Challenges to Providing Performance Isolation in Transactional Memories

Craig Zilles and David H. Flint
Dept. of Computer Science, University of Illinois at Urbana- Champaign
201 N. Goodwin Ave. Urbana, IL 61801
[zilles, dflint]@cs.uiuc.edu

Abstract

Due to the inevitability of chip multiprocessors and the difficulty of parallel software development, there has been widespread interest in techniques that facilitate parallel programming. Recently, there have been a number of proposals regarding hardware support for transactional programming models. A key advantage of transactional programming models over lock-based synchronization is that two critical sections may be executed in parallel if they do not conflict, without the programmer needing to implement fine-grain locking. In effect, transaction hardware provides an implicit system-wide lock that is speculatively elided.

In this paper, we identify three factors that allow one process to impact the performance of other concurrently running, independent processes to a degree not present in traditional (non-transactional) multiprocessors: (1) forward progress is defined in terms of user-specified transactions (rather than instructions), (2) inefficiencies in verifying the independence of critical sections with large data footprints, and (3) subsumption of kernel execution in user-mode transactions. We describe and discuss specific problematic scenarios in the context of existing transactional hardware proposals. We believe these scenarios should be considered in architecting a robust transactional-based system.

1. Introduction

With power becoming the primary implementation constraint in microprocessors, there has been a shift in focus in commercial processors from single-thread performance (whose power scales exponentially with performance) to multi-core designs (whose power and performance scales linearly with the number cores). The benefits of multi-core designs, however, rely on the presence of thread-level parallelism (TLP) in the workload. This TLP is not to be taken for granted, as parallel software development has all of the difficulties of sequential software development and the challenges of correctly synchronizing among threads.

The traditional approach to synchronization involves identifying critical sections — regions of code that access potentially shared data in which race conditions may produce undesirable results — and ensuring race-free execution by using a synchronization variable (*e.g.*, a lock) to provide mutually exclusive access to the data. Implementing synchronization presents a number of challenges: 1) correctness requires that all accesses to a given data item must be protected by the same synchronization variable, 2) ensuring sufficient concurrency may require fine-grain locking, which requires more programmer effort, and 3) avoiding deadlock requires that when multiple synchronization variables must be acquired, all critical sections must acquire them in the same order.

In light of the challenges of parallel programming, computer architects have sought techniques that help alleviate the burden of parallelization that they have thrust upon the programmer. In particular, there have been a number of proposals regarding hardware support for transactional memory programming models [1, 4, 6, 11].

With transactional memory, transactions can be used to atomically execute a critical section. Transactions are arbitrary code sequences delimited by markers that begin and end the transaction. When a transaction completes, all loads and stores within the transaction will appear to have occurred at the time of the commit. To achieve concurrency in the presence of potentially conflicting transactions, transactions must be executed speculatively. When no conflict occurs, the transaction can be committed, writing speculatively buffered stores to architected memory. If a conflict is detected, one or more transactions will need to be aborted. On an abort, the state of the affected process is rolled back to the state at the beginning of the transaction, with no stores having been committed. At this point the transaction can be restarted, perhaps with provisions to ensure forward progress.

The key advantage of transactional memory programming models over traditional lock-based synchronization is that they obviate the need to assign synchronization variables. Although programmers must still identify the bound-

aries of critical sections, transactions shift the burden of ensuring mutual exclusion from the programmer to hardware. Furthermore, multiple non-interfering transactions may be executed concurrently, without the programmer needing to implement fine-grain locking.

Although there are a number of intriguing advantages to machines that support transactional programming models, these machines are very different from existing systems and a number of questions remain as to how they should be architected. In this paper, we consider how a transactional programming model impacts the property of *performance isolation*, the ability of a machine to prevent the behavior of one thread/process from impacting the performance of other threads/processes [12]. In Section 2, we provide background on the concept of performance isolation and how it can be achieved in traditional (non-transactional) machines.

