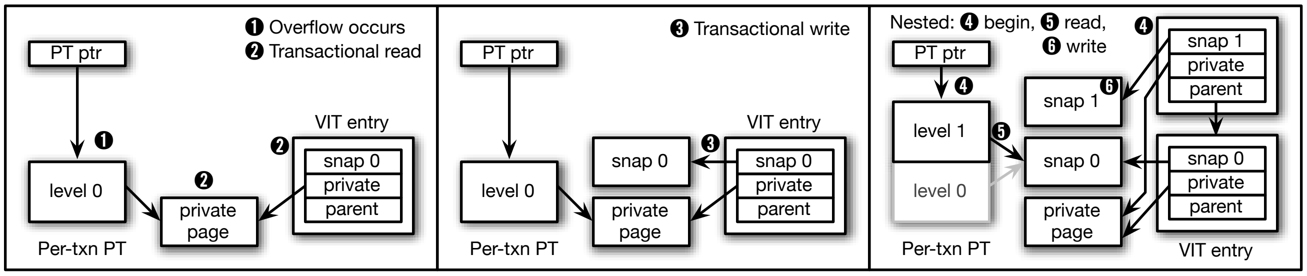

Figure 2. Example of space and nesting depth virtualization with XTM. ❶ When an overflow first occurs, a per-transaction page table (PT) and a VIT are allocated. ❷ On the first transactional read, a private page is allocated, and ❸ a snapshot is created on the first transactional write. ❹ When a nested transaction begins, a new PT and VIT entry are created. ❺ A nested read uses the same private page, and a ❻ nested write creates a new snapshot for rolling back the nested transaction.

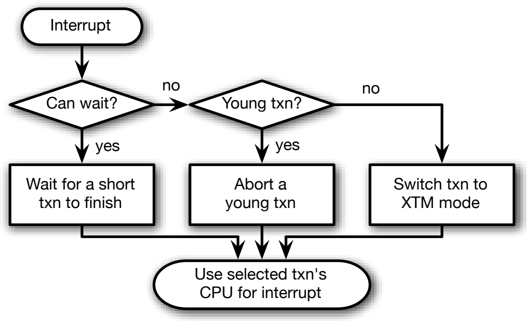

Figure 3. Interrupt handling in XTM.

rent transaction and then assign it to interrupt processing. Since most transactions are short, this will probably happen quickly. If the interrupt is real-time or becomes critical, we abort the youngest transaction and use its processor for interrupt handling. When we restart the transaction, we use the hardware mode. If a transaction is restarted many times due to interrupts, we restart it virtualized so further interrupts will not cause aborts. The latter case only happens if all transactions in the system are long, which is very rare [3].

To implement this process on an interrupt, the interrupt controller launches the OS scheduler to a statically selected core or one selected dynamically based on its current status. The scheduler runs in its own transactional context using an open nested transaction so that the transaction executing on this processor is not affected. The scheduler consults XTM to select a core to run the actual interrupt handler based on the process in Figure 3.

4.4 Discussion

XTM can be implemented either in the OS as part of the virtual memory manager or between underlying TM systems and the OS, like virtual machines [27]. Its only hardware requirements are exceptions for virtualization events (e.g., cache overflow, exceeding nesting depth, etc.). XTM implements per-transaction page tables, which can be cheaply realized by copying and modifying only the portion of the master page table containing the addresses accessed by the overflowed transaction. For example, XTM in x86 starts with a 4KB empty page directory and augments it with second-level 4KB page tables as needed. Since XTM is software-only, all its policies and algorithms can be tuned or changed without affecting the processor ISA.

Long virtualized transactions are more likely to abort due to interference. Fairness between overflowed transactions and un-overflowed ones depends on the validation scheme. The validation scheme based on TLB shootdown provides a chance for software to control fairness at conflict detection. Either the un-overflowed or the overflowed transactions can be selected to rollback with a prioritization policy such as aging. In the scheme with the TCC token, the token arbitrator can assign a high priority to the overflowed transactions to prevent starvation [6].

To reduce the overheads, XTM can allocate a number of virtual memory pages for each processor at the beginning of the program. These pages can be used by XTM on demand for private copies and snapshots, avoiding the overhead of allocation unless the number of pre-allocated pages proves insufficient.

5. Enhanced XTM Schemes

We now introduce XTM-g and XTM-e that use a limited set of hardware features to reduce the overhead of virtualization.

5.1 XTM-g

On a cache overflow in the base XTM design, we abort the transaction and switch to virtualized mode; this incurs large penalties. In XTM-g, a transaction overflows to virtual memory *gradually*, without an abort. Hardware handles the data accessed by the transaction up to now, and any new data are tracked in the virtual memory system. XTM-g is especially beneficial when the hardware caches overflow frequently by a small amount of data.

