Hardware Support for Unbounded Transactional Memory

by

Sean Lie

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 7, 2004 [June 2004]

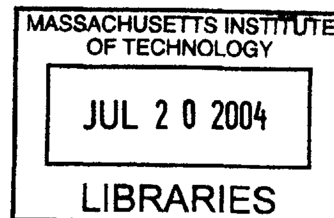
© Massachusetts Institute of Technology 2004. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 7, 2004

Certified by ....
Krste Asanovic
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses



BARKER

Hardware Support for Unbounded Transactional Memory

by

Sean Lie

Submitted to the
Department of Electrical Engineering and Computer Science

May 7, 2004

In partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I propose a design for hardware transactional memory where the transaction size is not bounded by a specialized hardware buffer such as a cache. I describe an unbounded transactional memory system called UTM (unbounded transactional memory) that exploits the perceived common case where transactions are small but still supports transactions of arbitrary size. As in previous hardware transactional memory systems, UTM uses the cache to store speculative state and uses the cache coherency protocol to detect conflicting transactions. Unlike previous hardware systems, UTM allows the speculative state to overflow from the cache into main memory, thereby allowing the transaction to grow beyond the size limitation of the cache. The clean semantics of UTM allow nested transaction support, nontransactional instructions, immediate aborts, a processor snapshot, and context-switching support; all features not found in previous hardware transactional systems. UTM was implemented in a detailed simulator, and experimental results show that it can be integrated with existing hardware straightforwardly while still performing better than conventional synchronization techniques.

Thesis Supervisor: Krste Asanovic
Title: Associate Professor, MIT CSAIL

Acknowledgments

Many of the ideas presented in this thesis were developed in collaboration with Krste Asanovic, Charles E. Leiserson, and Bradley C. Kuszmaul. Also, initial ideas were developed with the help of Marty Deneroff and Steve Miller from Silicon Graphics, Incorporated.

In particular, I would like to thank Krste for being a dedicated advisor. I also want to thank Charles for helping me develop my writing and presentation skills.

The compiler aspects of the evaluation were handled by C. Scott Ananian. I would also like to thank Scott for helping me understand his software transaction system as well as other software designs.

The lead software engineer of the UVSIM simulator was Lixin Zhang from the University of Utah. I would like to thank Lixin for his help during initial UVSIM development.

Last, but definitely not least, I would like to thank Jennifer Song, all my friends, and my family for their continued support.

This research was funded in part by National Science Foundation Grant ACI-032497, in part by the Singapore-MIT Alliance, and in part by a grant provided by Silicon Graphics, Incorporated.

Contents

1 Introduction **11**

2 The Case for Unbounded Transactions **15**

2.1 Atomicity in Parallel Systems 16

2.2 Problems with conventional locking 18

2.3 Transactions: a better atomicity primitive 22

2.4 Overcoming hardware limitations 25

3 Related Work **29**

3.1 Non-blocking synchronization 30

3.2 Software transactional memory 32

3.3 Hardware transactional memory 33

4 The UTM Design **39**

4.1 Design overview 40

4.2 ISA modifications 41

4.3 Transactional state in cache 44

4.4 Transactional state in main memory 47

4.5 Conflict detection 52

4.6 Processor snapshot 54

4.7 Nontransactional instructions 57

4.8 Context-switch support 60

5 Evaluation **67**

5.1 Evaluation environment 68

5.2 Overall performance 70

5.3 Transaction size and length 71

5.4 Memory latency overhead 74

5.5 Effect of pipeline modifications 75

5.6 Parallel transactional program behavior 77

5.6.1 The `NodePush` microbenchmark 77

5.6.2 The `Counter` microbenchmark 78

5.6.3 The `BinaryTree` microbenchmark 80

5.6.4 The `LinkedList` microbenchmark 82

6 Design Alternatives **85**

6.1 Integrated hardware-software approach 86

6.2 Linear overflow data structure 91

6.3 Software register snapshot 93

6.4 Nested independent transactions 96

7 Conclusions **103**

List of Figures

2-1 The node_push code example 16
2-2 A potential problem running node_push in parallel 17
2-3 node_push with global locking 18
2-4 node_push with incorrect fine-grained locking 19
2-5 A deadlock example 19
2-6 node_push with correct fine-grained locking 20
2-7 node_push with transactions 23

4-1 Speculative transactional storage in cache 44
4-2 Speculative transactional storage in cache and main memory 48
4-3 Overflow handler interface to cache and memory 49
4-4 Overflow hash table function and data structure. 50
4-5 Architectural register snapshot mechanism 55
4-6 A nontransactional logging example 58
4-7 Context-switch mechanism 61
4-8 Cache modifications for optimized context-switches 65

5-1 UVSIM simulation parameters 68
5-2 SPECjvm98 performance 71
5-3 SPECjvm98 transaction size and length distribution 72
5-4 SPECjvm98 transaction overflow statistics 72
5-5 SPECjvm98 memory access latency 74
5-6 Processor modification effects on SPECjvm98 performance 76
5-7 NodePush microbenchmark performance 78

5-8 Counter microbenchmark performance 79
5-9 BinaryTree microbenchmark performance 81
5-10 LinkedList microbenchmark performance 83

6-1 STM object structure 87
6-2 Additional checks performed on HTM operations 89
6-3 HSTM performance for various transaction sizes 90
6-4 SPECjvm98 performance with linear overflow data structure 92
6-5 Cache modifications for NIT implementation 97

Chapter 1

Introduction

Locks are conventionally used to achieve atomicity in parallel systems. Locks, however, have a host of problems such as deadlock. These problems introduce subtle correctness issues into parallel programs. Therefore, programming with locks is not an easy task.

Transactional memory [22, 23, 26, 27] has been proposed as an alternative means of achieving atomicity by presenting a more intuitive atomicity primitive to the programmer. Using transactional memory, the problems associated with conventional locks can be avoided. Transactional memory allows programs to read and modify many distinct memory locations atomically as a single operation, much as a database transaction [13, 14] can atomically modify many records on disk. The programmer simply specifies the scope of the transaction and the underlying system ensures it is executed atomically.

Although the implementation of transactional memory is undoubtedly more complex than conventional locks, previous work shows that hardware support can minimize the performance and implementation overhead [22, 23]. Unfortunately, previous hardware transactional memory systems restrict the size of a transaction to that of a hardware buffer such as a cache. Although this limitation simplifies the hardware implementation, it also exposes the transaction size limit to the programmer. Exposing such an implementation parameter greatly restricts the ability to use transactional memory in practice.

In this thesis, I propose a hardware transactional memory system that does not have a hard limit on transaction size. This system, called UTM (unbounded transactional memory), uses the cache and cache-coherency protocol to store transactional state and detect conflicting transactions in much the same way as previous hardware designs. Unlike previous schemes, however, UTM allows transactional state to overflow from the cache into main memory. Therefore, UTM transactions are not limited to the size of a specialized hardware buffer such as the cache.

UTM can be implemented in a modern microprocessor with few changes to existing hardware while still achieving better performance than conventional locking techniques. To accommodate UTM, only the cache and processor require modification. Therefore, UTM is a practical design that can be implemented in today's microprocessors with low risk. Further, the simple implementation does not entail a loss in performance. UTM was implemented in a detailed system simulator, and experimental results with the SPECjvm98 [51] benchmark suite show that UTM outperforms conventional locking in all cases.

UTM also provides many other features not found in previous transactional memory systems. The semantics are simple and support transaction nesting and immediate aborts. The semantics are provided using a hardware snapshot of the processor architectural state. UTM also permits logging or debugging code within a transaction through nontransactional instruction blocks. Lastly, context-switches are supported during transaction execution without aborting the running transaction.

In the following chapters, I describe the UTM design in detail and present qualitative and quantitative evaluations of the design. In Chapter 2, I discuss the case for UTM. I argue that unbounded hardware transactions are necessary for a practical and efficient system. In Chapter 3, I present previous transactional memory designs and other related work. I contend that although previous designs have many useful characteristics, they all lack features that prevent them from being useful in practice. In Chapter 4, I describe the UTM design in detail and discuss the reasons behind the design decisions. I show that UTM can indeed be implemented with only minor hardware modifications. In Chapter 5, I provide a quantitative evaluation of UTM

based on simulation results. I show that UTM has low performance overhead and, in fact, performs better than conventional locking in most cases. In Chapter 6, I discuss some design alternatives and their design trade-offs. I argue that UTM strikes a good balance between hardware complexity and programming ease. In Chapter 7, I discuss UTM's limitations but argue that they do not prevent UTM from being useful in practice.

Chapter 2

The Case for Unbounded Transactions

Locks are conventionally used to achieve atomicity in parallel systems. Conventional locks, however, have many fundamental problems. Locks are hard to use and have high performance overhead. Further, locks are inherently conservative and blocking. Transactions are designed to solve the locking problems. Transactions are easy to use and potentially have extremely low overhead. Further, transactions are also optimistic and non-blocking. Unfortunately, previous hardware transaction designs impose a limitation on transaction size and length. This limit restricts the practical use of hardware transactions.

In this chapter, I argue that unbounded transactions are necessary in a practical transactional memory system. In Section 2.1, I describe how conventional locks are used. In Section 2.2, I discuss the fundamental problems associated with locks. In Section 2.3, I describe the transaction concept and its benefits over conventional locks. In Section 2.4, I introduce the notion of an unbounded transaction that does not suffer from size or length limitations.

```
if (Flow[i] > Flow[j]) {
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X; }
```

Figure 2-1: The `node_push` code example. Flow X is being pushed from node i to node j only if node i has more flow than node j.

2.1 Atomicity in Parallel Systems

Shared memory multiprocessor architectures present a single unified address space to each processor. If one processor makes a change to main memory, the change is immediately visible to all other processors. Therefore, atomicity is often required for correct parallel program execution. Conventionally, this atomicity has been achieved through mutual exclusion locks. In this section, I describe the shared-memory architecture and how conventional locks are used.

Although shared-memory architectures present a single address space to all processors, the memory is usually physically distributed across the system. 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. Processors communicate implicitly through normal load and store operations on memory. Therefore, a load operation may result in fetching data from the memory or even the cache of a remote node. Data communication and cache-coherency is performed either over a common bus [11, 41] or over a network using directories [1, 35]. This mechanism is abstracted away from the programmer, however, which makes the shared-memory environment natural and well suited for many applications.