Much of the difficulty in achieving performance isolation in transaction memory machines revolves around support for "unbounded" transactions. Although it is generally anticipated that most transactions will be small, recent proposals explore mechanisms to support "unbounded" transactions — those whose run time is not limited and whose memory footprint is bounded only by physical or virtual memory. On the face of it, unbounded transactions seem compelling, as we want to architect transactions in a way that will enable portability across hardware implementations. Furthermore, if we are to specify limitations on transactions, useful limitations are likely to be in terms of a given memory footprint and/or number of dynamic instructions, two metrics that are difficult to reason about at the source code level.

Nevertheless, the implications of unbounded transactions are subtle. Transactional memory systems can differ from non-transactional systems in three important ways: 1) the granularity of forward progress is determined by the application, 2) the need to verify the independence of transactions, and 3) the potential for kernel data structures to be "held" by user code. We present these differences of transactional memory systems in considering three challenges to achieving performance isolation:

1. How a process can hog physical memory (Section 3),
2. How large, long running transactions can prevent or inhibit the forward progress of other, unrelated processes (Section 4), and
3. How user-space transactions can inhibit other processes from receiving kernel services (Section 5).

For concreteness, we consider these challenges in the context of three previously proposed implementations of transactional memory. The first two difficulties are observed in Transactional Memory Coherence and Consistency (TCC) [4] and Limited Transactional Memory (LTM) [1], which, for efficiency, extend the cache coherence mechanism to detect conflicts. In this discussion, we identify the potential that a directory-based coherence protocol has to isolate transactions from non-communicating processes. For the last challenge, we consider an Unbounded Transactional Memory (UTM) [1] machine that allows user-mode transactions to use kernel services, to explore the interaction between unbounded transactions, forward progress guarantees, and allowing kernel services in user-mode transactions. We conclude with a discussion (in Section 6) that considers whether unbounded transactions should be architected — "large, but bounded" transactions is a yet unconsidered alternative — or whether such transactions should be allowed, but with weaker forward progress guarantees.

2. Background: Performance Isolation

Performance isolation describes the degree to which a machine can prevent the behavior of one thread (or process) from impacting the performance of another thread. In the context of a multiprocessor machine, a good proxy for performance isolation is whether one thread executing on a multiprocessor with N cores is guaranteed to perform at least as well as it would running alone on a machine with 1/Nth as many resources (*i.e.*, 1 core, 1/Nth the physical memory, 1/Nth the memory bandwidth, etc.).

To achieve performance isolation, the allocation of system resources needs to be out of the hands of application processes. When the machine and/or system software can control resource allocation, attempts by a process to use a large amount of resources will cause **that process** to slow down, without impacting other processes. If performance isolation is not achieved, then a system is vulnerable to "microarchitectural denial-of-service attacks" [3].

Conventional (non-transactional) multiprocessor systems can be made to have a high degree of performance isolation. Below, we discuss a number of system resources and how conventional systems can control their allocation. Generally, these mechanisms can be implemented to have a gradual impact (*i.e.*, the performance penalty is slight when a process only slightly misbehaves).

1. **CPU time:** CPU time is assigned to processes by the scheduler in the operating system. A process can be forced to relinquish a processor by using an interrupt (timer or interprocessor).
2. **I/O bandwidth:** As access to I/O devices is generally under the control of the operating system, the OS can allocate these resources as it sees fit. If a process tries to exceed its allocation (by requesting many I/O's in a short period of time), its time slice can be shortened to keep it's I/O rate in line with that of other processes.
3. **Physical memory:** Allocation of physical memory is in control of the operating system, which can assign

each process a budget of pages. If the working set of a process significantly exceeds its page budget, the process will frequently fault and will be descheduled while an I/O is requested.

