Token Coherence for Transactional Memory

Jayaram Bobba Michelle Moravan Umair Saeed

University of Wisconsin, Madison

Abstract

Concurrent programming holds the key to fully utilizing the multi-core chips provided by CMPs. However, traditional concurrent programming techniques based on locking mechanisms are hard to code and error-prone. Transactional Programming is an attempt to simplify concurrent programming where transactions are the main concurrent construct. Hardware Transactional Memory systems provide hardware support for executing transactions and take help from modified versions of standard cache coherence protocols for doing so. This report presents a token coherence protocol for the transactional memory system, TTM [6]. It also presents some comparisons between standard locking based systems and transactional memory systems. It has been observed that the performance of transactional memory systems is workload-dependent. However, transactional programming appears to be much easier than programming with locks.

Keywords: Transactional Memory, Token Coherence, TTM, Coherence Protocols, Concurrent Programming

1 Introduction

Traditionally, the onus for striving for greater computer performance has been on computer architects. Continual advances in computer technology dictated by Moore's Law have allowed architects to come up with newer and more innovative ways to increase computer performance. This has led software programmers to enjoy these benefits without having to worry too much about performance issues. Until now, no matter how fast processors get, software consistently found new ways to eat up the extra speed. Increase the speed of the hardware ten times, and software usually finds ten times as much to do [1]. This has allowed most classes of applications to enjoy free and regular performance gains.

With processor speeds leveling off due to severe limitations imposed by packing more and more transistors in a given area, the traditional laid back approach of software developers will have to change. Processors are going to continue to become more and more powerful, however the current trends suggest that these improvements are not going to come from increase in processor speed. Rather, these improvements are going to result from using multiple processors in parallel. This impacts the existing software programming models in a big way. In order to benefit from this increase in performance, software would have to depart from its traditional single-thread execution flow, and would have to center around concurrent programming.

There are several limitations to concurrent programming which have led to a slow adoption of this model. The most severe of these is that concurrent programming is hard. In the traditional software programming model, the programmer has to worry about errors within a single flow of execution. The addition of multiple threads introduces data races, deadlocks and starvation issues which are not straightforward to detect, and even more difficult to debug. To overcome these new issues means the programmers have to fundamentally alter how they think about programming.

There are several existing approaches which allow the programmers to write correct concurrent programs. One of the most common of these is to use locks or barriers to protect critical sections within the program from multiple threads executing in parallel. These require a significant overhead on part of the programmer. Transactional programming is a way to reduce this overhead for the programmers, by dividing the program into several transactions which can execute concurrently. Transactional memory models that support transactions can be implemented either in hardware or software. In this report, we confine our attention to hardware approaches to transactional memory.

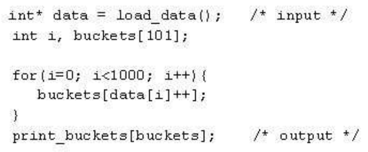
Transactional memory models typically use cache coherence protocols to detect conflicts among transactions. Conflict detection is used to maintain isolation among transactions. Atomicity is ensured by logically making all the changes visible at once to the rest of the system. In this report, we will study the performance of some of the cache coherence protocols on a transactional memory system, TTM [6]. We also come up with a new cache coherence protocol based on token coherence. In addition, we also implement some simple transactional programs and study the performance on these systems.

The rest of the report is organized as follows: In section 2, we give an overview of concurrent programming with transactions. Section 3 discusses Transactional Memory, and its various implementations. It also discusses the implementation that we chose for our project. Section 5 looks at the various coherence protocols that are used in transactional memory systems. Section 6 gives details of our Transactional Token Coherence protocol. We then discuss some performance results in section 7 and then finally conclude in section 8.

2 Concurrent Programming with Transactions

Existing parallel programming approaches require the programmer to manage concurrency directly by creating and synchronizing parallel threads [3]. This places a huge burden on the programmer. At the same time, the task is made more difficult by the fact that often conflicting goals of performance and correctness need to be balanced too. For example, a frequent method to ensure correctness in concurrent programming is the use of locks to regulate access to critical segments of the code. Using a large number of fine-grain locks increase performance, since the threads do not waste time competing. On the other hand, using many smaller locks increases the locking overhead incurred and makes the program tough to correctly implement.