In shared memory programs, the ability to perform several memory operations atomically is often required for correct program execution. Consider, for example, the `node_push` code given in Figure 2-1 which pushes flow X from node i to node j only if node i has more flow than node j. Similar code can be found in graph algorithms such as the parallel push-relabel maximum-flow algorithm [3, 10]. The code in Figure 2-1 operates as expected when run on a single processor. Running the code on several processors, however, may give unexpected results.

Initial values: Flow[0]=5, Flow[1]=4

Processor 0	Processor 1
`if (Flow[0] > Flow[1]) {`	
	`if (Flow[0] > Flow[1]) {`
	`Flow[0] = Flow[0] - 2;`
	`Flow[1] = Flow[1] + 2; }`
`Flow[0] = Flow[0] - 2;`	
`Flow[1] = Flow[1] + 2; }`	

Final values: Flow[0]=1, Flow[1]=8
Expected values: Flow[0]=3, Flow[1]=6

Figure 2-2: A potential problem running node_push in parallel. The node_push code is run on two processors. The order in which instructions are executed is shown with time running down.

Consider the possible parallel execution of **node_push** shown in Figure 2-2. The result of that execution is not possible in a single-processor since the `if` condition would fail on the second push attempt. To get the expected result in parallel, it is necessary to execute the entire operation as one atomic operation. Although not illustrated in the example, it also is necessary to ensure that the increment and decrement occur as one atomic operation. For example, the statement [`Flow[j] = Flow[j] + X`] is usually translated into at least 3 instructions (load, increment, and store) in a RISC ISA. Therefore, it is necessary to ensure that these three instructions occur atomically as well.

Traditionally, mutual exclusion locks have 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 acquire, release, and check the status of any lock.

```
Lock(globalL);
if (Flow[i] > Flow[j]) {
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X; }
UnLock(globalL);
```

Figure 2-3: node_push with global locking. A single global lock globalL is used to ensure atomicity.

2.2 Problems with conventional locking

Although locks can be used to achieve atomicity, they inherently have three unavoidable problems. Firstly, locks are hard to use for programmers because they can lead to undesirable results such as deadlock when used incorrectly. Further, even when locks are used correctly, the addition of locks to serial code generally results in significant performance overhead. Lastly, locks are not well suited to exploit the concurrency in some applications since locks are conservative and blocking. In this section, I describe these problems in detail.

Locks are hard to use

The easiest way to achieve atomicity is with coarse-grained locks. Coarse-grained locking can be done in the node_push example by using a single global lock as shown in Figure 2-3. A single global lock protects the atomic block so that only one processor can run that block at any given time. Using coarse-grained locks, however, can result in poor performance since it can hide the available concurrency. For example, in node_push, theoretically, two processors can execute the same code block at the same time as long as they are operating on different nodes. A single global lock prevents such concurrency.

Since a fundamental goal of running an application in parallel is to increase performance, coarse-grained locking is often avoided in favor of fine-grained locking, which enhances concurrency. Unfortunately, fine-grained locking requires more locks, and the relationship between the locks can become complicated. For this reason, deadlock can become a problem once there is more than one lock. Deadlock is the situation

```
Lock(L[i]);
Lock(L[j]);
if (Flow[i] > Flow[j]) {
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X; }
UnLock(L[i]);
UnLock(L[j]);
```

Figure 2-4: node_push with incorrect fine-grained locking. A lock for each node is acquired before the atomic block and released after the atomic block.

Processor 0	Processor 1
Lock(L[0])	
	Lock(L[1])
Lock(L[1]) Waiting on Processor 1	
	Lock(L[0]) Waiting on Processor 0

Figure 2-5: A deadlock example. The node_push code with incorrect locking is run on two processors. The order in which instructions are executed is shown with time running down. No progress is made since processor 0 is waiting on processor 1 and processor 1 is waiting on processor 0.

where two processors cease to make progress since each processor waits on the other to release a lock. Unfortunately, deadlock can be hard to avoid.

To illustrate fine-grained locking and the problem of deadlock, consider using fine-grained locks in the node_push example. Since it is desirable for processors to operate on different nodes concurrently, it is natural to associate a lock with each node. Since node_push operates on two nodes within the atomic region, it is necessary to acquire the locks for both nodes. Figure 2-4 shows a naive incorrect implementation for node_push. A lock for each node is acquired before the atomic block and released after the atomic block. Although, in most cases the naive implementation works correctly, in some cases it deadlocks. As shown in Figure 2-5, it is possible for two processors to wait on each other indefinitely.

It is possible to avoid deadlock by simply ordering the lock acquisition as shown

```
if (i<j) {
  a = i;
  b = j;
} else {
  a = j;
  b = i; }
Lock(L[a]);
Lock(L[b]);
if (Flow[i] > Flow[j]) {
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X; }
UnLock(L[b]);
UnLock(L[a]);
```

Figure 2-6: `node_push` with correct fine-grained locking. Lock are acquired in order to avoid deadlock.

in Figure 2-6. Since locks are acquired in order, the deadlock situation from Figure 2-5 is not possible. It should be evident that deadlock avoidance adds significant complexity, however, even in such a simple example. In more complicated situations, reasoning about deadlock becomes even more error prone, leading to buggy programs.

When using locks, there is a tradeoff between programming ease and concurrency. It is always possible to use conservative coarse-grained locks but doing so greatly limits the concurrency and can result in poor performance. To achieve better performance, programmers can use fine-grained locking. Unfortunately, because of deadlock, programming complexity only increases as more fine-grained locking is used.

Locks have high overhead

Although fine-grained locking can increase concurrency, locks still add a significant performance overhead as it takes time to perform deadlock prevention and lock acquisition and release. As shown in Figure 2-6, even simple lock ordering adds a significant overhead to a small transaction. There are simply many more instructions to run. There may even be an additional unpredictable branch if the ISA does not have instructions such as minimum, maximum, or conditional-move. Further, lock ac-

quisition is conventionally done using load-linked/store-conditional [25] instructions or atomic read-modify-write [30] instructions such as test-and-set, fetch-and-add, or compare-and-swap. These instructions can incur higher performance overhead compared to normal memory instructions.

In additional to performance overhead, using locks adds a significant space overhead as well. Locks are memory locations. In our example, each node in the graph requires one lock. For a large graph, this space overhead can be quite large. Further, although in theory a lock only needs to be one bit in size, in practice every lock must reside on a different cache line. Giving each lock its own cache line prevents problems such as false-sharing [17, sec. 8.3] that are caused by the underlying cache-coherency mechanism used to obtain exclusive access to lock variables. Therefore, the space overhead incurred by locks is often significant and cannot be ignored.

Locks are conservative

Conventional 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 common in highly parallel applications where contention is usually low. 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. For example, in `node_push` the overhead of lock reordering and lock acquisition/release is incurred even if all other processors are accessing different nodes, which is likely in a large graph. Thus, locks are always used but only rarely needed. Further, locks must be acquired even if the `if` condition fails and no push is necessary. Therefore, even highly optimized fine-grained locks can lead to suboptimal performance and unnecessarily high resource overhead.

Locks are blocking

Conventional locking does not have a strong forward-progress guarantee. If a thread holding a lock is suspended for any reason, all other threads waiting for the lock do not

make any progress. Threads can be suspended for many reasons in modern systems since context-switches occur regularly. For example, if a thread running a small atomic region is switched out by the operating system, all other threads contending for the lock are left idle waiting for the thread holding the lock to be switched back in. The blocking nature of locks causes poor and unpredictable performance.

2.3 Transactions: a better atomicity primitive

The transaction is a primitive designed to provide atomicity without the problems associated with locks. Transactions are easier to use and potentially have much lower performance overhead. Further, transactions can exploit the concurrency often hidden by locks since transactions are optimistic and non-blocking. In this section, I describe the transaction concept in more detail.

Locking is used because it is the only way to achieve atomicity in current parallel systems. The programmer does not need locks per se. Atomicity suffices. Locks can achieve atomicity but they also distract the programmer from his true needs. Ideally, the underlying system provides an atomicity primitive that is deadlock free and able to exploit available concurrency while abstracting away the implementation details. The transaction is such a primitive.

A transaction, as in a database system, is an atomic block of code that can either commit or abort. If a transaction commits, the underlying system ensures that all memory operations within the transaction appear atomic with respect to other transactions or memory operations in the global memory system. If the underlying system cannot run the transaction atomically for any reason, it is aborted and the transaction appears to have not run at all.

Conceptually, the code within a transaction is run speculatively until the end of the transaction. If the speculative execution does not conflict with another transaction or memory operation, the speculative changes are committed, thereby committing the transaction. On the other hand, if a conflict occurs before the end of the transaction, the speculative changes are rolled back to the beginning of the transaction and it

```
xBEGIN;
if (Flow[i] > Flow[j]) {
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X; }
xEND;
```

Figure 2-7: `node_push` with transactions. The `xBEGIN` and `xEND` commands mark the start and the end of the transaction respectively. The underlying system ensures that the block is run atomically.

appears as if the transaction was not run at all. In the event that two transactions conflict while running speculatively, only one needs to be aborted and the other can be allowed to continue. Which transaction to abort is an implementation specific policy. In theory, both transactions can be allowed to continue as long as the underlying system ensures that each transaction always sees a consistent view of memory. For example, one can create multiple versions of the modified data as in multi-version database systems [4, 52]. The amount of bookkeeping necessary for such a system, however, often precludes an efficient hardware implementation.

Transactions are easy to use

Transactions are much easier to use than locks since the programmer need not worry about issues such as deadlock. As shown in Figure 2-7, using transactions is straightforward. The programmer simply wraps each atomic region in a transaction and the underlying system ensures that it is executed atomically.

Transactions can have low overhead

It is possible to implement the transaction primitive in hardware with negligible performance overhead. Herlihy and Moss [22,23] and Knight [26,27] propose transactional memory implementations that use the cache and cache-coherency mechanisms of modern parallel systems to provide the transaction primitive with virtually no overhead. These hardware systems use the processor cache to store the transaction's speculative state and use the cache-coherency protocol to detect conflicts. The specu-

lative state in the cache can be committed or aborted by simply flash clearing certain bits in the modified cache. Therefore, there is essentially no performance loss. More details about these systems can be found in Chapter 3. The previous work on hardware transaction systems provides evidence that transactions can be achieved with low overhead.