4. **Memory bandwidth:** Allocation of memory bandwidth is controlled by the hardware. If there is contention for a link or a bank, a round-robin policy can be used to ensure that each process is allocated its fair share.
5. **Shared caches:** The advent of chip multiprocessors has led to the sharing of caches between cores. These shared caches can be partitioned (either dynamically or statically) to prevent threads from significantly impacting each other's miss rates [7].

In these ways, a process can be throttled back so as to use only its share of resources. A misbehaving process can be allowed to continue to make forward progress without negatively impacting other processes.

3. Granularity of Forward Progress

The first challenge to performance isolation in systems that support transactional programming models is that the granularity of forward progress is determined by the application. Whereas in a conventional machine, an execution can be stepped forward one instruction at a time, transactions must be atomically committed.

Importantly, the transactional programming model provides no alternative to achieving forward progress. Unlike techniques like Speculative Lock Elision (SLE) [10], which achieve a transactional-memory implementation of a conventionally-synchronized critical section, there is no non-transaction implementation to fall back on. We cannot expect the programmer to produce two copies of the code (one transactional, one conventionally synchronized), as that negates all of the benefits of the transactional programming model. As a result, an application determines the granularity of steps its execution can progress in, and the system can only control when (and if) these steps will be taken.

To see how this can affect performance isolation, consider the allocation of physical memory to processes. In a traditional architecture, a process can be guaranteed to make forward progress with a physical memory page budget of only a few pages; some RISC architectures require as little as 2 pages, not counting the page table. As a result, the operating system has almost complete flexibility is deciding how many pages of physical memory should be allocated to a process.

In contrast, consider the proposed LTM implementation. Because this machine uses the cache coherence protocol to detect conflicts between transactions, the whole transaction must be resident simultaneously in physical memory. To support "unbounded" transactions, the operating system has to be prepared to allocate the whole virtual memory image of a process into physical memory (provided there is space).

In LTM machines, unbounded transactions introduce a coupling between physical and virtual memory allocations by the operating system [1]. In traditional systems, the operating system can allocate a large amount of virtual memory — necessary to make forward progress — to a process because it is a cheap resource. It can then limit the amount of physical memory — an expensive resource — it allocates to the process. With a small allocation of physical memory, a non-transactional process can make forward progress, albeit at a slower rate. In a transactional system, a minimum physical memory allocation may be required to make forward progress.

In fact, given a chance, an LTM process can require up to twice as much physical memory as it has been allocated virtual memory. Transaction stores must be buffered speculatively until the transaction commits. Until commit, the system must retain both copies of the data. For small transactions, LTM holds the speculative copy in the processor's cache and the architected version in memory. When the size of a transaction exceeds the size of the cache, LTM uses an overflow table in main memory (provided by the operating system) to hold the data that does not fit in the cache. This overflow table can grow to be as large as the application's virtual memory space.

Of course, the operating system is never obligated to allocate resources so as to let a process make forward progress; it could kill the process at any time. What is different with systems with unbounded transactions is that an application can put the operating system in a position in which it has to choose whether it is going to provide the application the necessary resources — bounded only by the virtual memory size of the application — or to refuse the process forward progress; there is no compromise.

4. The Burden of Verifying Transaction Independence

As noted in Section 1, the fundamental advantage of the transactional programming model is that it obviates the need to assign synchronization variables to critical sections. Programmers are no longer required to identify which critical sections can conflict; this becomes the responsibility of the hardware.

In relieving the programmer of the burden of identifying which critical sections could conflict, we lose the information about which critical sections definitely do not conflict. Without such information, they will all need to

1 This coupling can be avoided by allowing transaction state to be paged, as is done in UTM

be checked for conflicts, even if they belong to processes with non-overlapping virtual address spaces. If this verification becomes a bottleneck, two processes can impact each other in ways that have no analog in conventional (non-transactional) machines. Support for unbounded transactions presents a challenge to ensuring that this verification does not become a bottleneck.

In this section, we review how the proposed hardware transactional memory systems implement transactions in a cost effective manner. We then demonstrate the challenges of performance isolation in TCC and LTM in the presence of large, long running transactions.