Hammond et al. [4] introduce parallel programming techniques for transactional coherence and consistency (TCC) systems [3]. TCC relies on programmer defined transactions as the basic unit of parallel work, communication, memory coherence, memory consistency and error recovery. This greatly reduces the overhead associated with concurrent programming.



```
int* data = load_data();    /* input */
int i, buckets[101];

for(i=0; i<1000; i++){
   buckets[data[i]++];
}
print_buckets[buckets];     /* output */
```

Figure 1: Simple Program

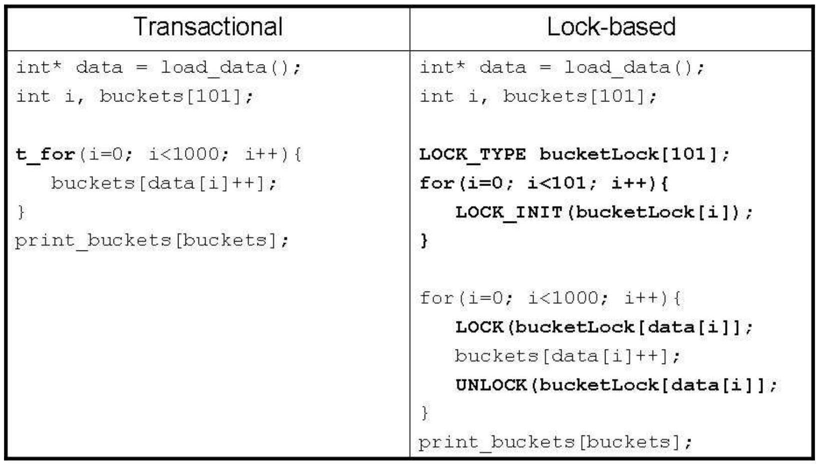
The first step in converting a program to run concurrently on transactional memory is to divide it into transactions. *If needed*, the programmer can also specify the order of each transaction. Once the program has been divided into transactions the programmer can then use the feedback obtained from the system to further tune the system to achieve best performance. This includes increasing or decreasing the size of the transactions to achieve maximum benefit.

The following example program from Hammond et al. [4] gives a clear indication of the ease of concurrent programming with transactions as compared to the traditional approach using locks (Figure 1. The example is a simple sequential code segment that calculates a histogram of 1000 integer percentages using an array of corresponding buckets.

Figure 2 shows what the concurrent version of the program looks like for both transactional programming, and using lock based approach. Changes from the sequential version are showed in bold.

In the transactional version, each iteration of the *for* loop is converted into a separate transaction. This is the only change needed for the transactional version. The underlying implementation guarantees that the sequential code will be executed correctly. On the other hand the lock based version requires the programmer to define the locks, and then acquire them before each critical section. Transactional programming thus eliminates this overhead associated with locks, and is able to overcome one of the fundamental difficulties associated with concurrent programming.

Another approach to transactional programming is to let the user explicitly mark out transactions. By extending the ISA, the user can be provided with language primitives to specify the start and end of a transaction. This approach makes the transition from lock-based programming



Transactional

```
int* data = load_data();
int i, buckets[101];

t_for(i=0; i<1000; i++){
   buckets[data[i]++];
}
print_buckets[buckets];
```

Lock-based

```
int* data = load_data();
int i, buckets[101];

LOCK_TYPE bucketLock[101];
for(i=0; i<101; i++){
   LOCK_INIT(bucketLock[i]);
}

for(i=0; i<1000; i++){
   LOCK(bucketLock[data[i]];
   buckets[data[i]++];
   UNLOCK(bucketLock[data[i]];
}
print_buckets[buckets];
```

Figure 2: Lock-based and Transactional Programs

much easier for a programmer. The critical sections need to be protected by transactions instead of locks. We follow this approach rather than the TCC approach which requires extensive modifications to the existing models.

2.1 Microbenchmarks

We implemented two microbenchmarks to understand the degree of difficulty in using transactions and to study certain performance characteristics of our transactional memory system.

Btree: The first microbenchmark we developed was a Btree implementation. We were able to implement this program completely independent of the fact that we planned to use it in a multi-threaded environment. The benchmark itself involves two C files: a header, and the body of code comprising the tree itself. To make this concurrent, we merely instrumented the code that called the tree: we surrounded all calls to insert, lookup, and delete with wrapper functions. For locking, the first half of the wrapper acquired a single lock on the entire tree, and the second released it. The actual locking mechanism (locking variable and atomic read-modify-write operation) were abstracted away by a library. The transactional implementation occurred analogously, except that a `begin_transaction` command denoted the beginning and an `end_transaction` end.