When virtualized, a transaction buffers state in both hardware and virtual memory. To distinguish the two, we introduce the overflow or *OV* bit to page table entries[3], TLB entries, and cache lines. If OV is set, the corresponding data have been virtualized. Upon eviction, data that do not belong in the read-set or write-set of any transaction are evicted as usual. If a line accessed by a transaction is evicted, XTM-g copies the line to a private page and marks its OV bit in the page table. It is possible for virtualized lines to re-enter the hardware caches, in which case the OV bit is set in the cache. Lines with the OV bit set can simply be evicted from the cache, since they are already virtualized.

Figure 4 illustrates XTM-g. In this example, the hardware cache has three lines written by a transaction, where two of them belong to the same page. When one of the two lines is evicted because of an overflow, an exception is raised and the XTM-g software starts. It first uses reverse translation to find the virtual address for the overflowed line. Then it creates the private page and snapshot, updates the VIT, and writes the evicted data to the private page. Before returning, XTM-g queries the cache about other lines from the same virtual page and finds the other line. It is also evicted to the private page and their metadata are placed in the VIT. Finally, the

[3] Several architectures, such as x86, provide a few bits per page table entry for future uses.

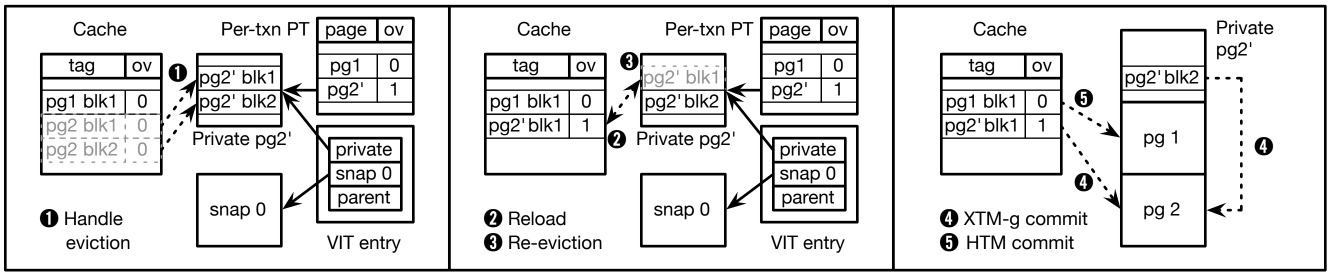

Figure 4. Example of space virtualization with XTM-g. The transaction starts with three modified cache lines. ❶ When a line is evicted because of overflows, the transaction is not aborted. A private page and a snapshot are made (since the line is modified), the OV bit is set in the PT, and the line is copied. ❷ When the evicted line is reloaded in the cache, the line's OV bit is set. ❸ The line is re-evicted to the private page without an eviction exception. ❹ XTM-g commits first, then ❺ the hardware TM (HTM) commits.

OV bit of the page is set in the page table so cache lines that re-enter the cache will have their OV bit set. By the end of this process, the transaction has one page in virtual memory while the rest is still in the hardware cache (❶). If other overflows occur, more pages can be moved to virtual memory as needed. Once evicted, the cache lines are reloaded with the private page address and their OV bit set in the cache (❷). When they are re-evicted, cache eviction logic checks their OV bits. Since the bits are set, they are allowed to be evicted to the private page without an eviction exception (❸). To properly commit such a transaction, the hardware TM system must support a two-phase commit protocol [14]. Once validation of hardware-tracked data is complete, control is transferred to XTM-g to validate the pages in virtual memory. If that validation passes, we first commit the data in virtual memory (❹) and then return to the hardware system to commit the cache data (❺). Essentially, XTM-g runs as a commit handler [14].

5.2 XTM-e

XTM and XTM-g operate at page granularity and are prone to aborts due to false sharing: a transaction may abort because it read a word in the same page as a word committed by another transaction. XTM-e allows cache line-level tracking of read- and write-sets, even for overflowed data. First, each VIT entry is made longer to have one set of R and W bits per cache line in the page. XTM-e evicts cache lines to virtual memory similar to XTM-g, but also sets the fine-grain R/W bits in the VIT using the metadata found in the cache. Second, XTM-e must handle the case where a cache line in a virtualized page is accessed later.

A naïve solution would be to switch to software on every access to a page with the OV bit set in the TLB in order to update the VIT entry. To avoid this unnecessary overhead, XTM-e uses an *eviction log buffer (ELB)*. The ELB is a small hardware cache, addressed by cache line address, that stores a tag and the R and W bits for a cache line. When a line from an OV page is accessed, we note the R or W metadata in the ELB without interrupting the user software. When the ELB overflows due to associativity or capacity issues, we merge the metadata in all valid ELB entries into the proper VIT entries. In other words, the ELB allows us to amortize the cost of an exception over multiple cache lines. If the ELB does not fill until the transaction completes, we transfer metadata from the ELB to the VIT before the validation step. During validation, XTM-e uses the fine-grain R/W bits available to determine which cache lines within a snapshot or private page should be compared with the original page in memory to check for conflicts. Overall, XTM-e improves on XTM-g by eliminating most of the false sharing, which is common if pages are sparsely accessed by transactions.