4.1. Hardware Support for Transactions

Providing support for transactions in hardware requires two mechanisms: 1) the ability to buffer stores for commit at transaction completion time, and 2) the ability to monitor the addresses of loaded values to identify conflicts. Proposed implementations provide these mechanisms in a cost-effective way by augmenting existing structures and mechanisms. The cache can be used to hold speculative transaction state by tagging lines that have been dirtied by an uncommitted transaction. Conflicts can be detected by tracking which blocks in the cache have been read from and using the cache coherence protocol to abort a transaction when the necessary permission for a cache line involved in the current transaction is lost. In this way, transactions that fit within the cache can be supported quite efficiently. To support larger transactions, each proposed implementation falls back on a less efficient mechanism that can support arbitrarily-sized transactions.

In TCC, transaction state can only be buffered in the cache and its associated victim buffer. Cache lines that have been read or modified by a transaction "may not be flushed in TCC from the local cache hierarchy in mid-transaction." If cache conflicts or capacity necessitate flushing to make forward progress, then TCC must request commit permission in the middle of the transaction. Once commit permission is obtained the transaction can be completed non-speculatively: the existing transaction state can be committed and conflicts are prevented by holding commit permission until the transaction completes execution. Until this transaction commits, other processors are not allowed to commit their transactions.

In LTM, when a transaction's memory footprint exceeds what the cache can hold, evicted blocks are written to an *overflow hash table*: a region of uncacheable memory whose size and location are determined by the operating system. Overflow bits are added to each set of the cache; an overflow bit is set when a cache line of a transaction has to be evicted from the cache because of capacity or limited associativity. When the cache receives an intervention for a line that maps to an overflowed set, then "the overflow hash table is searched for the requested line." The time to perform this search depends on the number of conflicts in the hash table, which in turn depends on how densely populated the hash table is. At minimum, servicing such an intervention requires one round-trip memory access time. While handling an overflow, the proposed LTM machine NACKs all incoming cache interventions.

```
address ← first address to load
B ← size of cache block in bytes
last ← address + 2 × B × N {last address to load}
BEGIN TRANSACTION
while address ≤ last do
   read MEM[address]
   address ← address + B
end while
while true do
   do nothing {infinite loop}
end while
END TRANSACTION
```

Figure 1. *A transaction that overflows every set of a cache with N blocks of size B-bytes each (by reading 2N words of data) then enters an infinite loop.*

4.2. TCC and unbounded transactions

As proposed, TCC provides no performance isolation in the presence of large, long running transactions; one process can completely prevent other processes from making forward progress. Consider the code fragment shown in Figure 1. This transaction has two phases: 1) it reads enough data to overflow the transaction hardware, and 2) it enters an infinite loop so the transaction is never committed.

In the first phase of the transaction, enough data is read so that the processor's transaction support is exceeded. At this point, execution of the transaction is stalled and the processor arbitrates for commit permission, so that the transaction can be completed non-speculatively. Eventually, commit permission will be granted along with exclusive access to the memory bus. If the process never ends its transaction (the second phase of the transaction), the bus lock is never released. This would prevent all other processes from committing or even making progress past a cache miss or upgrade. As proposed, a TCC machine will allow this attack to lock up a machine indefinitely with no mechanism for it to be interrupted. In fact, a TCC machine could be deadlocked by overflowing as little as one set of the cache and the victim buffer.

Clearly, deadlock can be avoided by including a watchdog timer and aborting the partially committed transaction if it holds commit permission too long. This potentially leaves the process's memory in an inconsistent state. As a result, it is likely that the associated process will need to be

killed to prevent this inconsistent state from being propagated to persistent state.