For us, then, implementing locking and transactions seem equally easy, and both are trivial compared to implementing the body of the code. The advantages of transactional programming are thus hidden: first, no external library need keep track of lock variables. Second, the usefulness of transactions lies in their performance advantage. A single lock over the entire tree defeats the purpose of multithreading. In contrast, since transactional memory limits concurrency only when truly necessary, making entire tree operations atomic need not result in serialization.

Bounded Buffer: This is a simple multithreaded program that models the standard producer-consumer problem. Half of the threads in the program produce data while the other half consume data. The think times for each of the threads were set to zero, in order to stress the transactional memory system. The implementation ran on similar lines to that of Btree.

For the programs that we implemented, we did not find that using transactional memory made programming easier compared to using coarse-grain locks. But we are convinced it made it much easier than it would have been had we been required to implement fine-grained locking to achieve tolerable performance.

3 Transactional Memory

In their seminal paper "Transactional Memory: Architectural Support for Lock-Free Data Structures" [5], Herlihy and Moss introduced the concept of transactional memory, a mechanism for easing concurrent program by providing lock-free data structures. Transactional memory lets programmers specify multiple, non-contiguous memory blocks for atomic execution: either all operations will appear to complete simultaneously, a transaction commit, or none will, a transaction abort. All processors will further view these sequences as serializable, which means that multiple transactions cannot appear to interleave.

The goal of easing concurrent programming largely motivated this new paradigm. Compared to locking, the authors found that transactional memory eliminates several problems, such as priority inversion, convoying, and deadlock related to the order of lock acquisition. Herlihy and Moss also demonstrate that transactional memory usually outperforms even the most efficient locking schemes (on several microbenchmarks). The authors attribute this to the fact that transactional memory limits sharing only when strictly necessary, while by nature programmers incur the overhead of locking even when threads actually make no attempt at concurrent access. This is especially true when programmers ensure correctness and avoid time-consuming corner-case analysis by using a coarser grain of locking than actually required. Using transactional memory, they can instead specify transactions at a coarser granularity, and so reduce reasoning about sharing patterns while still maintaining the performance of locking at a finer granularity.

Herlihy and Moss presented a hardware implementation of transactional memory; we will limit our focus to this medium as well. To provide further grounding in existing work, we will begin by comparing the two main approaches to concurrency control.

Transactional memory transactions naturally have much in common with database transactions. As in databases, a transaction in transactional memory will either commit, and atomically expose its memory accesses to other threads, or abort, in which case it exposes no operations and must usually retry. While transactions may abort for a variety of reasons [5], they perhaps most commonly do so due to avoid conflicting accesses, which break the illusions of atomicity and serializability. To instigate an abort, transactional memory implementations must provide some mechanism, typically called concurrency control, for detecting these conflicts.

Both database and transactional memory designs usually take one of two approaches: conservative (blocking) concurrency control, or optimistic concurrency control. The blocking approach requires processors to check for conflicts on every memory access. As soon as one is detected, one or both transactions attempting to access the same block must abort. In contrast, optimistic approaches tend to involve two phases. A transaction first executes in its entirety, making sure to save the previous state of altered memory blocks as as it goes. In the second phase, it performs validation to ensure that no conflict occurred. Clearly, optimistic concurrency control is optimized for the workload where conflicts occur infrequently, because when they do happen, the processor must roll back the entire transaction. In the abort case, the blocking approach seems more efficient, because the processor will not continue to execute subsequent instructions of the transaction after the first conflicting memory access occurs. Of course, the preferred case of no conflicts requires more overhead, as the processor must communicate with all other threads on every access before deciding to continue. Note that since conservative concurrency control can also result in aborts and rollbacks, it too must record previous values before making transactional changes. Herlihy and Moss chose the optimistic approach [5].

3.1 TCC

We next discuss the transactional memory implementations which most influenced our work. First, Transactional Memory Coherence and Consistency (TCC) [3] treats all memory references as part of some transaction. This differentiates it from the other approaches we present, as does its use of optimistic concurrency control. In TCC, multiple nodes can hold the same line at the same time. When a node completes a transaction, it broadcasts all of its changes at once to the other nodes. During a transaction, when a node receives such a packet, it checks these modifications against its own accesses. If it detects a conflict, it must roll back and abort. Hammond et al. implement this with an additional read and write bit on each line, which the processor sets when it makes a speculative read or write. It clears them when the corresponding transaction either commits or aborts. For correctness, the nodes must maintain this information throughout the life of a transaction. Thus, to handle the case where a new memory access might replace such a line, TCC recommends adding a victim buffer to hold its tag, read, and write bits. Without this optimization, or if the buffer becomes full, the processor must stall until it can acquire commit permission, which assures it exclusive access until it completes validation. Finally, Hammond et al. handle conflicts by having each node checkpoint its register state each time it commits. When a node in a transaction sees a broadcast that accessed one of its speculative lines, it rolls its registers back to their state prior to the beginning of its transaction.