6. Qualitative Comparison

Here, we provide a qualitative comparison between the software-based XTM system and the hardware-based VTM system [19]. Table 2 summarizes the key differences. Note that VTM does not provide virtualization of nesting depth.

Hardware Cost and Memory Usage: The only HW requirement for XTM is an exception when virtualization is needed. XTM-g requires OV bits in page tables and caches, while XTM-e adds the ELB as well. On the other hand, VTM require significant hardware components and complexity: a cache for its overflow structure (XADT), hardware walkers for the XADT, and hardware to enforce coherence and consistency for the overflow counter and the filter. Unfortunately, the complexity of VTM goes beyond microarchitecture. The XADT organization and any software visible issues about the way it is cached (e.g., consistency) must become part of the ISA.

On the other hand, XTM can lead to memory usage issues as it requires storage for the per-transaction page table, the VIT and the private/snapshot copies. Even though the page tables are not fully populated, the XTM space requirements will be higher than that for VTM, particularly if transactions overflow hardware caches by a few cache lines. VTM uses memory space only for the XADT, which is a hash table for evicted cache lines.

Implementation Flexibility: XTM is implemented purely in software. XTM-g and XTM-e have small and simple hardware requirements. Since most of these three systems is in software, there is significant flexibility in tuning their policies and integrating them with the operating system. On the other hand, VTM requires both hardware and firmware, which means that there little flexibility in data structure organization, underlying coherence protocols for the XADT caching, etc. Nevertheless, the hardware implementation of VTM allows for better performance isolation between virtualized transactions and non-virtualized transactions. With XTM, making the software commit atomic can cause stalls to other transactions from the same process. Nevertheless, processors executing transactions from other processes are never affected, which is particularly important.

Performance: So far we have argued that XTM and its enhancements provide lower hardware cost and better flexibility than VTM. Hence, the question becomes how they compare in performance. If the software-based XTMs can also provide competitive performance, then they have an edge over the hardware-based VTM.

The base XTM system can introduce significant overheads. When XTM virtualizes a transaction, it starts with an abort. The necessary re-execution can be expensive for long transactions. Of course, these overheads are important if virtualization events are often. If this is the case, XTM-g (eliminates aborts for switch)

	XTM	XTM-g	XTM-e	VTM
Virtualization	space, time, nesting depth			space, time
HW Cost	OV exception	OV exception, OV bit	OV exception, OV bit, ELB	XADT walker, XADT cache, virtual tags in caches
SW Cost	VIT, page tables, extra copies per accessed page			XADT, XSW, XF, overflow count
Switch Overhead	transaction abort	OS handler		HW handler
Other Overheads	page copying, page comparisons		page copying, cache line comparisons	accessing XADT/XF, XF/count consistency
Sensitivity	page occupancy, false-sharing		page occupancy	XF miss ratio
Flexibility	pure SW	mostly SW		mostly HW

Table 2. A qualitative comparison between XTM, XTM-g, XTM-e, and VTM.

and XTM-e (reduces aborts due to false sharing) will be necessary for the XTM approach to be competitive. XTM also benefits from applications that access most of the data in each page they touch, as this makes page copying operations for versioning or commit less wasteful.

On the other hand, VTM's overhead comes mostly from accessing the XADT when the XF filter misses. Hence, the case when VTM can be slow is when many transactions overflow and searching, inserting, and deleting in a large XADT becomes too slow. Note that manipulating the VIT is faster, as each entry is for a whole page, not a cache line. Furthermore, the VIT is private to each transaction, while the XADT is shared within a process; hence, some synchronization is needed for the XADT across processors. In all other cases, VTM provides fast virtualization as it avoids switching to OS handlers and operates on data at fine granularity. Again, this is particularly important if virtualization events are often.

For time virtualization, XTM has a better process that avoids swapping transactions in many cases. On the other hand, VTM always swaps out transactional state, even to run a short interrupt handler. Hence, VTM can be inefficient for handling frequent interrupts.