4.3. LTM and unbounded transactions

On the LTM machine, the first phase of the transaction will similarly overflow the transactional hardware support of the processor. When such an overflow occurs, the LTM machine marks the overflowing set and spills transaction state to an uncached region of main memory. To commit a transaction, the LTM processor has to retain (for the duration of the transaction) read and write permission to all cache blocks that are loaded from and stored to[2].

To check for conflicts with a transaction, the LTM machine will need to check every incoming coherence request to see if it will require the transaction to relinquish permission to one of its blocks. As the overflow transaction state is stored in a hash table, the hardware does not know a priori how many hash table entries will need to be checked. At least one main memory access will be required for each coherence request to an overloaded set.

If all sets have overflowed, then every incoming coherence request will trigger at least one additional main memory access. Because other incoming coherence requests are being NACKed during an overflow, coherence requests are no longer overlapped; *the coherence throughput of this node drops to the inverse of main memory latency*. In a system with a snooping coherence protocol, all coherence requests are sent to every processor, bringing memory system performance to a crawl when any processor overflows many sets of its cache.

Although the authors do not identify this as an important consideration, LTM was evaluated in the context of a directory-based multiprocessor. In a directory-based machine, the directory maintains a list of potential sharers, and interventions are only sent to those processors that could have a cached copy of the line in question. A directory-based coherence scheme largely prevents a process in one address space from affecting others, because there will be few[3] cache interventions between the processes. In effect, the directory is tracking the physical address space overlap of two processors.

It is important to note that directory schemes can effectively filter coherence requests only to the degree that transaction state of individual threads can be distinguished by the directory. In the context of chip-multiprocessors—one of the major motivations for transactional programming models—it is common for lower-levels of the cache hierarchy to be shared. If a shared L2 is used to hold transaction state, then, when one processor's transaction overflows the L2 cache, all requests to the overflowed sets of the cache (both requests from the CMP's cores and external interventions for data on any core of the CMP) will be penalized. The same problem occurs if transaction state is limited to the L1 cache and a hierarchical coherence protocol that uses a snooping protocol for intra-CMP coherence is used. This problem occurs also with multithreaded processors that share an L1 data cache.

Achieving performance isolation in the presence of unbounded transactions appears to require limiting transaction state to unshared caches and using a flat directory-based coherence protocol that recognizes all of the cores independently. Only in this way can one processor retain coherence permission to more memory than it can cache (so that it will receive interventions for all data that belong to the transaction), but will not receive interventions for data it has not accessed. Hierarchical coherence schemes, as they have generally been proposed, fall short of this mark, even when they use directory protocols for intra-CMP coherence. Typically, the on-chip directory (*e.g.*, Piranha's duplicate L1 tags [2]) are sized to only maintain directory information for what the caches can hold. When a transaction overflows its cache, it would also overflow its local directory, requiring snooping the overflowed transaction state for all coherence requests arriving at this chip that map to overflowed sets.

In one of the above scenarios, when a directory can only distinguish groups of nodes, impact to the whole machine's performance can be avoided by sacrificing one group. Consider a system composed of CMPs in which the intra-CMP coherence is maintained by a snooping protocol, but a directory is used for inter-CMP coherence. If one core of a CMP overflows its cache, the coherence traffic of the threads on the CMP **and** any threads that share with them could be impacted. To isolate the overflowing thread, all other cores on the CMP could be halted (for the duration of the transaction) and the caches flushed of non-transactional data. This technique bounds the impact that one thread can have on the whole machine. Even in the context of a machine with a full directory, if the cores are multithreaded (SMT or otherwise), it may be necessary to halt the other threads running on a core with an overflowing transaction.

LTM does bound the length of time a transaction can run to the length of a time slice, but this provides little help. If a transaction has not completed at the end of its time slice, it is aborted and restarted the next time the process is scheduled. Because it takes little time to overflow a cache, a pro-

2 It should be noted that this is only one means to ensuring the appearance of atomic execution. Strictly speaking, the permissions to the read and write sets only need to be held simultaneously at commit time. If we record the address/value pairs of all loads and stores within the transaction, then permission to blocks can be lost without aborting the transaction. If blocks are lost, they must be reacquired at commit time and any read values must be re-validated and stored values re-written. The more conservative scheme proposed in LTM can be easily justified because it likely performs as well in the common case and is simpler and has lower storage requirements.

