

Transactional Memory

MEng Thesis Proposal

Sean Lie

sean_lie@mit.edu

MIT Laboratory of Computer Science

Thesis Advisor: Krste Asanovic

May 14, 2003

1. Background

Shared memory parallel architectures present a single unified address space to each processor. Usually the memory is physically distributed across the system but each processor is able to access any part of it through a single address space. The system hardware is responsible for presenting this abstraction to each processor. Communication between processors is done implicitly through normal memory load and store operations. For this reason, shared memory architectures generally have much lower processor-to-processor latencies than message passing architectures which often rely on software to handle communication. In addition, the shared memory programming environment is more natural than that of message passing and is better suited for many applications.

Often the ability to perform several memory operations atomically is required for correct program execution. Unlike message passing systems, achieving such atomicity in shared memory is a nontrivial task. Traditionally, a technique called locking has been used to solve this problem. A lock is a memory location that, by convention, protects a block of code that needs to be run atomically. Once a processor obtains the lock, that processor can execute the atomic block. All other processors wanting to execute that code must wait until the lock is released. The processor holding the lock must release it once it is done executing the atomic block. This entire mechanism is a convention that is maintained by software alone. The hardware only provides the necessary mechanisms to obtain, release, and check the status of any lock.

2. Introduction

Dealing with locks is a very difficult aspect of parallel programming in shared memory. Locking is not natural for programmers and thus can often lead to undesirable results such as deadlock when used incorrectly. Even the simple task of locking two independent objects can result in deadlock if done incorrectly. There are standard methods of avoiding such undesirable

situations but they are often complex and difficult to reason about. Therefore, having to think about locks simply distracts the programmer from the actual algorithm being implemented.

For this reason, programmers often use very conservative locking in the interest of correctness and ease of implementation. Conservative locking achieves these two goals but can result in very poor performance since it can hide the available parallelism in the code. Moreover, the very concept of locking is inherently conservative. A lock may be obtained and released without any attempts by other processors to obtain that lock. Such a situation is very common when contention is low. This implies that there are often very few conflicts between processors running atomic blocks. Therefore, locking is not necessary for the most part, but is nevertheless required to ensure correctness in the few cases where a conflict does occur. Since the action of obtaining and releasing a lock can be very expensive, the use of locks (even when optimized) can lead to suboptimal performance and unnecessarily high resource overhead.

Conventional locking is not the most efficient means of achieving atomicity. Ideally, the programmer should have the ability to perform several memory operations atomically without having to worry about all the issues associated with locking. Transactional memory achieves this goal. Transactional memory is intended to replace conventional locking techniques to achieve higher performance and a more intuitive programming environment.

3. Transactional Memory

Transactional memory is a hardware mechanism that allows the programmer to define atomic regions (called transactions) containing memory accesses to multiple independent addresses. Atomicity requires that a valid execution of multiple regions must produce the same result as a possible serial execution. Instructions are added to the processor ISA to provide the programmer a method of defining these transactional regions. The hardware is responsible for ensuring that every transaction is executed atomically when viewed from the global memory system.

Our current design uses a shared memory architecture similar to that of the SGI Origin with the MIPS R10k processor. All references in this document to system and processor architecture refer to the SGI Origin and the MIPS R10k. In our current design, instructions within a transaction are executed on one processor speculatively. All memory addresses accessed by the transaction are marked. If another processor attempts to access any of the transactional addresses, the transaction is aborted. Transaction abort consists of discarding all data written by the transaction thus ensuring that the memory system appears as it did before the transaction was started. Therefore, other processors attempting to access transactional addresses will execute as if the transaction was never started. If, on the other hand, the execution of the transaction is allowed to complete without any conflicting accesses, the transaction is committed. Transaction commit ensures that all data written to the memory system by the transaction is visible to other processors for future accesses.

In our current design, the ability to detect conflicts is achieved by modifying the cache and cache-coherency mechanisms. On all memory accesses, data is brought into the cache. When the access is transactional however, the cache line containing the data is flagged. The cache will receive a normal cache-coherency message from any processors attempting to access that data. When such a message is received, a simple check of the cache-line flag can determine if an abort should occur.