Transactions are optimistic

The speculative nature of transactions implies that transactions are inherently optimistic. Two transactions can run concurrently as long as they do not access the same memory location in a conflicting manner. The two transactions can even be separate instances of the same atomic region that would have been protected by a single lock conventionally. With a single lock, only one processor can execute the atomic region at any given time. The other processors are forced to wait even if no conflict occurs. With transactions, on the other hand, many processors may execute the same atomic region simultaneously in the absence of conflicts. Each processor simply executes speculatively independent of the other processors. If no conflict occurs, every processor can commit its transaction independently as well. For example, in `node_push`, two transactions can even operate concurrently on the same pair of nodes if both `if` statements fail since neither transaction is writing any data.

Transactions are non-blocking

A transaction guarantees forward-progress if all conflicting transactions suspend execution for any reason. The running transaction simply proceeds by aborting all conflicting suspended transactions. Therefore, unpredictable interruptions such as context-switches and exceptions cannot affect other threads. The non-blocking nature of transactions thus results in lower runtime and more predictable performance.

2.4 Overcoming hardware limitations

Although previous hardware transactional memory designs have low overhead, they also impose a restriction on the size and length of transactions. These limitations restrict the use of these systems in practice. In this section, I first describe these limitations in more detail. Then I describe the notion of an unbounded transaction that does not suffer from these limitations.

The designers of previous hardware transaction designs assume that transactions are generally small and short. Therefore, they claim it is acceptable to imposed a *size* and *length* limitation on transactions. For clarity, transaction *size* refers to the number of distinct memory locations touched by a transaction. Transaction *length* refers to the time it takes to run a transaction. Systems with size or length limitations are referred to as *bounded*. Systems without these limitations are referred to as *unbounded*. In previous bounded systems, transactions that are too big or too long are simply aborted.

The designs by Herlihy and Moss [22,23] and Knight [26,27] expose a strict transaction size limitation since they only use a hardware buffer to store the running transaction's speculative state. When the buffer is full, the transaction is aborted. Although such an implementation can lead to low overhead, it also imposes an awkward restriction. If a transaction aborts because of a space limitation, that transaction can never run successfully on the system.

The length limitation imposed by previous bounded systems is not a hard limit like the cache capacity. Instead, the length limitation arises from the fact that previous hardware systems do not support context-switches during a transaction. When a context-switch is performed, the running transaction is simply aborted. Since modern multithreaded systems have context-switches at regular intervals, such a restriction effectively limits the length of the transaction. Like the size limitation, this restriction is awkward. If a transaction aborts because it is longer than the process time slice, that transaction can never run successfully on the system.

Although the assumption that most transactions are small and short may still

hold in common applications, the bounded approach leads to a impractical system. In the remainder of this section, I discus in more detail the reasons why unbounded transactions are necessary.

The need for unbounded transactions

The main goal of transactions is to simplify concurrent programming. ¿From the point of view of a programming language or compiler, however, bounded transactions lead to complications that reduce their advantages over locking. For transactions to be truly useful in a systems setting, they must be unbounded.

The transactional size limitation exposes an awkward architectural parameter in the instruction-set architecture (ISA). Ideally, such a limitation should be an implementation parameter such as memory or cache size. Implementation parameters can vary from machine to machine of the same type. Unfortunately, the size limitation must be exposed as an architectural parameter since machines with different limitations cannot run the same programs. For example, an application binary that contains a transaction that touches 180 bytes will not run correctly on a system with a size limitation of 160 bytes. It will run correctly, however, on a system with a size limitation of 200 bytes. Often the cache, whose size is an implementation parameter, is used to store transactional data. Different revisions of the same processor may have different cache sizes and thus behave drastically differently when running the same transactional binary.

Even if the size limitation is exposed as an architectural parameter, many unsolved problems remain. For example, what should that limitation be? Should the system support transactions that touch 10 bytes? 100? 1000? And, how is this bound enforced? Suppose that the system supports transactions of 1000 bytes or less. What should the compiler or programmer do with a program containing larger transactions?

It is even unclear how the programmer or the compiler would find the size of the transaction. For example, modern programming languages such as C [28] seldom expose memory references to the programmer. In C, without the `volatile` keyword the programmer has no way of knowing whether a given memory reference actually causes

a memory operation or if the compiler optimizes it away. Since the programmer cannot find the size of the transaction, one might argue that perhaps the compiler can. Unfortunately, that task is just as difficult for the compiler. General programming practice allows the calling of subroutines without knowing how many memory references they make. Also, often the number of memory references is only determined at runtime. Of course, it may be possible for the compiler to expose the number of memory references to the calling function. Moreover, it may be possible for the compiler to perform conservative analyses and forbid practices such as using loops within transactions to restrict the possibility of dynamically sized transactions. Perhaps we can force that transactions are only accessible to the programmer through hand-coded runtime libraries. Although all of these options are technically viable, they prevent transactional memory from being a general system-building tool.

The transaction length limitation is equally difficult to deal with in a real system. Should the minimum process time slice be an architectural parameter? What does the operating system do if a transaction cannot finish within the process time slice? How does the programmer or the compiler know how long a transaction will run for?

Because none of these questions has a good answer, I contend that the size and length limitations of the underlying system should be viewed as implementation parameters which can vary without affecting the portability of binaries. Code should not need to be rewritten or recompiled to deal with a different internal hardware limit. Internal limitations may be visible as a matter of performance like the cache size but the same application binary should always run correctly on different revisions of the same machine. Therefore, from a systems perspective, unbounded transactions are essential to the design of a practical transactional memory system. Truly unbounded transactions are, of course, never possible since no system has unlimited resources. The visible limits should be sufficiently large, however, that programmers never encounter it. For example, there is technically a limit to the amount of virtual memory in a system, but it is sufficiently large that the programmer need not worry about it.

Chapter 3

Related Work

The concept of optimistic transactions was first described in the database context by Gray [13] and Reuter [14] and Kung and Robinson [32]. Since then, the concept has been proposed for normal parallel programs, not just those related to databases. The general goal is to make parallel programming easier and to exploit concurrency often hidden by conventional locks. Three general approaches have been explored: the non-blocking approach, the software transaction approach, and the hardware transaction approach. The non-blocking approach does not require hardware changes and has a forward-progress guarantee. The software transaction approach also does not require hardware changes since the atomicity primitive is provided through programming languages, compilers, and libraries. The hardware transaction approach provides the atomicity primitive through hardware and ISA modifications.

In this chapter, I describe related work in these three areas. Unfortunately, certain aspects of all previous systems restrict their use in practice. In Section 3.1, I describe non-blocking synchronization. I argue that non-blocking synchronization adds too much programming complexity. In Section 3.2, I explore previous software transactional memory designs. I argue that software designs have impractically high performance overhead. In Section 3.3, I describe previous hardware transactional memory designs. I argue that although hardware designs have low performance overhead, they all have restrictions such as size or length limitations that restrict their practical use.

3.1 Non-blocking synchronization

Non-blocking synchronization is a software technique that provides atomicity without mutual exclusion locks. Since locks are not used, their associated problems are also avoided. For example, non-blocking synchronization is optimistic and deadlock free. In addition, since it is a software technique, hardware changes are not necessary. Standard instructions such as compare-and-swap or load-linked/store-conditional are sufficient. Further, non-blocking synchronization has a strong forward-progress guarantee. Unfortunately, non-blocking synchronization increases program complexity and makes parallel programming more difficult. Several varieties of non-blocking synchronization have been proposed including lock-free [33], wait-free [18], and obstruction-free [20] synchronization. In this section, I describe these non-blocking techniques in more detail and discuss their impact on programming complexity.

Lock-free synchronization was first explored by Lamport [33] and later formalized by Herlihy [19]. A concurrent object is lock-free if *some* concurrent thread is guaranteed to complete an operation in a finite amount of time. A systematic methodology can transform any concurrent object into a lock-free object [19]. First, a copy of the concurrent object is made and used for modifications. The modified copy is then switched back into the concurrent data structure by changing a pointer with a load-linked/store-conditional operation. If the store-conditional fails, the operation is retried since another thread must have changed the data structure after the copy was made. The lock-free technique guarantees forward-progress since at least one of the threads attempting to modify the concurrent object will succeed.

Wait-free synchronization was later proposed by Herlihy [18] to have an even stronger progress guarantee. A concurrent object is wait-free if *every* concurrent thread is guaranteed to complete an operation in a finite amount of time. To achieve this stronger progress guarantee, wait-free objects are significantly more complex than lock-free objects. A methodology similar to that for lock-free objects can be used to transform any concurrent object into a wait-free object [19]. The thread also starts by making a copy to which modifications are made. In addition to making the copy,

the thread also globally announces its intended modification. Each time a thread attempts to make any modification, it also performs any outstanding modifications announced by other processors. As in lock-free synchronization, at least one of the concurrent threads will succeed. The successful thread, however, has also made the modifications for unsuccessful threads. Therefore, all the threads will be successful within a finite amount of time.

Both lock-free and wait-free objects are optimistic and have attractive progress guarantees. Therefore, the techniques exploit much of the concurrency that conventional locks tend to hide. Herlihy shows that lock-free and wait-free synchronization can outperform spin-locks in some situations.

Herlihy suggests that the systematic transformation of any object into a lock-free or wait-free object can be done automatically by the compiler. Therefore, in theory, the programmer need not deal with the added complexity. In practice, however, lock-free and wait-free techniques are rarely used. A possible explanation for the lack of acceptance is that implementing lock-free and wait-free objects is not as straight forward as Herlihy first imagined. If the process cannot be automated, the added programming complexity is clearly a significant deterrent. Even the simple wait-free dequeue operation Herlihy presents in [19] is difficult to understand and significantly more complicated than using conventional locks. Therefore, although lock-free and wait-free synchronization have many of the performance benefits of transactions, they do not address the programmability problem since they can make parallel programming even more difficult.

The programmability problem was, in fact, recognized by Herlihy who recently proposed another type of non-blocking synchronization: obstruction-free objects [20]. A concurrent object is obstruction-free if an isolated thread is guaranteed to complete an operation in a finite amount of time. An isolated thread is a thread running without any conflicts from other threads. Although the progress guarantee is much weaker than that of lock-free or wait-free objects, Herlihy shows that obstruction-free objects share many of their advantages while being easier to program. Unfortunately, there is no systematic transformation that can be performed by the compiler. Herlihy

shows the obstruction-free technique by example with a few simple data structures. Although the technique is easier to program than lock-free or wait-free objects, there is still a significant complexity overhead.