Unlike TCC, the following two implementations both use conservative concurrency control. Ananian et al.'s Unbounded Transactional Memory (UTM) homes in on the problem of overflowing cache lines [2]. They propose a hardware implementation that lets a transaction access the entire virtual address space and take an arbitrary amount of time. The authors were motivated by the observation that, while most transactions are small, a very small percentage are quite large. Ananian et al. wished to provide a mechanism for a uniform transactional interface: programmers may use transactions without first having to calculate whether they meet time or memory limitations. However, their UTM implementation involves significant hardware additions. First, the instruction set architecture must add transaction begin and transaction end instructions. A begin instruction tells the processor to take a snapshot of its register state and register-renaming table in case of rollback. The node cannot reuse physical registers while working on a transaction. The node must expose this state, along with the address of an abort handler, to the operating system to allow that software to correctly handle context switches. UTM maintains memory state with the xstate data structure, which logs the accesses of all pending transactions. Associated with each line is a read/write bit and a log pointer, which points to an entry recording the previous value, a back-pointer to the block address, and a pointer to other transactions that have read-shared the same block. The log for each transaction also tracks that transaction's progress, which may be pending, committed, or aborted. The processors perform clean-ups on abort, by using the log to restore previous values; log entries of committed transactions may also be periodically wiped. Clearly this entire scheme requires extensive system alteration.

3.2 UTM and LTM

To mitigate the costs incurred by TCC, Ananian et al. introduce Large Transactional Memory [2]. In this proposal they limit the size of transactions to physically addressable memory, and their duration to the length of a time-slice. They do not permit transactions to migrate among processors while in progress. This reduction in scope limits hardware changes to only the processors and caches, as opposed to affecting the entire memory subsystem, as in UTM. Like UTM, though, LTM detects conflicts by maintaining read and write bits in the cache lines. It also supports overflows by storing the replaced data in a main memory hash table. This involves the addition of two more bits, a T bit to indicate whether a line belongs to a transaction, and an O bit to indicate whether that transaction has overflown. LTM stores speculative data in the cache, so that a commit only requires flushing the bits. On an abort, however, a processor must restore the previous values from main memory.

3.3 TTM

Finally, we base our study on a third version of transactional memory, Thread-Level Transactional Memory (TTM) [6]. TTM itself actually specifies an interface, not a specific configuration. By doing so, Moore et al. aim to deliberately separate transactional semantics from a particular implementation. This lets the same transactional program to run on arbitrary combinations of hardware and software. The authors define their interface in terms of threads. The abstraction encompasses both cacheable virtual addresses and threads' user-visible registers. Programmers indicate transaction boundaries with explicit `begin_transaction`, `commit_transaction`, and `abort_transaction` instructions. These calls depend on library-level support mechanisms which in turn depend on lower-level support mechanisms from hardware. When the user begins a transaction, the thread must allocate virtual memory space for a check-pointing log. In the cache, read and write bits track the read and write sets of the transaction, and coherence protocols detect conflicts. TTM resolves these by either stalling or aborting the younger of the offending transactions. The first time a transaction changes a line, it lazily writes the "before image" to the associated log. Commits flush the caches' read and write bits. TTM offers a significant degree of flexibility in abort handling. The hardware itself may restore values from the log, while a software abort handler performs all other operations. Note that since TTM stores the log in the thread's virtual address space, the log itself is also cacheable, thus improving the performance of software that must access it.

Moore et al. show several TTM implementations [6]. All of them, including the basis for this project, model a cache-coherent shared-memory multiprocessor with private, write-back, write-allocate, and set associative L1 and L2 caches. The authors extend the processor to include a TTM mode bit, nesting count, and log pointer; logging occurs at the granularity of a cache-block.

On the first transactional access to a block, the TTM mode bit is set and the previous value is copied to a shadow register file, which may eventually update the virtually-addressable log. This TTM work focuses on write-invalidate MSI protocols, where loosely synchronous timestamps order transactions. When the protocol detects a conflict caused by an younger transaction, Moore et al.'s implementation stalls the current transaction. In contrast, if the other transaction is older, the current transaction aborts. This guarantees freedom from both deadlock and starvation. The TTM paper [6] presents results for two coherence protocols: broadcast and directory. While both can record accesses and detect conflicts, they differ slightly in how they handle overflows. The broadcast protocol utilizes a two-bit Bloom filter, (one bit each to indicate read and write access) which is set appropriately whenever a block is evicted from the corresponding line. Note that this may cause false positives. The directory protocol extends the idea of silent replacements to a sticky modify state. In this state, when a cache writes back a particular line, the directory still forwards it requests. This owner will also have immediate M access and must set the read and write bits the next time it accesses the same block.