7. Quantitative Comparison

We compared XTM to VTM using an execution-driven simulator. To our knowledge, this is the first quantitative analysis of TM virtualization. Table 3 shows the simulation parameters for our experiments. The simulator models a CMP with 16 single-issue PowerPC processors. For an underlying hardware TM system, we used TCC because it allows transactions to be used for both non-blocking synchronization and thread-level speculation [6]. The latter case leads to larger transactions likely to stress hardware resources. Each processor can track transactional state in its 32KB, 4-way associative L1 data cache (32B lines) [3]. A 16-entry victim cache is used as a simple mechanism to eliminate most of the conflict misses [15]. The simulator captures the memory hierarchy timing, including all contention and queuing.

As mentioned in section 4.4, XTM can be implemented in the virtual memory module of an OS or a virtual machine. We implemented XTM as privileged software module invoked in our simulated system by event handlers as described in [14]. For example, commit handlers invoke XTM if the transaction is virtualized. XTM is written in C and PowerPC assembly and is approximately 3,000 lines of code. It takes about 1,400 dynamic instructions to overflow a page gradually for XTM-g/e. The base XTM restarts an overflowed transaction instead of overflowing pages gradually. For XTM and XTM-g, it takes about 2,300 instructions per validation and 1,400 instructions per commit per page. Instruction counts for XTM-e vary by page occupancy. Dynamic instruction counts can be reduced using unrolling and/or SIMD instructions for faster page copying (Altivec [4], SSE [26], etc.).

Feature	Description
CPU	16 PowerPC cores, 200 cycle exception handling overhead
Cache	Private, 4-way, 32KB, with 32B lines
Victim Cache	16 entries
Main memory	4KB page, 100 cycle transfer latency
Bus	16B wide, 3 cycle arbitration, 3 cycle transfer latency

Table 3. Parameters for the simulated CMP.

We also implemented VTM as a hardware extension to the simulator (XADT walker, coherence for XF and overflow count, etc.). Our experiments focus on virtualization events when a single application runs on the CMP—we did not conduct multiprogramming experiments.

We used three parallel benchmarks from SPLASH2 [28] (radix, volrend, and water-spatial) and one from SPLASH [23] (mp3d), which used transactions to replace locks for non-blocking synchronization and use mostly small transansactions. We also used two SPEC [24] benchmarks (equake and tomcatv), which use transactions for speculative parallelization at the outermost loop which results in many long transactions. To explore other interesting patterns not generated by the six applications, we also designed a microbenchmark to produce randomized accesses with a desired average transaction length, size of read-/write-sets, and nesting depth.

7.1 Space Virtualization

Figure 5 presents a performance comparison between XTM and VTM for space virtualization (i.e., overflow of hardware caches). We have omitted mp3d, water-spatial, and equake because they never overflow with a 32KB cache and thus experience no virtualization overheads with the studied schemes. It is expected that short transactions that run successfully without virtualization will be the common case [3]. The microbenchmark was configured with three average read-/write-set sizes, where -Pn means accessing a uniformly random number of pages between 1 and n, inclusive.

Figure 5 shows the breakdown of the execution time for each virtualization scheme relative to VTM. The overhead is broken down into time spent for data versioning, committing, and validation (conflict detection). The cycles not used by the schemes are categorized into three items: useful time for committed work, violation time for the cycles lost due to violations, and idle time. For radix and micro-P10, the base XTM works well and introduces overhead of less than 6%. For the rest of the programs, XTM introduces significant overhead due to the transaction aborts when switching to virtualized mode and due to the time necessary to make the private and snapshot copies. However, XTM-g and XTM-e reduce the overhead of XTM significantly and make it comparable to that of VTM (less than 0.5% for several cases). Note that the results from the microbenchmark show the idle time varying across the schemes. It is due to the barrier placed at the end