Non-blocking synchronization is able to overcome many of the problems of associated with conventional locks. Despite its many advantages, however, non-blocking techniques have not been widely adopted by the parallel programming community. The reason for its lack of popularity lies in its inherent complexity. Instead of making parallel programming easier, non-blocking synchronization actually makes it more difficult.

3.2 Software transactional memory

Several software transaction systems have been proposed. Software designs have the advantage that they require no change to the hardware and generally do not impose transaction size or length limitations. Software systems are generally implemented by inserting additional code by modifying the compiler. Therefore, they can also take advantage of compiler analysis and optimization. Unfortunately, software transactions generally have high performance overhead. In this section, I describe previous work on software transactional memory and the associated performance overheads.

The first design for software transactional memory was proposed by Shavit and Touitou [47]. Their system requires that all input and output locations touched by a transaction be known in advance. This restriction limits its application. Further, it performs at least 10 additional loads and 4 additional stores per memory location accessed as part of a transaction.

Rudys and Wallach [45] proposed a copy-based transaction system to allow rollback of hostile code blocks. On one processor, they show an order of magnitude slowdown for field and array accesses and a 6-23x slowdown on their benchmarks.

Herlihy, Luchango, Moss, and Scherer's scheme [21] allows transactions to touch a dynamic set of memory locations. The user needs to explicitly open every object before it can be used in a transaction. This implementation is based on object copying,

and thus has poor performance for large objects and arrays. They present a list insertion benchmark on one processor that shows a 9x slowdown over a locking scheme.

Harris and Fraser built a software transaction system on a word-oriented transactional memory abstraction [16]. Their scheme effectively makes a speculative copy of each word in the transaction and operates on that copy during the transaction. Their scheme only copies the data being accessed at a word granularity thus avoiding the problems with large objects. With this technique, they are able to decrease the overhead significantly. Single-processor overhead, however, is still 2x over locks for some specially-coded microbenchmarks.

Unlike non-blocking synchronization, software transactional memory is implemented in the compiler. Therefore, the synchronization details are abstracted away from the programming making software transactions easy to use.

Unfortunately, all of these software designs have high performance overhead. The fastest system is still 2x slower than conventional locks under ideal situations. The high overhead is not surprising since all the bookkeeping is done using normal software techniques. The large amount of bookkeeping simply precludes an efficient implementation.

All of these software designs generally have more reasonable performance overheads when running on many processors. With a lot of processors, and the appropriate application, the optimistic nature of transactions can overcome the high overhead. Typically the number of processors required for acceptable performance is much more than in modern small SMP systems. Thus, software transactional memory is impractical for general use in real systems.

3.3 Hardware transactional memory

Hardware can be used to accelerate transactional bookkeeping. The technique has been explored in many different contexts. Hardware transactional memory was first proposed as a cache and cache-coherency mechanism to facilitate lock-free synchronization. A similar technique was independently studied as part of work on thread-

level speculation and speculative locks. Recently, there has also been a transactional memory system developed at Stanford University. Unfortunately, all of these systems impose limitations that restrict their practical use. In this section, I describe in more detail these hardware systems, their advantages, and their disadvantages.

Hardware transactional memory was first proposed by Herlihy and Moss [22, 23] and Knight [26, 27]. Herlihy and Moss coined the term *transactional memory* in the context of lock-free synchronization. I will call these systems HTM (hardware transactional memory). HTM exposes the transaction primitive in the ISA and uses a hardware buffer such as a cache to store speculative data. It also uses the cache-coherency mechanism to detect conflicting transactions. By using the cache and cache-coherency mechanisms, HTM provides the transaction primitive with low performance overhead.

In HTM, every transactional load or store operation brings the target memory block into the hardware buffer in the same way a block is brought into the cache. All speculative modifications are made to the data in the hardware buffer without affecting the previous consistent value in main memory. The hardware buffer is snooped on each incoming cache intervention message. If the incoming intervention hits a speculative line before the transaction commits, a conflicting access from another processor has been made. When this conflict is detected, the running transaction is aborted. Since the previous consistent values remain in main memory, the transaction can be aborted by simply invaliding all speculative lines in the hardware buffer. All future requests for these blocks will miss and require a load from main memory as in a silent drop. This mechanism effectively rolls back all speculative changes made by the transaction. When the transaction commits, all speculative lines in the hardware buffer are changed to non-speculative. This change effectively commits the speculative changes since non-speculative data is globally visible through the cache-coherency mechanism. Using the cache and cache-coherency mechanism this way allows execution of transactions with negligible performance overhead.

Although HTM was able to maintain low performance overhead, it imposed awkward size and length limitations on transactions as discussed in Chapter 2. These

limitations prevent HTM from being useful in practice. The general cache-based approach is nevertheless valuable because of the low overhead. UTM uses this approach to achieve good common case performance.

The general notion of a transaction that HTM advocates is not shared by all previous hardware designs. Some designs have described the notion of a transaction as an extension to load-linked/store-conditional [25] and other complex atomic instructions. In fact, CISC machines such as the VAX have complex atomic instructions such as enqueue and dequeue [6]. These approaches limit the use of transactional memory to only a few specific applications.

Using the cache and cache-coherency protocol to store speculative data and detect conflicts was not only studied in the context of transactions. The notion was also explored independently in work on thread-level speculation (TLS). TLS has been investigated separately as part of the Multiscalar [49] project, the Hydra [42] project, the Stampede [53] project, and by Krishnan and Torrellas [29]. TLS executes interdependent threads speculatively, often out of order, but maintains the appearance of in-order execution at completion. The goal is to achieve better performance by speculating past false dependencies. For example, several iterations of a single-threaded loop can execute concurrently without losing the appearance of in-order execution in the absence of true inter-loop dependencies.

In TLS, there is one non-speculative thread and many speculative threads. All speculative threads can be aborted by the non-speculative thread but the non-speculative thread cannot be aborted. Speculative threads store speculative data in a hardware buffer such as a cache. The cache and cache-coherency mechanism detect conflicts and provide abort and rollback functionality as in HTM. Each speculative thread contains a timestamp. Once a non-speculative thread commits, the speculative thread with the smallest timestamp becomes non-speculative. If the speculative buffer requires overflow, TLS simply waits until the speculative thread becomes non-speculative. Then the hardware buffer can be flushed into main memory since the thread is no longer speculative.

The speculative nature of TLS is similar to that of transactions. Therefore, the

mechanisms necessary for TLS are similar to those used in HTM systems. TLS and transactions have different goals however. The goal of TLS is to increase performance and to automate code parallelization. On the other hand, the goal of transactions is to achieve atomicity without the problems associated with locks. TLS does not address the atomicity problem at all.

Another approach was proposed by Rajwar and Goodman called Speculative Lock Elision and Transactional Lock Removal (SLE and TLR) [43,44]. SLE/TLR is based on speculating past conventional locks. Similar schemes have also been developed by Martinez [36] and Rundberg [46]. SLE/TLR dynamically identifies locks and speculatively executes past them using the cache and cache-coherency mechanisms in a similar way as HTM. Since the programmer still uses locks as the underlying atomicity primitive, programs need not be recompiled to take advantage of SLE/TLR. In addition, if speculative execution cannot proceed for any reason (such as buffer overflow or context-switch), SLE/TLR falls back to locks to guarantee forward-progress. Although this technique has many of the performance advantages of a true transactional memory system, the codes must still obey a deadlock-free locking protocol. Thus they do not make programming easier.

Lastly, and most recently, a hardware design called Transactional memory Coherence and Consistency (TCC) [15] was developed at Stanford University. TCC takes a slightly different approach to transactions. TCC advocates that transactions be used everywhere. In TCC, the basic unit of work is a transaction. In contrast, in conventional systems, the basic unit is a memory operation. TCC proposes an entirely new cache-coherency and memory consistency model. In TCC, all transactions are executed speculatively in the cache but there is no conventional cache-coherency. Instead, at the end of each transaction, all speculative updates are broadcast to main memory and all other processors. Conflicts are detected by each processor upon receiving the broadcast.

TCC also overcomes any hardware size limitations with the broadcast mechanism. When a transaction requires more room than the cache, TCC simply starts broadcasting updates immediately as they are executed. The broadcasting processors does

not release the bus until the entire transaction completes. Since the processor holds the bus for the remainder of the transaction, it cannot be aborted.

Although TCC can overcome the hardware size limitation, it does not provide a mechanism that allows transactions to span context-switches. TCC does, however, provide a *pseudo-overflow* mechanism that allows non-atomic code blocks to broadcast updates before the end of the block. Pseudo-overflow can be used to prematurely end a non-atomic code block in the event of a context-switch or other exception. Pseudo-overflows, however, cannot be used in atomic transactions since it would expose inconsistent data. Therefore, like HTM, TCC must abort all atomic transactions on a context-switch.

The goal of TCC is to provide the transaction primitive while reducing the hardware complexity associated with conventional cache-coherency and memory consistency models. Since processors only broadcast on transaction commit, the TCC hardware need not support small low-latency messages required in conventional shared-memory systems. Unfortunately, although the hardware may be conceptually simpler, there are other hardware tradeoffs associated with the simplicity. For example, performing a global broadcast consumes more power than individual point-to-point transfers. Also, broadcasts are inherently not scalable. Thus TCC may only be practical for small SMP systems and not well suited for large scale multiprocessor machines.

In TLS, SLE/TLR, and TCC, transactions are not limited by the size of a hardware buffer. In all of these schemes, however, the size limitation is overcome by allowing the overflowing transaction to become non-speculative. In TLS, the oversized thread simply waits until its timestamp is the smallest in the system. In SLE/TLR, the system simply falls back to locks. In TCC, the oversized thread simply locks the broadcast bus until transaction commit. Although these approaches solve the problem, they effectively halt the rest of the system. Since the oversized transaction is non-speculative, it cannot be aborted. Therefore, all other speculative threads cannot commit until the overflowing transaction finishes. Such a mechanism can drastically decrease performance.