Despite the abundance of transactional memory proposals [2] [3] [5] [6], significant challenges still remain. Many of them have to do with smoothing interaction with the operating system. For instance, transparent activities such as paging and context switches should not affect the correctness of transactions. Current implementations handle such events by rolling back, but this seems unnecessarily wasteful. Similarly, transactions should have more definite semantics even for user-visible activities such as system calls, interrupts, and exceptions. If unrelated to the current transaction, ideally they should not affect its execution. Another interesting issue is input and output. Implementations may choose to buffer output until transaction completion, but this may cause problems if a later piece of the same transaction must use it. Likewise, any input used must be stored so that it can be re-read in the case of an abort. Such strategies naturally cause complications for real-time applications. The semantics of transactions themselves also offer some complexity. For instance, how should hardware treat nested transactions? Both TCC and TTM subsume inner transactions, which means an abort undoes work all the way back to the outermost level of nesting. This solution clearly lacks sophistication, especially in terms of performance.

To summarize, while researchers have made significant progress in exploring transactional memory, much work remains. The rest of this paper will tackle one piece of this greater project: how transactional memory, TTM in particular, interacts with a variety of coherence protocols.

4 Coherence Protocols

Cache coherence protocols are used by TTM for conflict detection during the execution of a transaction. The conventional cache coherence protocols are modified for this purpose. In this section, we will take a brief look at some of these protocols and their modified versions used by TTM.

4.1 Directory Protocol

This is a standard MOESI directory protocol. For each cache line in the memory, the directory maintains the list of caches that contain the corresponding data. It acts as an ordering point for all the coherence and data requests in the system. The protocol is enhanced with migratory optimization.

4.2 Hammer Protocol

This is a simplified version of the AMD hammer protocol. Its also a MOESI protocol. However, the directory does not maintain any list of sharers. It just keeps track of whether memory is the owner for any particular cache line. If not, it broadcasts the cache requests to all the caches. It also implements migratory optimization.

4.3 Token Protocol

This protocol uses tokens to keep track of coherence state. Each line of data has a fixed number of tokens associated with it through the life of a system. A cache requires atleast one token to read the data and requires all the tokens to write it. A special *Owner* token is used to keep track of ownership. This leads to a single writer, multiple readers implementation. A simple token counting scheme is used to maintain correctness in the protocol. Persistent Requests are used to prevent starvation. The caches first put out transient requests for data. These requests need not be serviced by other caches. In which case, the requesting cache sends out another transient request. After the number of retries reaches a *threshold*, it puts out a persistent request which has to be serviced by other caches.

4.4 Xact_directory Protocol

A modified version of the *Directory* protocol. It is modified for supporting TTM. Caches contain additional data for identifying data accessed during a transaction. Cache conflicts for lines resident in the caches are detected in the cache itself. For transactions that had cache overflows, Sticky-M bits are used for conflict detection.

4.5 Xact_hammer Protocol

A modified version of the *Hammer* protocol. The modifications made to support TTM are pretty similar to the modifications made to the *Directory* protocol. Since all requests are broadcast to the caches, bloom filters can be used for conflict detection.

All the above protocols were taken from existing implementations. The first two protocols are used in the study of conventional systems using lock-based concurrent programming. The transactional versions are used for the study of the transactional memory system, TTM.

5 Token Coherence for Transactional Memory

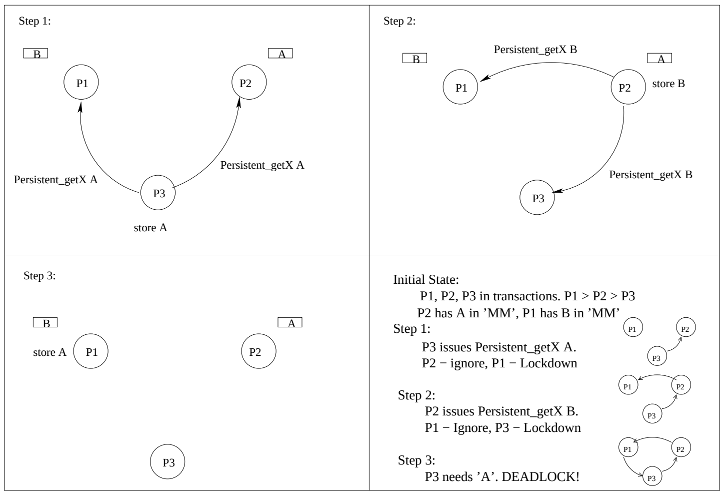
In this section, we present a modified version of token coherence protocol to support TTM. The base protocol is tokenB, which a MOESI token coherence protocol where all cache requests are broadcast.