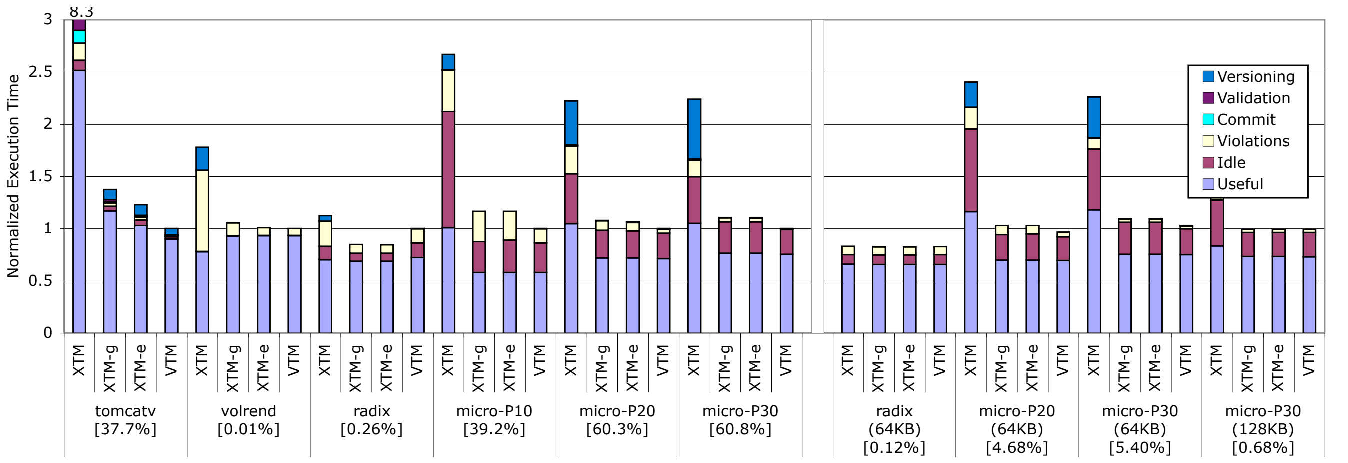

Figure 5. Execution time breakdown for space virtualization. For tomcatv, XTM's validation is 0.3× and versioning is 5.1×. The left set of bars is with a 32KB transactional state buffer, and the right set of bars shows the effect of larger buffers. Times are normalized to VTM with a 32KB buffer, and the bracketed numbers show the percentage of overflowed transactions.

of the parallel execution to synchronize the threads. Buffer overflows randomly introduced by the threads easily break load balancing around the barrier, forcing the threads with fewer overflows to wait a long time at the barrier.

The overhead breakdown for volrend and radix is shown enlarged in Figure 6. For volrend, VTM performs better, while for radix, XTM-e is the fastest. The reason is the time spent searching for overflowed data. VTM's data versioning cycles come from time spent overflowing data to the XADT and then accessing it again later. On the other hand, the XTMs' data versioning cycles come from changing the address mapping of overflowed pages to point directly to the corresponding private pages. For programs that repeatedly access overflowed data, search time is more significant than overflow time.

With tomcatv, all virtualization schemes lead to relatively large overheads. Other programs contain only a few transactions that overflow, but in tomcatv, a larger number of transactions require virtualization, with multiple transactions overflowing concurrently. VTM leads to virtualization overhead of 6%. Despite relying mostly on software, XTM-e and XTM-g perform reasonably well and lead to an overhead of 9%.

So far we have assumed that hardware can store transactional state only in the 32KB L1 data cache. However, one can also place transactional state in the L2 cache, reducing the frequency of overflow. Figure 5 shows the impact of cache space available for transactional buffering. If 64KB are available, tomcatv, volrend, and micro-P10 do not generate any overflows and all schemes lead to 0% overhead. For the remaining benchmarks, larger HW capacity means less-frequent overflows, hence the overhead of virtualization drops. At 128KB, no real application has any overflows and only micro-P30 requires 256KB before it shows the same behavior. Overall, the conclusion is that if the L2 cache is used to buffer transactional state, even applications that use TM for speculative parallelization of outer loops will rarely overflow hardware resources. Hence, the small overhead increase due to a software virtualization system like XTM is no longer significant.

7.2 Memory Usage

Table 4 measures the memory requirements of the virtualization schemes. For XTM, we measure the maximum number of VIT entries and extra pages needed for copies per transaction. For VTM, we count the maximum number of XADT entries per transaction, and we compare the number of VIT and XADT entries, which

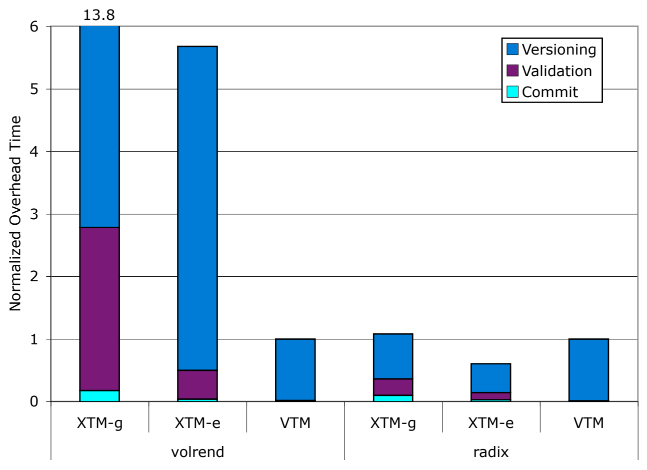

Figure 6. Overhead time breakdown for XTM-g, XTM-e, and VTM normalized to VTM.

affects the searching latency for both data structures. Since XTM has a VIT entry per page, the number of VIT entries is much smaller than the number of VTM's XADT entries. No benchmark uses more than 39 VIT entries, while some benchmarks use up to 17,000 XADT entries. The bulk of the memory overhead for XTM comes from the private and snapshot page copies. However, no benchmark uses more than 700 copies for base XTM (2.8MB) or 386 pages for XTM-e and XTM-g (1.5MB). For VTM, the XADT must store 32B cache lines and metadata. Hence, for a maximum of 17,300 entries, the XADT will occupy about 650KB. In summary, despite using page granularity, XTM does not have unreasonable memory usage requirements for any modern system.