A modification to the cache mechanisms also allows speculative execution of transactions. All transactional stores modify the data in the cache in the same way normal stores do. However, write back of transactional data to the memory system is only allowed after the transaction has committed. Since the cache is simply a copy of the actual data in memory, all transactional data in the cache can be discarded by invalidating the corresponding cache lines. Similarly, a transaction commit can be done by clearing all the transactional flags in the cache. Clearing the transactional flag makes the cache line visible to other processors by allowing write back to the memory system.

All cache actions are done atomically in hardware. Therefore, transactional memory operations suffer no performance impact. Very little change to the processor and cache hardware is necessary to support this design. In addition, the cache-coherency protocol, directory controllers, and all other hardware remain unchanged.

Using the cache alone to store transactional state, however, imposes a limitation on the size of transactions. Imposing such a limitation is awkward for the programmer since cache sizes vary between different systems and even between generations of the same chip. For this reason, our design will allow transactional state to overflow from the cache into normal memory. It is likely that this mechanism will impact performance for very large transactions but support will be available if necessary.

The importance of abort handling can be overlooked since aborts are likely to occur infrequently. However, a suitable hardware-software interface must exist to provide an efficient and graceful means of abort recovery. Our design will provide such a mechanism. In addition, it will give the programmer the freedom to decide what actions to take on an abort. For instance, different back-off or retry policies may be used in different situations.

Our design of transactional memory is intended to solve two important problems. The first is that of programming ease. Transactions provide the programmer with the ideal mechanism to achieve atomicity since conventional locking problems are completely avoided. The second problem being addressed is that of performance. Using transactional memory will result in higher performance for conservative code since atomic regions are speculatively executed. If it is possible to run the same atomic region on more than one processor while maintaining atomicity, transactional memory allows that parallel execution whereas conventional locks enforce serialization. In addition, since locks are simply memory locations, a transaction inherently has fewer memory operations than the equivalent atomic region using locks. Therefore, transactional memory code will offer higher performance than even the most optimized code using conventional locks.

4. Related Work

The notion of transactional memory was first introduced by Herlihy and Moss. Their implementation was an extension to the cache-coherence protocol and cache mechanisms. All transactional state and data is stored in the cache. Therefore, hardware modifications to only the processor and the cache were required. Their implementation was the basis of our initial design. However, their implementation was intended for very small transactions and thus did not support transaction sizes larger than the cache.

Herlihy and Moss's primary design goal was to provide a mechanism to make performing atomic operations easier for the programmer. However their simulation results showed that, in

some cases, transactional memory offered a significant performance increase over conventional locking techniques. This result was primarily due to the fact that lock memory accesses is not needed when using transactional memory.

The same performance issue was independently studied by Rajwar and Goodman. They proposed that, in most cases, locks do not have to be acquired for correct program execution. In such cases, locking instructions can be predicted as being unnecessary and elided. Their Speculative Lock Elision (SLE) scheme allows multiple threads to concurrently execute atomic regions protected by the same lock. Their results showed that, for many benchmarks, most locks can be elided. SLE was shown to offer a significant speedup for these benchmarks.

Rajwar and Goodman's results offer evidence that our transactional memory design will offer higher performance than conventional locking. Though SLE offers many of same features as transactional memory, SLE is inherently still a locking scheme designed to run conventional locking code. In fact, SLE reverts to conventional locking (actually obtaining and releasing locks) when lock elision cannot be done. Therefore, it does not offer one of the fundamental advantages of true transactions, programming ease. With SLE, the programmer must still deal with locks and all their associated problems.

In SLE, atomic regions are executed speculatively much like transactions in our design. However, SLE was designed only to execute small atomic regions. Speculative data is stored in a write-buffer that is only written to the cache when the lock elision has been validated. If a misprediction is detected rollback to the lock acquisition is done using either a register checkpoint or the reorder buffer. Therefore, atomic regions are limited to the size of the reorder buffer and the cache write-buffer. In modern processors, these structures are much smaller than even the primary cache.