5.1 Design Overview

We now give an overview of the major modifications made to the protocol.

- All outgoing requests from the cache are tagged with a timestamp which is loosely based on the timestamp mechanism used in TLR. Timestamps can be used to impose a global ordering among all cache requests. A transaction is assigned a timestamp when it first starts off. All the cache requests issued during this transaction will contain this timestamp. Non-transactional cache requests are also assigned a timestamp before they are issued. All the timestamps are loosely synchronized.
- Cache lines are augmented with transaction state bits that indicate if the line was read/written during the current transaction.
- Conflict detection mechanism is inserted. Bloom filters are added to each cache for conflict detection in case of cache replacements.
- Conflict resolution mechanisms are put in place. On a conflict, the current transaction can either abort or block.

The addition of timestamps gives us additional flexibility in designing the protocol. We present two versions of the protocol based on how they deal with transient requests.



Initial State:
P1, P2, P3 in transactions. P1 > P2 > P3
P2 has A in 'MM', P1 has B in 'MM'
Step 1:
P3 issues Persistent_getX A.
P2 − ignore, P1 − Lockdown
Step 2:
P2 issues Persistent_getX B.
P1 − Ignore, P3 − Lockdown
Step 3:
P3 needs 'A'. DEADLOCK!

Figure 3: Potential Deadlock with Persistent_Request Ordering in Transactions

5.2 xact_token1

5.2.1 Transient Requests

In a Non-Transactional Mode, all the state transitions are similar to the base token coherence protocol. However, when the cache is in a Transactional Mode, the protocol first checks if the incoming request conflicts with a line accessed during the current transaction. If it does not then it is treated as it would be in the base case. If it does conflict, then the timestamp of the incoming request is compared with that of the transaction. If a request contains an older timestamp, then the current transaction has to abort, else the current transaction just ignores the request.

5.2.2 Persistent Requests

Token coherence protocol issues a persistent request when it detects possible starvation, i.e after a certain number of transient requests fail. The protocol responds to a persistent request by first specifying an ordering on all the persistent requests at that particular moment. It then satisfies each persistent request by 'locking down' the corresponding cache line in all the other caches. The same semantics are observed if the cache is in a Non-Transactional Mode. However if the cache is in the middle of a transaction, it aborts the current transaction.

Timestamps could not be used for ordering persistent requests from transactions because of a potential deadlock situation. Figure 3 gives an example. In the example, three processors $P1, P2$ and $P3$ are each executing a transaction. $P1$ has cache line B in its private cache and it was modified during the current transaction. Similarly $P2$ has cache line A in its private cache and it was modified during the current transaction. The transaction timestamps specify the processor priority as $P1 > P2 > P3$. First, $P3$ puts out a persistent request for line A. $P2$ ignores the request, though it has the data, because of higher priority. $P1$ goes into a *lockdown* state, i.e it forwards any tokens it receives for this line to $P3$. So now $P3$ waits for $P2$ to finish the transaction and release the data. Next, $P2$ issues a persistent request for line B. $P1$ ignores the request and holds it off till it can finish its own transaction. So $P2$ waits for $P1$ to complete its transaction and release data. Finally, suppose $P1$ wants to issue a request for line A. It cannot do so because $P3$ has an outstanding persistent request for that line. So $P1$ waits for $P3$ to remove the persistent request. And we end in a deadlock situation.