All previous hardware designs have one clear advantage over all software designs. Hardware is simply faster than software and thus the hardware systems have much lower performance overhead than software systems. Unfortunately, all previous hardware designs are limited in one way or another that prevents them from being truly practical systems.

Chapter 4

The UTM Design

UTM is a hardware transactional memory design where the transaction size is not bounded by a specialized hardware buffer such as a cache. As in previous hardware transactional memory systems, UTM uses the cache to store speculative state and uses the cache coherency protocol to detect conflicting transactions. Unlike previous hardware systems, UTM allows the speculative state to overflow from the cache into main memory, thereby allowing the transaction to grow beyond the size limitation of the cache. The clean semantics of UTM allow nested transaction support, nontransactional instructions, immediate aborts, a processor snapshot, and context-switching support; all features not found in previous hardware transactional systems.

In this chapter, I describe these UTM features in detail. In Sections 4.1 and 4.2, I give an overview of the design and present the transaction semantics. In Sections 4.3 and 4.4, I present how UTM stores transactional data in the cache and main memory with only minor hardware modifications. In Sections 4.5 and 4.6, I describe how UTM detects conflicts and rolls back register state when one is detected. In Section 4.7, I describe how UTM handles nontransactional instructions to support logging and debugging. In Section 4.8, I outline UTM's context-switch mechanism that leverages the overflow hardware.

4.1 Design overview

The goal in UTM is to achieve unbounded transaction with low performance and implementation overhead. This goal is reflected throughout the UTM design. UTM only requires changes to the cache and processor core. The network, cache-coherency protocol, and directory controllers are not modified. UTM uses the cache, as in previous designs, to achieve low performance overhead in the common case. UTM, however, is a significant extension of the previous hardware transaction designs. In this section, I present an overview of the UTM design focusing on its advances over previous hardware transaction designs.

Like the previous designs by Herlihy and Moss [22, 23] and Knight [26, 27], UTM primarily uses the cache to store speculative transactional data. Unlike their designs, UTM allows transactional data to overflow from the cache into a hash table in main memory. Therefore, UTM does not suffer from any size limitations. UTM augments the processor cache with less than 2 bits per cache line to keep track of transactional data and overflows.

At the ISA level, UTM differs from previous hardware designs since UTM supports transaction nesting and nontransactional instructions. UTM allows transactions to be nested by subsuming all nested transactions into the outermost transaction. This feature is achieved by simply adding a hardware counter to track the nesting depth. UTM also supports nontransactional memory instructions for debugging and logging during transaction execution.

The semantics of UTM also differ slightly from previous designs. UTM supports immediate aborts as opposed to a branch-on-abort type instruction as in the Herlihy and Moss design. UTM transactions are aborted immediately once a conflict is detected. The UTM hardware immediately rolls back all changes to the memory system and the processor's register state. There is a register snapshot mechanism that records the architectural state of the processor at the start of each transaction. The UTM snapshot mechanism uses much of the branch prediction hardware already in modern microprocessors. If the transaction commits, the snapshot is discarded.

Otherwise, when the transaction aborts, the snapshot is restored and the processor jumps immediately to the abort handler much like jumping to an exception handler.

UTM also supports context-switches during transaction execution. The running transaction is suspended on a context-switch. All transactional data in the cache is pushed out to a data structure in main memory. When the thread is switched back in, the transaction is resumed by repopulating the cache with the transactional data.

UTM is designed for an out-of-order superscalar processor in a multiprocessor environment where cache-coherency is handled using a directory-based protocol. The simulation of UTM presented in this thesis uses the MIPS R10000 [37, 55] processor and a non-uniform memory access (NUMA) network similar to the SGI Origin 2000/3000 [34] as the baseline. The design, however, is flexible enough to be adopted in any out-of-order superscalar processor that uses an invalidation-based cache-coherency protocol.

4.2 ISA modifications

UTM provides the unbounded transaction primitive to the programmer through the processor ISA. UTM requires only minor changes to the ISA to accommodate easy integration into existing systems. The minor modifications, however, allow nested transactions, immediate aborts, and nontransactional instructions.

In this section, I describe the UTM semantics and the required ISA modifications.

UTM adds the following basic instructions to the MIPS instruction set [38]:

`xBEGIN pc`: Marks the start of a transaction. The `pc` argument is the PC-relative address of the abort handler.

`xEND`: Marks the end of a transaction.

All instructions executed between the `xBEGIN` and `xEND` are considered part of the transaction. Semantically, the `xBEGIN` instruction can be viewed as a branch that jumps to the abort handler (`pc`) if the transaction cannot run atomically. Otherwise,

execution falls through to the first instruction in the transaction and always runs to the end of the transaction.

UTM also requires other additions to the processor ISA. This section only focuses on the most important instructions xBEGIN and xEND. The other instructions will be described as needed in other sections.

UTM supports properly nested transactions by subsuming all nested transactions into the outermost transaction. The xBEGIN and xEND instructions forming the inner nested transactions are effectively ignored. Only properly nested transactions are supported so an exception is signaled if improper nesting is detected.

Normal nontransactional loads and stores that occur outside transaction boundaries are treated as single-instruction transactions that cannot be aborted. Therefore, normal nontransactional memory operations always complete, even if doing so aborts a conflicting transaction.

UTM also supports nontransactional memory operations within a running transaction. Performing nontransactional operations within a transaction is necessary for tasks such as writing debugging information or logging transaction activity. Nontransactional semantics are described in Section 4.7.

Design rationale

UTM semantics support transactional procedure calls naturally. If a procedure is called within a transaction, it is desirable treat the procedure as part of the transaction. The xBEGIN and xEND behavior support these semantics automatically since all instructions (even procedure calls) executed between the xBEGIN and xEND are treated as part of the transaction. Some previous hardware designs, such as Herlihy and Moss's system, used special transactional load and store instructions. As a result, two versions of every procedure must be compiler: a transactional version and a nontransactional version. The choice of which to use depends on whether the procedure is called from within a transaction. Since some codes such as legacy libraries cannot be recompiled, such a requirement complicates the integration of transactional memory into existing systems.

UTM nesting semantics also simplifies procedure calls from within a transaction. If transaction nesting was not supported, the programmer or the compiler needs to know if the called procedure actually uses transactions or not. Since this information is often difficult to obtain or even unavailable until runtime, UTM must support nested transactions. There are several ways to handle nested transactions however. The UTM approach of subsuming properly nested transactions is one of the simplest. Since UTM is not designed to provide general database-like transactions, simply subsuming nested transactions is sufficient.

The `xBEGIN` branch-like semantics are clean and easy to use. Some previous hardware designs, such as Herlihy and Moss's system, use branch-on-abort type instructions that are inserted throughout each transaction by the programmer. The branch-on-abort instruction branches to an abort handler address if the running transaction has been aborted. Such an approach, however, is difficult to use since it relies on the programmer to periodically check whether an abort has occurred. Since the memory system rolls back all transactional changes immediately after an abort, a transaction executing after the abort can read inconsistent data. Thus, the program can behave unexpectedly. For example, an infinite loop may occur before the program reaches a branch-on-abort instruction. UTM semantics avoid such situations altogether since the processor jumps directly to the abort handler immediately after the abort.

To provide sensible semantics for the immediate abort, UTM requires more hardware modifications. UTM needs to take a hardware snapshot of the processor's register state so that it can be restored on abort. The branch-on-abort approach does not require a hardware snapshot since the register state can be restored by software. Since the transaction can only enter the abort handler from one of a few branch-on-abort locations, the registers can only be in one of a few states after the abort. In theory, the abort handler software can then restore the registers.

Lastly, treating all normal nontransactional memory operations as small un-abortable transactions simplifies integration into existing systems. These semantics are natural and easy to understand since they are the same as the semantics of a normal nontransactional system. As in a normal system, memory operations not within a transaction

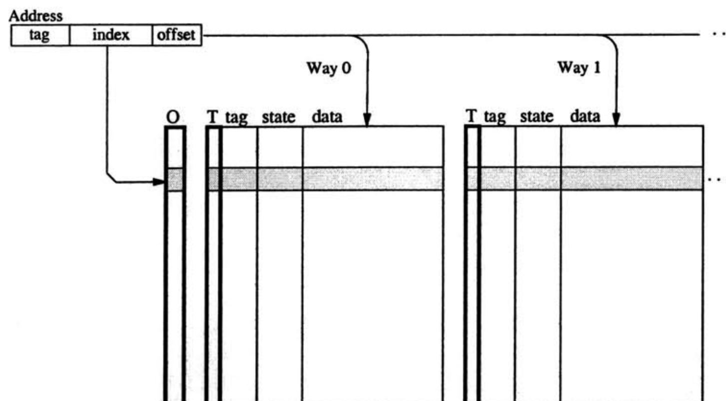


Figure 4-1: Speculative transactional storage in cache. The cache is the primary storage for speculative transactional data. Two ways of an N-way set-associative cache is shown. The T bit indicates if the cache line contains transactional data. The O bit indicates if the cache set has overflowed.

always affect the contents of main memory. All conflicting transactions are simply aborted. Therefore, transactional code can seamlessly operate alongside nontransactional code such as legacy libraries. Transactional applications can therefore be integrated into existing systems with low risk.

4.3 Transactional state in cache

UTM uses the cache as the primary storage space for speculative transactional data during transaction execution. As in previous designs, the cache allows for a straightforward implementation that has low performance overhead. In this section, I describe the necessary modifications to the cache.

While the cache holds the speculative data, the previous consistent data always remains in main memory. Speculative transactional updates are only performed on cache data. Speculative transactional data is only permitted to leave the cache after commit, thereby making all changes globally visible. Previous hardware transactional memory designs use the cache similarly.

Storing transactional data in the cache requires only minor modifications as shown in Figure 4-1. UTM assumes a set-associative cache that is common in modern processors. One transaction (T) bit per cache line is added to indicate if the cache

line contains transactional data. One overflow (O) bit per cache set is added to indicate if the cache set has overflowed. These bits represent the transactional state in the cache. This section focuses on the transaction bits. Overflows are discussed in detail in Section 4.4.

At instruction decode, memory instructions are tagged as transactional if they are decoded between the outermost `xBEGIN` and `xEND` instructions. The nesting depth is tracked by an internal hardware counter which is initialized to zero when not in a transaction. The hardware counter is incremented and decremented on `xBEGIN` and `xEND` decode respectively. If the nesting level is greater than zero when a memory instruction is decoded, the instruction is tagged as transactional.