7.3 Time Virtualization

To compare XTM to VTM with time virtualization, we simulated the arrival of I/O interrupts every 100,000 cycles. On an interrupt, we need to find a processor to execute its handler. We set the handler size to zero cycles, so all the overhead is due to switching in and out of the handler. VTM suggests that when an interrupt arrives, a transaction is swapped out to virtual memory to provide a processor for the handler. For XTM, we evaluated two policies.

Benchmark	XTM	XTM-g	XTM-e	VTM
tomcatv	21 (439)	15 (257)	15 (254)	4025
volrend	33 (316)	19 (136)	19 (139)	238
radix	21 (124)	7 (23)	8 (27)	779
micro-P10	19 (350)	9 (86)	9 (90)	1396
micro-P20	29 (495)	18 (207)	18 (202)	8039
micro-P30	39 (700)	28 (386)	28 (380)	17319

Table 4. Memory pressure. This table shows the maximum number of XADT entries for VTM and the maximum number of VIT entries for XTM. The maximum number of extra pages used by XTM is enclosed in parentheses.

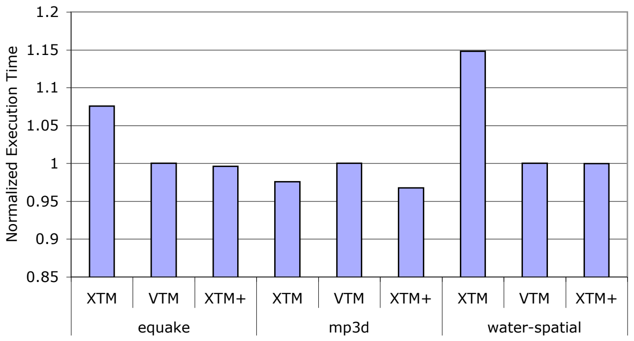

Figure 7. Execution time with time virtualization normalized to VTM. '+' stands for our time virtualization mechanism described in Section 4.3.

One is the VTM policy (abort transaction, restart later in virtualized mode). The other is the three-stage interrupt handling process explained in Section 4.3 that avoids virtualization unless necessary to guarantee forward progress.

Figure 7 shows the overhead introduced by the virtualization scheme as interrupts occur within a program. The reference is the case where no interrupts are raised. The XTM bar assumes the VTM swap-based policy. The XTM+ bar assumes the proposed approach that first attempts to abort and retry a young transaction in hardware mode. The absolute percentage of overhead is not particularly interesting as we set the handler to be empty. What is interesting is the relative comparison between the different schemes. In all cases, using XTM with the policy that favors aborts over swapping leads to lower overheads even when compared to the hardware-based VTM. The proposed approach essentially eliminates transaction swapping. Using VTM's swapping approach with the XTM system leads to the highest overheads as swapping in XTM is expensive.

7.4 Depth Virtualization

None of our programs use a nesting depth that exceeds what the hardware can support (2 to 4 nesting levels) [14]. Hence, to measure nesting virtualization overhead we used a microbenchmark that generates nesting depths that exceed the hardware support. Table 5 shows the execution time overhead as we vary the percentage of deeply nested transactions that exceed the hardware capabilities from 0.5% to 5%. This range is reasonable given a recent study that found that most programs have less than 1% nested transactions overall and the average nesting depth for that subset is 2.2 [3].

For this experiment, we measure only the base XTM as VTM does not support nesting depth virtualization and XTM-g and XTM-e behave identically to the base XTM. The overhead is measured against a simulation with hardware suport for infinite nesting levels. The execution time overhead for XTM is 2.7% and 5.9%

Nesting Freq.	Versioning	Validation	Commit	Total Overhead
5%	42.86%	2.32%	0.42%	45.60%
2%	16.61%	0.84%	0.16%	17.61%
1%	5.54%	0.27%	0.06%	5.87%
0.5%	2.60%	0.12%	0.02%	2.74%

Table 5. Nesting depth virtualization overhead.