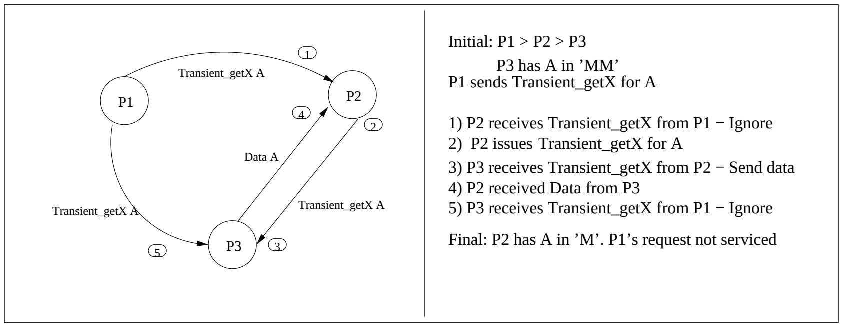
5.3 xact_token2

In the first version of the protocol, transactions abort unnecessarily on a persistent request even if there is no conflict. So the second version, attempts to minimize the number of persistent requests issued in the system. This is done with the help of timestamps. They are used to specify a global ordering of all the transient requests for a particular cache line.

5.3.1 Transient Requests

All Transient requests are treated based on their timestamp. Requests with older timestamps are serviced while requests with younger timestamps are ignored. If a transaction receives a conflicting request with an older timestamp it aborts. Intuitively, this places an order in which all requests are serviced, since at any point there is only one *oldest* request in the system.

However, tricky race conditions could lead to the *oldest* request not being serviced, in which



Initial: P1 > P2 > P3
P3 has A in 'MM'
P1 sends Transient_getX for A

1) P2 receives Transient_getX from P1 − Ignore
2) P2 issues Transient_getX for A
3) P3 receives Transient_getX from P2 − Send data
4) P2 received Data from P3
5) P3 receives Transient_getX from P1 − Ignore

Final: P2 has A in 'M'. P1's request not serviced

Figure 4: Potential Starvation with Transient_Request Ordering

case it has to be reissued. Figure 4 gives an example. We have three processors $P1, P2$ and $P3$. $P3$ has cache line A in its private cache in a modified state. Processor $P1$ first puts out a transient request for cache line A. This request first reaches $P2$ which ignores it since it does not have the line in its cache. After some time, $P2$ issues a transient request for line A which reaches $P3$. Note that the earlier transient request from $P1$ has not reached $P3$ yet. $P3$ services the request from $P2$ and sends the data to $P2$. Now the initial request from $P1$ reaches $P3$ at which point it ignores it since it no longer has the data. Thus $P2$'s request which was issued after $P1$, gets serviced even though it is not the oldest.

5.3.2 Persistent Requests

As noted above, even with the modified transient request semantics, we could end up with a potential starvation scenario. Hence we would still need persistent requests. The response to persistent requests is similar to that in xact_token1. Transactions abort on a persistent request, even if its for a non-conflicting line. However due to global ordering of requests, we expect persistent requests to be less frequently used.

5.4 Debugging

The protocol was tested using random test drivers. These drivers generate random requests to the caches in a way that stresses the protocol and tries to take it through all the transitions. A

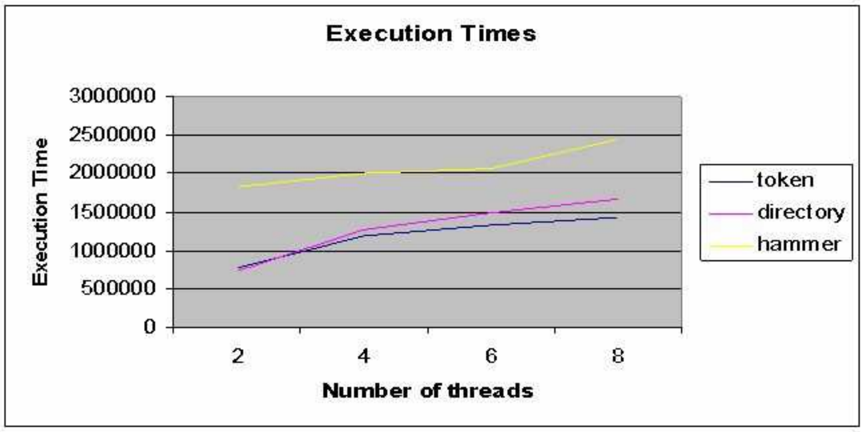


Figure 5: Execution times for bounded buffer with varying threads

number of bugs were identified through random testing. While the random tester identifies the manifestation of these bugs, the actual identification of the bug required going through thousands of lines of log files. While this was time-taking, the experience increased our confidence in the correctness of our protocol.

6 Results

The results were gathered using the various coherence protocols in conjunction with TTM. TTM is built on top of a Simics full-system simulator and customized memory models built using the Wisconsin GEMS toolset. We use parallel applications from the SPLASH-2 benchmark suite for evaluating conventional lock based programs. These were also modified to run with transactions and used for evaluating transactional programs.