Our design of transactional memory is very much based on Herlihy and Moss's original implementation. However, our design is intended to make several fundamental improvements on the original. Motivation for our design, at least in terms of performance, is based on the SLE notion that conventional locks often hide parallelism. However, the SLE and Herlihy and Moss implementations are very limiting and our design strives to correct these deficiencies.

5. Research Goals

In my thesis I will present a transactional memory design for a shared memory parallel architecture. The main design goals are, as stated previously, to provide a mechanism for achieving atomicity that is easier to use and that provides higher performance than conventional locking. In designing the system, the focus will be on evaluating design alternatives and their tradeoffs. Most of my work will be on the hardware aspects of the architecture. However, it is important that the hardware implementation presents a useful interface to the software. Therefore, some investigation of the hardware interface and its semantics will be done as well.

To evaluate design decisions, the UVSIM software simulator will be used. UVSIM is an execution-driven simulator that accurately models the MIPS R10k processor in a shared memory system similar to the SGI Origin 3000. It was developed by the University of Utah based on the RSIM simulator originally developed at Rice University and the University of Illinois at Urbana-Champaign. Potential design choices will be implemented in the simulator for evaluation. UVSIM can accurately run normal IRIX binaries. Therefore, it is possible to evaluate the effects of design decisions on the performance of normal codes and benchmarks.

6. Design Considerations

There are many factors that need to be considered in the design and implementation of transactional memory. The following sections outline some of the factors that have been identified at this point and that will be investigated as part of the project. Design alternatives will be evaluated based on their effectiveness and performance.

6.1 Context Switches

In our current design, all normal transactional state is stored in the processor cache. However, a cache is a hardware structure that is coupled to a processor. A transaction, on the other hand, is coupled to a process. When a process is running on a specific processor, the process has complete control of the cache. However, context switches can change the running process at any time, even during a transaction. On a context switch, the running process is separated from the cache. When the process is eventually switched back, the cache state may be completely different. In the case of normal memory operations, this is not a problem since the cache is completely abstracted away from the process. In the case of the transactions, however, a context switch separates the transaction from its speculative state.

There are several ways to solve the context switching problem. The easiest solution is to simply abort all transactions on a context switch. Aborting the transaction discards all transactional state from the cache. The transaction can simply be retried when the process is switched back. However, this scheme may result in many unnecessary aborts. For example, a transaction may cause a TLB refill and, for that reason, be aborted. In addition, aborting on every context switch poses a more fundamental problem for large transactions. It becomes possible for the length of a transaction to prevent it from ever completing because context switches occur at regular intervals in modern systems. Since one of the goals of our design is to allow for large transactions, such an approach must be avoided.

Allowing a transaction to span context switches, however, is not a trivial task. The transaction must be stalled when the current process is switched out and then allowed to continue when the process is switched back. However, the new process running after the context switch may use transactions as well. In addition, other processors, or even the new process, may make memory accesses that conflict with the stalled transaction. Such conflicts need to be detected and the stalled transaction should be aborted.

6.2 Processor Pipeline Interaction

Modern superscalar processors allow speculative out-of-order execution. This technique helps maintain high throughput by masking the large branch misprediction penalties and highly non-uniform memory latencies of modern processors. As a result, the cache is often accessed speculatively. In our current design, all normal transactional state is stored in the cache. Therefore, to maintain performance and correctness, it is important to ensure that transactional state is written to the cache at an appropriate time. Transactional instructions must be worked

into the processor pipeline effectively so that the existing speculation and out-of-order execution mechanisms continue to perform their function.

Normal loads from memory are generally permitted to access the cache speculatively but stores are not. This policy ensures that the memory is always consistent with the instruction most recently committed. To maintain performance, transactional loads must be allowed to access the cache speculatively just as normal loads do. However, if transactional state is written in the cache on speculative loads, the cache will no longer be consistent with committed instructions. Several issues arise if this policy is adopted. Firstly, the speculative transactional instruction may be flushed from the processor pipeline by branch misprediction or exception. Therefore, there must be a mechanism to annul any speculative transactional state in the cache. In addition, unwanted stalls in the processor pipeline may be caused when executing adjacent transactions. Since the cache can only store state for one transaction, the processor cannot speculatively execute past the current transaction into the next. These stalls can result in a significant performance penalty in cases where there are many small back-to-back transactions.