When a transactional memory operation hits in the cache, the target line is marked transactional by setting the T bit. If it misses in the cache, the target line is brought into the cache as in a normal memory operation, and then marked transactional. If a transactional store request hits a nontransactional modified cache line, the cache writes the old data back to main memory. The write-back is required to keep the non-speculative consistent value in memory so that speculative transactional changes can be rolled-back.

UTM stores transactional state in the outermost largest level of cache. In the MIPS R10000, transactional data is stored in L2 since it is the largest outermost cache. The L1 cache obeys the inclusion property so L1 data is always a subset of L2 data. UTM's transactional state obeys the inclusion property as well. Therefore, the L1 cache also contains transaction bits but they are simply a subset of those in the L2 cache.

The MIPS R10000 processor, like most speculative out-of-order processors, often speculatively issue memory load requests. For example, a load after an unresolved predicted branch is issued to the cache once the address is calculated. At that time, however, the load instruction is still speculative and thus may not even graduate from the processor pipeline. For example, if the branch is mispredicted, the load is flushed along with the rest of the instruction pipeline. Thus, some of the data brought into the cache may not be associated with any of the perceived graduated instructions.

Although the processor issues speculative load requests, UTM only stores transactional state for graduated instructions. Speculative loads cannot change the transactional state of the cache. Therefore, on a speculative load, the data is returned as normal but the transactional state is not affected. An additional request is sent to the cache when the transactional load graduates. When the non-speculative request hits in the cache, the target cache line is marked transactional. Although the additional request adds cache traffic, the effect is minimal since no data is returned.

An additional store request is not necessary since out-of-order processors like the MIPS R10000 generally do not speculatively issue store requests. Instead, store requests are only issued at instruction graduation when the instruction is no longer speculative. Therefore, when a transactional store request hits in the cache, the target cache line is marked transactional immediately. Some processors, however, issue store prefetches once the store address has been calculated. The prefetch simply brings the target line into the cache but leaves the data unchanged. Like speculative load requests, these prefetches do not affect the transactional state in the cache.

After a cache line is marked transactional, it remains transactional until the transaction commits. In the absence of overflows, simply clearing all the T bits commits the transaction. Clearing the T bits makes the once transactional cache lines visible in global memory since future cache interventions will cause the new values to be written back to memory. Transaction commit takes only one clock cycle since the T bits can be flashed cleared.

Design rationale

The cache is used as the primary storage for transactional data since it introduces low performance overhead and requires only minimal changes to existing hardware. As shown in the Herlihy and Moss design, using the cache in this manner results in negligible performance overhead. This low overhead is expected since the cache mechanism does not fundamentally affect the processor pipeline or other hardware components in the critical execution path. The code is simply executed normally except transactional operations mark the cache.

Using one bit per cache line and one bit per cache set does not add much hardware overhead. In the MIPS R10000 with 128 byte cache lines, the overhead is less than 1.3% additional bits per cache line. There is also the added complexity associated with the ability to flash clear transactional bits in one cycle. Fortunately, these operations can be achieved by using additional global bit-lines connected to the modified SRAM cells used to store the state bits.

Issuing each transactional load twice generates higher cache traffic. The second request, however, is necessary since the UTM cache cannot store transactional state for instructions before graduation. Alternatively, speculative instructions could be allowed to mark the cache. Unfortunately, this alternative requires more complicated hardware since the cache must be unmarked if the speculative instruction does not graduate. Therefore, UTM issues transactional loads twice to simplify cache modifications. Further, as shown by results in Section 5.2 the additional cache traffic should not decrease performance drastically since no return data is required.

4.4 Transactional state in main memory

UTM overflows transactional data into a hash table in main memory if the cache limit is reached. Unlike previous systems which only used the cache to store transactional data, UTM allows transactions to grow effectively without bound. Since the hash table is in main memory, UTM incurs an additional performance overhead. The performance loss, however, is reduced significantly since all operations on the hash table take constant time. In this section, I describe the overflow hash table in detail.

The hash table in main memory is set up by the operating system but maintained by hardware. From the cache-coherency point of view, the hash table extends the cache. Although using main memory in this manner can potentially be slow, the overflow hash table will only be used rarely. In the common case, transactions fit in the cache and do not require overflow. Therefore, a slight performance decrease is acceptable since it can be efficiently amortized over the fast common case.

When the cache is about to evict a transactional cache line for capacity reasons,

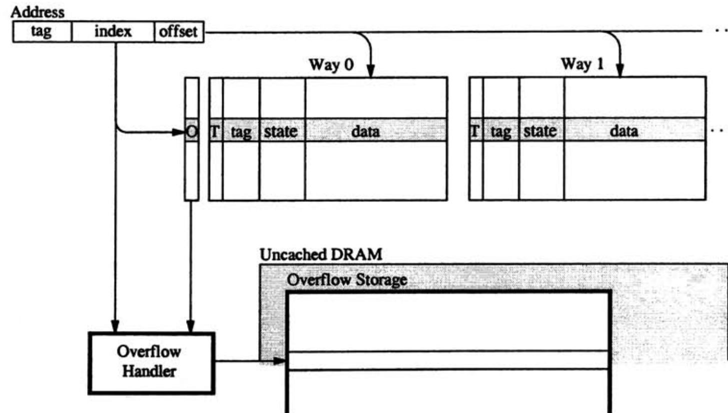


Figure 4-2: Speculative transactional storage in cache and main memory. The cache is the primary storage for speculative transactional data. When the cache is full, however, transactional data is overflowed into a data structure in uncached main memory. The overflow (O) bits indicate which cache sets have overflowed.

the overflowing cache set is marked as overflowed by setting the O bit. Then the evicted cache line is added to the overflow hash table as shown in Figure 4-2. To ensure that overflows only occur when necessary, the cache replacement policy is changed to evict nontransactional cache lines before transactional cache lines in all cases.

When a processor or network request misses in the cache but indexes into an overflowed cache set, the overflow hash table is searched for the requested line. If the line is found, it is swapped with another line in the cache according to the replacement policy, and the request is handled as a cache hit. If the requested line is not found, the request is handled as a cache miss.

The transactional data contained within the overflow hash table must be made globally visible when the transaction commits. To accomplish this task, each modified transactional line in the overflow hash table is written back to main memory. In addition, the overflow bits in the cache are also cleared.

Since overflow operations can take much longer than operations on the cache, incoming cache requests are stalled during overflow handling. Stalling incoming requests is necessary when the overflow data structure is searched and when it is being written to memory on commit. To stall network requests, UTM uses a negative-

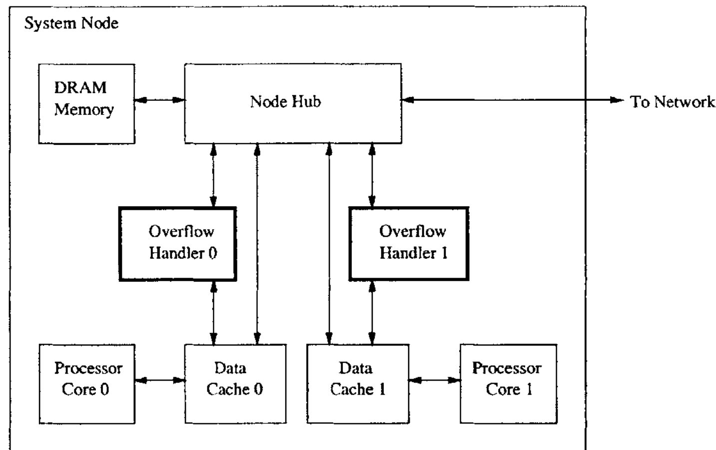


Figure 4-3: Overflow handler interface to cache and memory. There is one overflow handler per processor in the node. Only two processors are shown. The overflow handler shares the same connection to memory as the cache. Through the hub, the overflow handler has access to the local DRAM as well as the distributed memory through the network.

acknowledgement (NACK) network protocol. Incoming network requests are queued as normal but the queue is not serviced when handling overflow. When the queue is full, a NACK is sent to the requestor. Similarly, to stall processor requests, UTM simply allows the memory request queue to fill up. Once the queue is full, the processor pipeline will stall.

The hash table is maintained by the overflow handler hardware which shares the cache's connection to memory as shown in Figure 4-3. In the SGI Origin architecture, there may be several processors per node. Each processor has a cache and an overflow handler which both communicate with the hub. The hub services memory requests for local and remote memory. The overflow handler resides between the cache and the hub allowing it to directly access the entire memory address space. Since the overflow handler is after the cache, the overflow hash table resides in a region of uncached main memory. Also, since the interface is after virtual address translation, all memory addresses used by the overflow handler are physical addresses.

The UTM overflow hash table uses the low order bits of the address as an index like a direct-mapped cache as shown in Figure 4-4. Conflicts are resolved using linear probing [5, sec. 11.4]. Each element in the hash table contains all the necessary

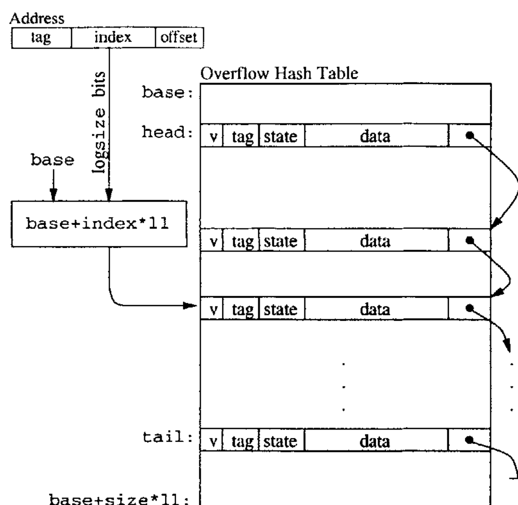


Figure 4-4: Overflow hash table function and data structure. There are four internal registers: `base`, `size`, `head`, and `tail`. Each line in the hash table has 5 fields: v, tag, state, data, and ptr. The length of each line (all 5 fields) is `ll` and is fixed based on the architecture. The linked list of elements does not necessarily only progress in one direction. It is only drawn that way for clarity.

information to reconstruct the cache line. Each element contains the address tag, cache line state, and data. In addition, to support faster traversal on transaction commit, each element also contains a valid bit and a pointer. The valid bit specifies whether the element needs to be written back to main memory on commit. The pointer indicates the next element in a linked list of all the overflowed elements. Each time a line is added to the hash table, it is added to the linked list. When the hash table is written back to main memory on commit, the overflow handler simply traverses the linked list and writes back each element in turn. When an element is swapped out of the hash table, it is simply marked invalid and not written back on commit.