Figure 5 presents the results for bounded buffer with varying number of threads using different transactional cache coherence protocols. Each thread is bound to a single processor. The program was executed for a fixed number of transactions (10000). We notice an increase in the execution times as the number of threads increase. This might seem counter-intuitive at first look. However, we observe this increase due to a corresponding increase in the number of transaction aborts in the system. At 8 threads, upto 40% percent of the transactions were observed to abort before completion. Hence transactional memory systems would perform well only on systems where the contention for shared data is fairly low and hence the frequency of aborts is also low.

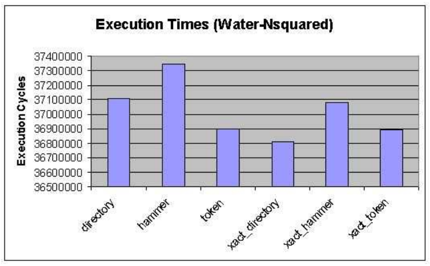


Figure 6: Execution times for WATER

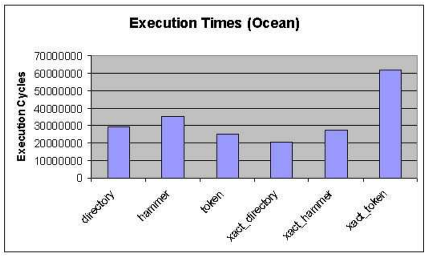


Figure 7: Execution times for OCEAN

We next present the results for four applications from the SPLASH-2 benchmark suite. We have the results for transactional token coherence protocol for two of them. We also observed that the second version of transactional token coherence outperformed the first version in all the cases. So we present the numbers only for the second version. The results show that the performance depends on the workload. While we notice performance improvements for transactional protocols in case of water, raytrace and ocean, barnes seems to degrade with transactional memory. We also note that transactional token protocol does not perform well over the other transactional protocols especially in the case of Ocean. It was observed that the protocol aborts in large transactions involving considerable number of system calls. Re-execution of these transactions causes significant overhead. This indicates the need to prioritize transactions.

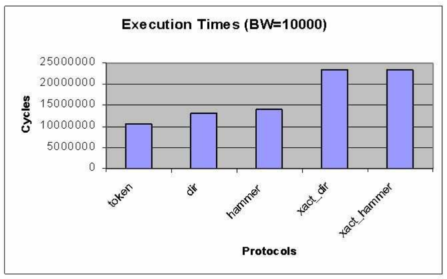


Figure 8: Execution times for BARNES

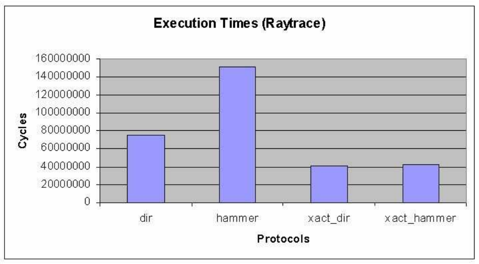


Figure 9: Execution times for RAYTRACE

7 Conclusions

Transactional Programming seems to make concurrent programming easier. It enables the implementation of fast and correct parallel programs without significant intellectual effort when compared to lock-based parallel programs. However transactional memory programs need not always result in performance improvements over carefully designed lock-based programs. Careful fine-tuning of transactions might still be needed to achieve performance gains. Finally, we present a transactional token coherence protocol that supports TTM. The presence of cache request ordering in TTM presents significant opportunities to optimize the protocol.

References

[1] Herb Sutter, ”The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software,” *http://www.gotw.ca/publications/concurrency-ddj.htm* March 2005

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, Sean Lie, “Unbounded Transactional Memory”, *Proc. of the Eleventh International Symposium on High-Performance Computer Architecture*, February 2005.

[3] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun, “Transactional Memory Coherence and Consistency”, *Proc. of the Thirty-First International Symposium on Computer Architecture (ISCA)*, June 2004.

[4] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, Kunle Olukotun, ”Programming with Transactional Coherence and Consistency(TCC)” *Proceedings of Annual Symposium on Programming Languages and Operating Systems (ASPLOS)*, 2004

[5] Maurice Herlihy and J. Elliot B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures”, *Proc. of the Twentieth International Symposium on Computer Architecture (ISCA)*, pp 289-300, May 1993.

[6] Kevin Moore, Mark Hill and David A. Wood, “Thread-Level Transactional Memory”, *Submitted to Parallel Architecture and Compilations Techniques (PACT)*, March 2005.