The problems introduced by storing speculative state in the cache can likely be solved. However, the alternative policy of storing speculative state in the processor avoids many of these problems. As in normal systems, the cache remains consistent with committed instructions. Speculative transactional state is stored in the address queue of the processor. Since the address queue is already used to hold speculative data, it is the ideal structure to use in this case. Speculative transactional loads can still be allowed to access the cache but no transactional state is written until the instruction commits. This policy makes better use of the existing processor architecture. However, all transactional loads require two accesses to the cache; one to access the data speculatively and another to write the transactional state. Cache bandwidth will increase and may potentially cause performance problems.

6.3 Abort Recovery

Aborts are likely to occur infrequently however there needs to be a mechanism that allows transactions to be restarted. The processor state at the start of a transaction needs to be saved and, if necessary, restored on when the transaction is retried. From the programmer's point of view, only the register values need to be restored since the abort mechanism will restore the memory system to its original state. Therefore, saving the processor state can be done entirely in software. However, doing so may result in performance loss especially in small transactions since the state needs to be saved before every transaction. Fortunately, if state is being saved entirely in software, compiler optimizations can be applied to reduce the number of saved registers. However, even a few added instructions at the beginning of every transaction may result in a large performance penalty for small transactions.

Processor state can also be saved and restored entirely in hardware. In fact, existing hardware mechanisms already save some processor state on every branch decode to allow misprediction recovery. When a branch is decoded, only the register rename table and the free list are saved. These two structures are sufficient since speculation past a branch is restricted to the length of the reorder buffer. Physical registers do not need to be saved since they are not overwritten until instructions are removed from the reorder buffer on instruction commit. However, in the case of transactions, physical registers need to be saved since speculation of the transaction can proceed well beyond instruction commit. In considering such a hardware

implementation, it is important to ensure that the mechanism does not require too much overhead and does not impose any additional performance restrictions on the processor pipeline.

In addition to restoring the processor state, there needs to be a way of notifying the software that an abort has occurred. The two general methods of detecting abort are analogous to polling and exceptions. In the first case, the hardware interface offers a way of checking if the current transaction has been aborted. In this scheme, the software is then responsible for periodically checking (or polling) the status of the current transaction. The `VALIDATE` instruction was used for this purpose in the Herlihy and Moss implementation. For performance reasons it is desired to poll very infrequently. However, very frequent polling may be required just to ensure correct program execution. Since all transactional memory effects are discarded on an abort, continuing to execute transactional code normally after an abort can result in unintended behavior such as infinite looping.

The alternative to polling is using a technique similar to exceptions. Once the transaction is aborted, the processor automatically jumps to an abort handler. Once in the abort handler, the software is free to perform tasks such as back off and restart. However, such an approach is only effective if the processor state can be saved and restored entirely in hardware with no performance penalty. Saving the processor state in software would require too much overhead since every register would need to be saved when a transaction is started. Since a direct jump to the abort handler is possible from any instruction, the compiler cannot rely on software conventions (such as subroutine calling conventions) to reduce the number of registers saved.

6.4 Nested Transactions

A transaction is nested if it exists completely within another transaction. There are several ways to deal with nested transactions. The easiest is to simply not support nested transactions at all. The ISA can specify that a transaction must be ended before another can be started. Though such an approach will make the hardware implementation simpler, it does not provide a very useful interface for the programmer. It is desirable to allow the programmer to make subroutine calls from within a transaction. These subroutines may themselves contain transactions since they may be called from within normal code as well as transactional code. To prohibit such situations imposes an inappropriate restriction on the programmer.

With no hardware support for nested transactions, the programmer can still write subroutines that contain transactions as long as the compiler is capable of removing them. The compiler can maintain two versions of every subroutine, a transactional version and a non-transactional version. If the subroutine is called from within a transaction, the non-transactional version is called. Otherwise, the transactional version is called. Unfortunately, such a compiler approach can potentially result in double the code size. Maintaining two versions of every subroutine is also quite awkward and not very elegant. Therefore, some (even if very limited) hardware support for nested transaction is desired.