The overflow handler has four internal registers. There are two visible registers: `base` and `size`. The `base` register contains a pointer to the start of the hash table data. The `size` register contains the total number of lines the hash table can hold. These registers are accessible via special instructions in the ISA. The details of these instructions will not be discussed since they can be implemented in several different ways. The operating system can change these registers when setting up the overflow

hash table. The overflow handler also has two internal registers: `head` and `tail`. These registers contain pointers to the head and tail of the linked list respectively.

Since the number of elements in the hash table is restricted by the `size` register, the hash table can fill up before the transaction commits. In such a case, the transaction must be aborted. The operating system is informed and can increase the hash table size before retrying the transaction.

Design Rationale

A hash table is used to store overflowed data since it is efficient and can be implemented easily in hardware. In the common case, an insertion or search only requires constant time. Since the hash table is in main memory, the access time is much greater than that of cache but it does not grow with the size of the transaction. The constant access time is important since the UTM design relies on the ability to efficiently amortize this high overhead over the common cache access time. If the access time were not constant, the overhead may grow so high that overall performance is decreased dramatically. In fact, UTM was first implemented with an overflow data structure with access time that grows linearly with the size of the transaction. The linear data structure led to a drastic performance decrease. That design alternative is discussed in detail in Section 6.2.

To further optimize overflow handling, the hash table elements are part of a linked list. The linked list ensures that the commit time of a transaction is linearly proportional to the number of overflowed cache lines. Hash tables do not inherently support linear time traversal of all elements. Therefore, it is necessary to augment the hash table. Without a linked list, UTM must traverse every slot in the hash table, even the empty slots. Since the hash table can be large and overflows are rare, such an approach is highly inefficient.

4.5 Conflict detection

UTM detects conflicts between transactions using the cache-coherency mechanism. As in previous designs, UTM leverages the cache-coherency protocol's invalidation mechanism to detect conflicts without any modifications to the protocol. If an incoming cache intervention hits a transactional line either in the cache or in the overflow hash table, a conflict is signaled. In this section, I describe the conflict detection mechanism.

In a modern processor, an instruction takes many cycles to reach the cache after it graduates. The target cache line, however, is effectively part of the transaction immediately after instruction graduation. Therefore, to correctly detect all conflicts, it is insufficient to check only the cache for the target line of an intervention. All request queues between the processor core and the cache must also snooped on interventions. If an intervention hits a transactional memory operation in transit to the cache, an abort is also required. Fortunately, processors such as the MIPS R10000 already perform a similar check to maintain sequential consistency.

Similarly, if a transaction commit request is in transit to the cache, the `xEND` instruction has already graduated. Therefore, the transaction can no longer be aborted. In that case, all interventions hitting a transactional cache line must be stalled until the transactional commit reaches the cache. Therefore, UTM must check all request queues for a commit request before an abort is signaled.

If the transaction aborts, all transactional cache lines are invalidated. The cache intervention causing the abort still needs a reply. Since the transactional data cannot be made globally visible, a reply is sent back indicating that the line was not found. The same reply is sent for interventions for silently dropped lines. Once the reply is received by the requestor, the requestor simply reads the line from main memory, thereby retrieving the most up-to-date consistent value. Therefore, from the global memory system, it appears as if the transaction did not occur.

Design rationale

The UTM abort policy integrates well with the cache-coherency protocol and nontransactional semantics. Normal nontransactional instructions cause a cache intervention if the target line is not in the appropriate state in the cache. If the cache line is part of a transaction, the transaction will receive the intervention and will abort. This behavior precisely matches UTM semantics since nontransactional instructions cannot be aborted.

Using the cache-coherency protocol to detect conflicts allows multiple transactions to read a single memory location without aborting one another. Each processor has the cache line in the shared state and no intervention messages are sent. Allowing multiple transactional readers is desirable since there is no inherent conflict when multiple transactions are only reading. Moreover, often a shared variable, such as the head pointer of a linked-list, is read by many processors but rarely written.

The cache-coherency protocol is used in previous hardware designs because cache intervention messages correspond exactly to transactional conflicts. To understand this relationship, consider the following three cases.

1. Processor A transactionally writes X. Then processor B reads X.

2. Processor A transactionally writes X. Then processor B writes X.

3. Processor A transactionally reads X. Then processor B writes X.

If the underlying transactional memory system can only support one transactional writer or multiple transactional readers at once, as in UTM, these three cases represent all possible conflicts that can occur. Conveniently, in all three cases, the cache-coherency protocol sends an intervention message to processor A immediately when processor B makes the memory request. In the first and second case, the intervention retrieves and invalidates the modified cache line on processor A. In the third case, the intervention invalidates the cache line on processor A. Once processor A receives the intervention, the transaction aborts. Therefore, the cache-coherency mechanism detects all possible conflicts.

This conflict detection only works for systems that support one transactional writer or multiple transactional readers at once. Such a system, however, is conservative. In theory, a transactional memory system can allow many concurrent transactional readers and writers as long as the underlying system ensures that all data is consistent. In fact, multi-version databases allow this type of concurrency. Unfortunately, the necessary bookkeeping would incur impractically high performance and implementation overheads.

4.6 Processor snapshot

The `xBEGIN` instruction behaves like a branch that either jumps to the abort handler, when a conflict occurs, or falls through to the rest of the transaction, when atomicity is guaranteed. Therefore, UTM takes a snapshot of the register state at the `xBEGIN` and restores it if the transaction aborts. The register snapshot requires only minor changes to the processor core. In this section, I describe the necessary modifications.

The register snapshot mechanism requires only minimal changes to the processor core since much of the existing out-of-order and branch-prediction hardware can be used. As shown in Figure 4-5, an additional S (save) bit vector is added to the physical register file. There is one S bit per physical register. Also, one snapshot of the S vector per rename table snapshot is added. Lastly, an additional Register Reserved List FIFO is added to store otherwise free physical registers during transaction execution.

The active S vector tracks the active physical registers. Active physical registers are those that are in the active rename table. Therefore, when a rename table entry changes at instruction decode, the newly mapped physical register is marked in the active S vector and the previous physical register is unmarked. Thus, the S vector tracks the physical registers corresponding to the architectural registers visible to the programmer.

When the outermost `xBEGIN` is decoded, snapshots of the active S vector and active rename table are taken and stored away. At this point, the transaction has not actually started since the `xBEGIN` has not graduated. The `xBEGIN` may be a specula-

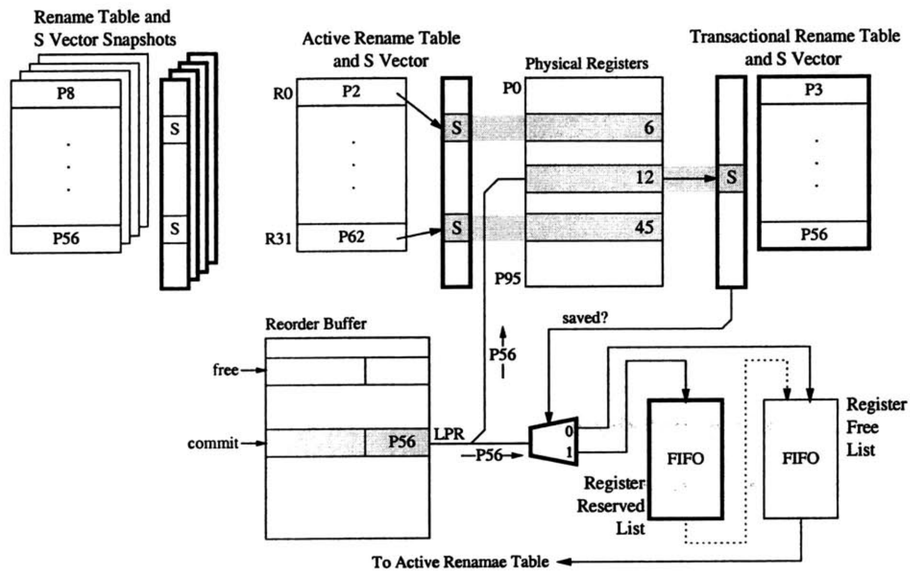


Figure 4-5: Architectural register snapshot mechanism. Physical registers that need to be saved are marked in the transactional S (save) vector. The transactional rename table is the register mapping for the saved registers. The last physical register (LPR) from the reorder buffer is normally freed. Saved physical registers, however, are not freed until after the transaction commits. Before transaction commit, otherwise free registers are added to the Register Reserved List. After transaction commit, the Register Reserved List is drained lazily into the Register Free List.

tive instruction after an unresolved predicted branch for example. All transactional memory operations are made visible only on instruction graduation. Therefore, if the `xBEGIN` is flushed from the pipeline for any reason before graduation, the associated rename table and S vector snapshot are freed as well. When the outermost `xBEGIN` graduates, however, the associated S vector and rename table snapshot are saved as transactional. The abort handler address is also set when the `xBEGIN` graduates.

After the S vector and rename table snapshot are saved as transactional, they are not freed until the transaction either commits or aborts. Before that point, the transactional S vector ensures that none of the saved registers are freed. After `xBEGIN` graduation, if a physical register marked in the transactional S vector is about to be freed, the register is added into the Register Reserved List instead of the Register Free List. If the freed register is not marked in the transactional S vector, it is simply added to the Register Free List as usual. Since new physical registers are only taken from the Register Free List, saved registers can not be overwritten. Therefore, this mechanism effectively snapshots the architectural register state.

When an abort occurs, the transactional rename table and S vector are restored as active. Since the saved physical registers have not been overwritten, restoring the rename table effectively restores the saved architectural register state. After the registers are restored, the Register Reserved List is cleared. The Register Free List is restored by rolling back the reorder buffer as on an exception.

The transaction commits if the `xEND` instruction reaches instruction graduation before an abort occurs. When the `xEND` graduates, a commit request is sent to the cache instructing it to commit the transactional state. Then the transactional S vector and rename table are released. Lastly, the state of the Register Reserved List is changed so that it can be lazily drained into the Register Free List.