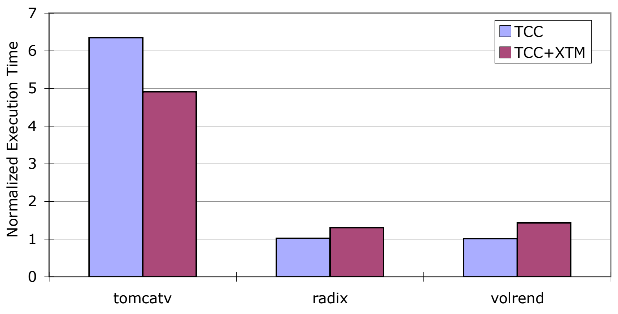

Figure 8. Normalized execution time for XTM and serialization-based virtualization. Total execution time is normalized to that with XTM-g.

when the frequency of nested transactions is 0.5% and 1% respectively. Hence, XTM can efficiently virtualize deeply nested transactions they are uncommon. On the other hand, at 2% and 5% frequency of deeply nested transactions, XTM lead to 17.6% and 45.6% overhead respectively. If deeply nested transactions become that frequency, we should probably revisit the hardware and provide direct support for additional levels of nesting. Note that in many cases, it is functionally correct to flatten nested transactions when the hardware support is exceeded. Nesting virtualization should be reserved for cases where independently aborting a nested transaction may change the program functionality (e.g., the nested transaction need not be re-executed after aborting). Open-nested transactions should always invoke virtualization if the hardware resources are exceeded.

7.5 Comparison to Serialization Schemes

Certain TM architectures use transaction serialization to provide space virtualization [6]. When a transaction overflows the contention manager or commit arbiter guarantees that it will commit. Hence, the transaction can overflow updates directly to non-transactional memory without the need for additional data structures for virtualization. Other transactions that conflict with the overflowed transaction are forced to roll back as if the overflowed transaction had already committed. One can also extend this scheme to provide time virtualization by allowing the metadata for overflowed cache lines to be stored in non-transactional memory as well. There are two basic disadvantages of such an approach. From the point of view of functionality, this scheme requires that an overflowed transaction will never attempt to roll back voluntarily (e.g., call abort). Moreover, it means that overflowed transactions do not provide fault atomicity. From the point of view of performance, this scheme limits scalability by preventing the other transactions from committing when an overflowed transaction is running. On the other hand, such as scheme allows transactional execution to happen in parallel with one commit and involves no software overheads.

To look into the performance issue, we compared XTM and XTM-g to the original TCC that uses the commit token for serialization on a cache overflow. Figure 8 shows the total execu-

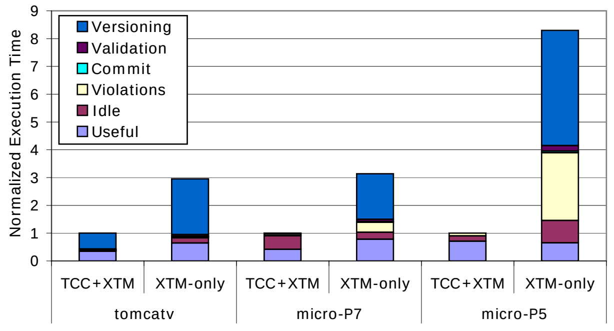

Figure 9. Execution time for TCC+XTM and XTM-only normalized to TCC+XTM.

tion time for the original TCC versus TCC+XTM, normalized to TCC+XTM-g which performs the best. The serialization scheme outperforms XTM for radix and volrend by 20% to 40%. Correlating this result with Table 4 produces an interesting observation. The big difference in the number of the pages used by XTM and XTM-g is a good indicator that XTM may incur a very high overhead for a specific application. Compared to XTM-g, XTM uses additional pages to re-execute the portion of the transaction until the first overflow. If most transactions that exceed hardware resources overflow nearly at their ends, XTM will experience significant overhead while the serialization scheme will handle them without a major slowdown. On the other hand, applications such as tomcatv have long-lived transactions that overflow early and by a lot. In this case, commit serialization is a major performance bottleneck.

Overall, Figure 8 demonstrates that serialization-based schemes do not lead to significant and consistent performance improvements that justify the functionality issues.

7.6 XTM-only Transactional Memory

The mechanisms of the base XTM can provide transactional execution semantics without any hardware support. Hence, instead of using XTM as a virtualization scheme for a hardware TM system, we can use XTM on its own as a software TM (STM) implementation. Conventional STM systems require that all code that may execute within a transaction is recompiled to insert the proper read and write barriers that allow the TM runtime to provide atomicity and isolation [22, 1, 8]. On the other hand, XTM is transparent to the user and can work with any binaries as atomicity and isolation are provided at the operating system level using page-granularity operations. Transparency is one of the main advantages of hardware TM systems over software implementations. If XTM on its own can provide sufficiently good performance for transactional execution, it can eliminate the need for a hardware TM architecture.