3 A few interventions are possible on a system with physically-addressed caches, because caches are generally not flushed on context switches. The number of interventions, however, is bounded by the size of the cache.

cess can spend almost all of the time it is scheduled in the overflowed state. Again, the operating system can choose not to schedule the process, but there is no means to grant the process the ability to make forward progress without impacting the system as a whole.

5. Sharing through the Kernel

One challenge in implementing a transactional programming model that supports long running transactions is ensuring forward progress. In order to commit, a long running transaction needs to retain coherence permission on all cache lines it touches. When other transactions are modifying the same data, the long running transaction is at risk of starvation. To avoid starvation, a number of transactional memory schemes have proposed recording the time a transaction is started the first time, and, when a conflict is detected, aborting the younger transaction using these timestamps [1, 9].

Such attempts to avoid starvation of long transactions can actually allow an unbounded transaction to starve other transactions. If conflicts are resolved by aborting or stalling the younger transaction, then one thread could starve another thread in the same virtual address space by performing a transaction that touches every cache block in the virtual address space. Eventually, this transaction will become the oldest, and all other threads would be prevented from executing memory operations. Such a denial-of-service is not a practical concern, because, if a malicious thread had access to the victim's address space, it could corrupt data—a much worse threat. That said, if care is not taken, such a denial-of-service could occur between two processes from different virtual address spaces, because all processes share the same kernel data structures.

The proposed Unbounded Transactional Memory (UTM) [1] supports the nesting of transactions, in which the outer transaction subsumes the inner transaction. If system calls are allowed to be performed within a transaction, then kernel critical sections could be subsumed into user-mode transactions. Although the transaction cannot directly modify the kernel data structures, it could prevent other processes from updating them for arbitrary amounts of time.

If transactions can subsume kernel execution, then a process could start a transaction, perform a number of system calls attempting to modify key kernel data structures (*e.g.*, the kernel memory allocator, the allocator of disk blocks, message buffers for an Ethernet adapter, etc.), and then never commit. When another process requires the same data structure, a conflict would be detected and the "attacking" process would likely be older, so UTM would stall or repeatedly abort the transaction of the "victim" thread, causing it to fail to make forward progress. Although modern kernels are designed to minimize accesses to system-wide data structures, these structures are not eliminated, and they will eventually be modified by a process that repeatedly performs system calls, due to the limited resources allocated to the per-processor data structures.

Although this problem could be resolved by not allowing user-mode transactions to include system calls, this does restrict the power of transactional programming. First, programmers would have to request any resources (*e.g.*, allocating virtual memory, opening/`mmap`ing files, reading from sockets) before starting the transaction; this could be difficult, especially for very long running transactions. Second, any side effects from the transaction that go beyond user memory (*e.g.*, writes to sockets, renaming of files) would have to be buffered within the application then performed in earnest after the transaction has completed; this approach presents two difficulties: 1) the kernel has provided no guarantee that these actions can be completed successfully, 2) there is no guarantee on when the actions will be completed, weakening the atomic nature of transactions.

The problem can be somewhat mitigated by resolving conflicts between transactions by priority rather than (or in addition to) age, as was suggested by Herlihy and Moss [6]. If higher priority transactions are never aborted by lower priority ones, then at least priority inversion—in which a lower priority thread starves a higher priority thread—will not occur.

6. Discussion

In this section, we question whether unbounded transactions are a reasonable goal. Clearly it is important to not limit transactions to fit in a small cache with a given associativity, but does that necessitate that unbounded transactions should be supported?