In general, we have the option to interpret nested transactions in one of two ways: i) treat every nested transaction as a separate transaction that can be aborted individually or ii) merge all nested transactions into one large transaction. Treating every nested transaction individually is ideal since only those transactions that have conflicts to be aborted. However supporting this option in hardware is complicated and the overhead required may be much too high. On the other hand, merging nested transactions lends itself to a much simpler hardware implementation.

When nested transactions are merged, all inner transactions are treated as being a part of the outermost transaction. All conflicts abort the outermost transaction and, in effect, abort all nested transactions as well. Some unnecessary aborts may occur but the performance penalty is minimal since most transactions are likely to be very short.

6.5 Overflow Handling

Since transactional state is stored in the cache, the cache size can potentially pose a limitation on the size of a transaction. To allow for larger transactions, transactional state must be allowed to overflow from the cache into another storage area. The most natural choice is to overflow into normal memory. However, the implementation of such an overflow mechanism has some subtle issues. If overflows are handled in software in the same way as exceptions, there is plenty of freedom in which data structures and algorithms to use. However, the software overflow handler must not be allowed to use the cache normally since doing so may cause another transaction overflow. In addition, since cache-coherency messages are used to detect conflicts, the cache-coherency mechanism must be able to handle conflicts that occur on overflowed locations as well. Such a scheme will inevitably require much cooperation between the software and some new hardware mechanisms.

The alternative is to handle overflow completely in hardware. The advantage of hardware overflow handling over the software approach is that the performance penalty will not be as large. The performance advantage may be very important since overflow checking can potentially be required on every cache-coherency message. However, since most transactions are likely to be very short, that performance advantage will not be apparent for most transactions. In addition, hardware overflow handling does not give as much flexibility as the software approach. The software mechanism can be modified, for example, to use data structures or algorithms that are better suited for different situations.

7. Progress

I have been working on this project for approximately one academic term. I have been working with the UVSIM simulator extensively to investigate how it can be used for evaluation. I have been working on many of the design considerations given in the previous section. In the case of processor pipeline interaction, abort handling, and nested transactions, I have done substantial design and implementation. I have also completely implemented a simple transactional memory system in the UVSIM simulator. The current implementation is similar to the Herlihy and Moss design. It does not support overflow or nested transactions. The current implementation is intended to serve as the basis for evaluating future design decisions.

7.1 Future Work

The following is an approximate timeline of when key components of the system will be designed and evaluated. The projected completion date of the project is June 2004.

Summer 2003

Finish design, implementation, and evaluation of processor pipeline changes
Start design and implementation of nested transaction and abort recovery mechanisms

Fall 2003

Finish evaluation of nested transaction and abort recovery mechanisms
Start design and implementation of overflow handling

Spring 2004

Finish evaluation of overflow handling
Finish writing thesis

References

Hennessy, J. L. and Patterson, D. A. [1996]. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, California.

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

Herlihy, M. and Moss, J. E. B. [1992]. *Transactional Memory: Architectural Support for Lock-Free Data Structures*, Technical Report 92/07, Digital Cambridge Research Lab, Cambridge, Massachusetts.

Pai, V. S., Ranganathan, P. and Adve, S. V. [1997]. *RSIM Reference Manual Version 1.0*, Technical Report 9705, Rice University, Houston, Texas.

Rajwar, R. and Goodman, J. R. [2001]. *Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution*, Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture, 294-305.

Rajwar, R. and Goodman, J. R. [2002]. *Transactional Lock-Free Execution of Lock-Based Programs*, Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems.

Yeager, K. C. [1996]. *The MIPS R10000 Superscalar Microprocessor*, IEEE Micro Volume 16 Issue 2, 28-41.

Zhang, L. [2001]. *URSIM Reference Manual*, Technical Report UUCS-00-015, University of Utah, Salt Lake City, Utah.