Figure 9 shows a performance comparison between XTM when used with a hardware TM system (TCC+XTM) and XTM used independently as an STM (XTM-only). Each of the three benchmarks represents a different scenario: no overflows with micro-P5, rare overflows with micro-P7, and frequent overflows with tomcatv. From the figure, the fewer overflows an application has, the larger the performance gap between TCC+XTM and XTM-only. Rare overflows allows the application to spend more time in hardware mode; thus, as 6.6% of the transactions overflow in micro-P7, TCC+XTM runs 3.1 times faster than XTM-only. In the extreme case of no overflows (micro-P5), TCC+XTM is 8.3 times faster. There are two factors that account for the slowdown of XTM-only. One is a higher overhead for TM suport in software only mode shown as the increment of versioning and validation cycles. The other is frequent violations from false sharing due to page-granular conflict check all the time shown as the increment of violation cycles. This result suggests that, while cost-effective, XTM may best be used as a backup mechanism from the viewpoint of balancing performance and cost. It is interesting to note that the performance gap between TCC+XTM and XTM-only is unlikely to be as large as that between hardware TM and software TM as TCC+XTM has the additional overhead of switching execution mode from TCC to XTM at overflows.

8. Related Work

Proposed software TM implementations also provide transactional semantics without hardware constraints as they are always built on top of virtual memory [22, 7, 9, 13, 1]. This paper focuses on virtualization for hardware TM systems because they provide transactional semantics with minimal overheads and make the implementation details transparent to software. XTM can also be seen as a hybrid TM system, as it supports transactions in both hardware and software modes. Unlike [11] and [16] that use user-level software and compiler support, XTM uses kernel-mode software. XTM is completely transparent to all levels of user software (application and compiler).

XTM builds upon the research on page-based, cache-coherent DSM systems [25, 21]. Unlike page-based DSM, XTM is a backup mechanism utilized only in the uncommon case when hardware resources are exhausted. XTM also draws on research that uses virtual memory to implement transactional semantics for the purpose of persistent storage [20, 12].

9. Conclusions

For transactional memory to achieve its potential as a technology that simplifies parallel programming, virtualization of transactional hardware is necessary. Transactions must be able to overflow hardware, survive interrupts and context switches, and deal with arbitrary nesting depths without compromises in functionality and transactional semantics.

This paper presented eXtended Transactional Memory (XTM) as the first mechanism to virtualize all three TM aspects: space, time, and nesting depth. XTM is a software-only approach that requires no hardware support since, in the common case, virtualization will likely be invoked infrequently. XTM operates at the operating system level and handles transactional state at the granularity of pages. We also presented two enhancements to the base XTM, XTM-g and XTM-e, that use limited hardware support to address basic performance overheads if overflows become more frequent. Finally, we provided the first quantitative evaluation of TM virtualization schemes, which included all three XTM schemes and a hardware-based alternative (VTM). We found that, despite being software based, the XTM designs provide similar performance to VTM with the cache sizes easily affordable in modern multi-core chips (e.g., 32KB). Even for demanding applications, emulated by microbenchmarks in our experiments, XTM-g and XTM-e show competitive or better performance than VTM. We also compared XTM to schemes that use serialization to provide virtualization and studied its performance without a hardware TM substrate. Overall, XTM provides a fully-featured and flexible solution for the virtualization of TM hardware at a low complexity cost. Using XTM, software developers can develop simpler parallel code that runs on TM architectures and is not limited by any implementation-specific constraints.

Acknowledgements

This research was sponsored by the Department of Energy under grant number DE-FG02-03ER25564. It was also funded by the Defense Advanced Research Projects Agency (DARPA) under grant number NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of Energy, the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Additional support was also available through an NSF Career Award (CCF-0546060) and through the MARCO Focus Center for Circuit & System Solutions (C2S2) under contract 2003-CT-888.

References

[1] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[2] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005.

[3] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, February 2006.

[4] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, March 2000.

[5] T. Haerder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984.

[6] 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 International Symposium on Computer Architecture*, pages 102–113, June 2004.

[7] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[8] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[9] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[11] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[12] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th symposium on Operating systems principles*, Saint Malo, France, October 1997.

[13] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *19th International Symposium on Distributed Computing*, September 2005.

[14] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006.

[15] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74, St. Louis, MD, USA, September 2005. IEEE Computer Society.

[16] M. Moir. Hybrid transactional memory. Unpublished manuscript, July 2005.

[17] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *12th International Conference on High-Performance Computer Architecture*, February 2006.

[18] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 5–17, New York, NY, USA, October 2002. ACM Press.

[19] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Madison, WI, USA, June 2005. IEEE Computer Society.

[20] 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.

[21] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1996.

[22] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Canada, August 1995.

[23] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*.

[24] Standard Performance Evaluation Corporation, *SPEC CPU Benchmarks*. `http://www.specbench.org/`, 1995–2000.

[25] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th Symposium on Operating Systems Principles*, Saint Malo, France, 1997.

[26] S. T. Thakkar and T. Huff. The internet streaming SIMD extensions. *Intel Technology Journal*, (Q2):8, 1999.

[27] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.