In the absence of performance isolation, systems will not guarantee forward progress for unbounded transactions. In some of the scenarios previously discussed, truly unbounded transactions with forward progress guarantees can permanently prevent other threads from making forward progress. It seems inevitable that watchdog timers would be instituted, causing a trap to the operating system when a thread was being starved, so that an abusive thread could be killed.

Note that systems will not be able to distinguish an abusive process from a well-intended one that happens to use very large transactions. Although we have illustrated the problematic scenarios in their simplest form, they could be arbitrarily obfuscated so as to be indistinguishable from a useful program.

If unbounded transactions will not be guaranteed to be executed, should they be architected at all? Is it reasonable to expect a transactional memory system to support transactions that touch hundreds of megabytes of data and run for minutes of execution time? Is supporting such transactions

worth restricting the transactional programming model in other ways?

For example, the recent VTM proposal [11] is not sensitive to the scenarios described in Section 3 and Section 5, but only because the machine does not support transactions that extend past the user space of a single process (no system calls or shared memory regions are supported).

If it is impossible to simultaneously support unbounded transactions, starvation avoidance, and transactions that communicate with the kernel and/or other processes, which should be supported? From our perspective, unbounded transactions appear to be the least useful. For the rare, very large, critical sections that programmers want to implement, traditional lock-based synchronization is an option. In contrast, starvation avoidance and system calls/interprocess communication will be hard to support if not provided by the transactional memory directly.

Alternatively, "unbounded" transactions can be supported, but without the same quality of forward-progress guarantee. Among threads of an application or set of cooperating applications, contention can be resolved through a software "contention manager" that selectively schedules a subset of the threads to ensure forward progress at the application level [5, 8]. Forward progress guarantees, however, are harder to realize when the contention occurs between two independent applications, as can occur when transactions require kernel services. For such circumstances, a forward progress *guarantee* appears unrealistic, and such transactions can be executed with "best effort."

From our standpoint, unconstrained transactions appear to be most useful early in the software development process, a time when performance isolation is largely a non-issue. During development, forward progress can be guaranteed at the expense of performance isolation to enable functional testing. In the testing process, potentially difficult transactions can be identified for refactoring by the developers prior to the program's distribution.

7. Conclusion

This paper shows that transactional memory systems present new challenges to achieving performance isolation. In particular, transactional memory systems have three novel aspects: 1) the granularity of forward progress (*i.e.*, the transaction) is controlled by the application, 2) the burden of verifying the independence of transactions, and 3) the potential for kernel critical sections to be subsumed by user transactions. We have demonstrated how each of these aspects present challenges to performance isolation in the context of previous transactional memory proposals.

Although we do not present a complete solution to these problems, two insights into the problem are provided. First, if implemented properly, directory protocols can be used to isolate the transactional state of one process from others, preventing an overflowing transaction from impacting threads that do not share with it. Second, a fundamental difficulty is encountered when trying to achieve all three of the following: unbounded transactions, forward progress guarantees, and the potential for contention between independent processes (as could occur through the kernel). This difficulty suggests that large transactions be architected with only "best effort" forward progress guarantees, with application-level support for managing contention within an application and feedback to the developer on large transactions that potentially conflict through the kernel. Regardless, performance isolation will need to be considered in architecting transactional programming models.

8. Acknowledgments

This research was supported in part by NSF CCR-0311340 REU, NSF CAREER award CCR-03-47260 and a gift from the Intel Corporation. We thank Milo Martin, Ravi Rajwar, Pierre Salverda, and the anonymous reviewers for feedback on previous drafts of this paper.

References

[1] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[3] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: Insuring microarchitectural fairness. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 409–418, Nov. 2002.

[4] 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*, pages 102–113, June 2004.

[5] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, May 2003.

[6] 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*, pages 289–300, May 1993.

[7] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2004.

[8] R. Rajwar. Personal communication, May 2005.

[9] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2000.

[10] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 294–305, July 2001.

[11] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[12] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, Oct. 1998.