To drain the Register Reserved List lazily, some additions are made to the FIFO. A bit is added to each entry of the Register Reserved List to indicate whether the entry can be freed. When `xEND` graduates, all transactional entries in the Register Reserved List are set to be freed. In the MIPS R10000 processors, 4 instructions can graduate per cycle. Therefore, the Register Free List has 4 write ports. In most

cycles, however, less than 4 instructions actually graduate, leaving a few free write ports. Free registers are lazily drained into the Register Reserved List using these free writes ports on each cycle so that performance will not be affected.

In theory, the additions mentioned thus far are sufficient to support register snapshots in the processor core. In practice, however, two additional modifications are necessary to sustain high performance during transaction execution. Firstly, since the rename table snapshots are normally used for branch prediction, each time an `xBEGIN` is decoded, one less branch can be predicted. The number of predicted branches and decoded `xBEGIN`s in flight must sum to less than the total number of snapshots. Therefore, the depth of speculation may be reduced significantly if additional snapshots are not added. Secondly, since the physical register file is used for the transactional snapshot, the out-of-order renaming mechanism has fewer physical registers to use during transaction execution. Therefore, the amount of register renaming may be reduced if more physical registers are not added.

To solve these two issues in UTM, an additional rename table snapshot is added, and an additional 32 integer and 32 floating point physical registers are added. One rename table snapshot was added to keep the depth of branch prediction the same as before while accommodating the outermost `xBEGIN` instruction. The additional registers are added to accommodate the 32 integer and 32 floating point architectural registers in the MIPS ISA.

4.7 Nontransactional instructions

Nontransactional instructions are necessary within a transaction to accomplish uncommon but necessary tasks such as logging or debugging. Since all transactional memory updates are unrolled on an abort, allowing information to *escape* an aborted transaction is only possible with nontransactional instructions. Fortunately, nontransactional instructions can be implemented with negligible hardware change. The resulting semantics, however, are awkward since it exposes the cache line size. In this section, I describe nontransactional semantics and its implementation. I argue that

```
xBEGIN;
  Op1();
  AppendToLog("Finished Op1");
  Op2();
  AppendToLog("Finished Op2");
  Op3();
xEND;
```

Figure 4-6: A nontransactional logging example. After each operation in the transaction, the transaction writes a progress indicator to a log data structure.

the awkward semantics are acceptable since nontransactional instructions are rare.

In most cases, all memory operations within a transaction are transactional. In some special cases, however, nontransactional memory operations are necessary within a transaction. Nontransactional instructions are memory operations that *escape* the transaction and affect the global memory system immediately whether the transaction aborts or not. Such nontransactional memory operations are necessary for tasks such as logging or debugging during transaction execution. For example, it may be desirable to log the progress of a long transaction for debugging or performance-tuning purposes, as shown in Figure 4-6. Without the ability to execute nontransactional operations, performing such activities would not be possible.

In UTM, the approach to nontransactional instructions is similar to that of transactional instructions. The following additional instructions are added to the ISA:

`nxBEGIN`: Marks the start of a nontransactional block.

`nxEND`: Marks the end of a nontransactional block.

All instructions executed between the `nxBEGIN` and `nxEND` are nontransactional. Since the underlying mechanism uses cache lines to store transactional state, the granularity of the nontransactional memory operations is limited to the cache line. Therefore, nontransactional memory operations cannot write to cache lines previously accessed by the running transaction since they may contain uncommitted data. If a nontransactional store hits a transactional cache line, the transaction is aborted with an exception indicating the problem.

Nontransactional operations can read transactional data. Reading transactional data is necessary to transfer data from the transaction to the nontransactional region. For example, the nontransactional region may need to log the contents of a transactional data structure for debugging purposes. In that case, the nontransactional code must read memory written by the uncommitted transaction. For this reason, when a nontransactional load hits a transactional cache line, the data is returned normally leaving the cache state unchanged.

In all other cases, nontransactional memory operations are treated exactly like normal memory operations not executing within a transaction. Nontransactional instructions flow through the processor pipeline normally, and do not mark the cache in any way.

Design rationale

UTM nontransactional semantics support procedure calls in the same way as `xBEGIN` and `xEND`. The semantics also require fewer modifications to the instruction set than adding nontransactional versions of every memory operation.

Unfortunately, nontransactional instructions have several awkward restrictions since they operate at a cache line granularity. The programmer must ensure that nontransactional writes do not hit transactional cache lines. This restriction complicates portable programming significantly but allows a simple implementation. No changes to the processor, cache, and memory system are necessary.

Although the semantics are awkward, UTM's nontransactional semantics seem to be sufficient since nontransactional instructions are expected to be rare. Better semantics can be achieved by modifying the cache-coherency protocol, network messages, and directory controllers as shown in Section 6.4. The high implementation overhead, however, does not seem justified since nontransactional instructions are used only for rare tasks such as debugging and logging.

4.8 Context-switch support

Since large transactions inherently take longer to run, they are more likely to be interrupted by context switches. To avoid unnecessary aborts, UTM supports context-switches during transaction execution by suspending the running transaction. When the thread containing the transaction is switched back, UTM resumes the transaction. Transactions are suspended and resumed using the overflow mechanism already implemented. Therefore, context-switch support requires only minor hardware modifications. Unfortunately, resulting implementation has high performance overhead. In this section, I outline UTM's context-switch support and its implementation. I argue that the high overhead is acceptable since context-switches are rare. In case where they are not rare, however, I also present some optimizations which require more hardware modification.

The UTM context-switch mechanism leverages the overflow hash table mechanism. When a transaction is suspended, all the transactional data in the cache is pushed into the overflow hash table. All transactional data is cleared and invalidated in the cache, as on an abort. Then, the thread is switched out and a new thread is switched in.

The new thread may also run transactions. These new transactions have their own overflow hash tables. Since the suspended transactional data was flushed from the cache, the new transaction can use the cache as usual to store transactional data. Since the suspended transaction has not committed, another processor or even the running thread may conflict with it. To detect these conflicts, UTM performs additional checks of the suspended transactions. All checks are performed by the overflow handler.

On a context-switch, UTM writes all the data necessary to restore the transaction to a *suspend* data structure as shown in Figure 4-7. The suspend data structure contains the overflow hash table. The cache flushes all transactional data into the overflow hash table of the suspend data structure. The overflow handler hardware performs this flush without any software support. As in normal overflow handling,

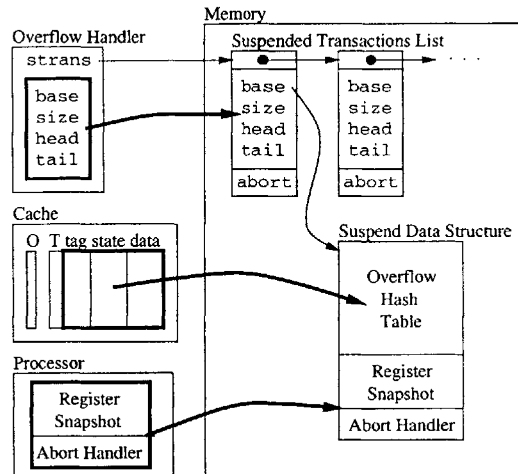


Figure 4-7: Context-switch mechanism. The overflow handler contains a register `strans` that points to the suspended transactions list containing the addresses of all the suspend data structures. On a context-switch, the overflow handler's internal register values are inserted into the suspended transaction linked list. Then, all transactional data in the cache is pushed into the suspend data structure. Lastly, the register snapshot and abort handler address are also written to the suspend data structure. Only one way of the set associative cache is shown.

the cache stalls all external cache interventions until the entire transaction has been pushed out.

To enable transaction restore, the overflow handler also writes the register snapshot and the address of the abort handler to the suspend data structure. Special instructions in the ISA accomplish this functionality. These instructions give the programmer access to the register snapshot and abort handler address. These instructions are not described in detail since they can be implemented in several ways and their performance is not crucial to the design.

When UTM suspends a transaction, the cache clears all the transactional bits and invalidates all the exclusive transactional lines as on an abort. Then, UTM adds the address of the suspend data structure to the processor's suspended transactions list. The suspended transactions list is a doubly linked list in main memory and is maintained by hardware. UTM writes the overflow handler's registers to the linked list entry and stores the head of the list in the `strans` register.

The overflow handler hardware searches all suspended transactions by traversing

the suspended transaction linked list, and searching each transaction's hash table for the requested line in turn. Transactions marked aborted in the suspended transactions list are not searched. The suspended transactions are searched on each potentially conflicting request from the processor or the network. If UTM finds a suspended transaction in a conflicting state, it aborts the suspended transaction by simply marking the `abort` flag in the linked list entry. UTM does not check suspended transactions that are marked aborted.

A cache intervention causes a suspended transaction search only when the intervention hits a shared cache line or when it *misses the running transaction*. A request *misses the running transaction* when it misses in the cache and is not found in the running transaction's overflow hash table. If UTM finds the requested line in a suspended transaction, it aborts the suspended transaction. The abort is necessary since the cache intervention indicates a conflict as in normal transactions.

A processor request causes a suspended transaction search only when the request misses the running transaction or requires a cache state upgrade. The search is necessary to detect conflicts between the running thread and the suspended transactions. If the memory request is a read and a suspended transaction has the line in the modified state, UTM aborts the suspended transaction. If the memory request is a write and a suspended transaction has the line, UTM aborts the suspended transaction.

When a suspended transaction resumes, the overflow handler first checks if it has been aborted. If the transaction is not aborted, UTM writes the register snapshot and the abort handler address back into the processor. Additional instructions in the ISA accomplish this functionality. Again, the details of these instructions are not discussed. Then, UTM restores the overflow handler registers and the cache data. To restore the cache data, UTM traverses the hash table, writing back each transactional cache line. If a line does not fit in the cache, UTM marks the overflow bit in the target cache set. Lastly, UTM removes the transaction entry in the suspended transactions list and the processor restores the active registers using the normal context-switch mechanism. The processor then continues execution of the transaction.

If the resumed suspended transaction is aborted, the processor simply restores the

register snapshot from the suspend data structure and jumps to the abort handler. UTM discards the overflow hash table but leaves the cache unaffected. Then the processor simply executes the abort handler code.

Design rationale

The UTM context-switch mechanism suffers from two main sources of performance overhead. A performance loss is incurred when UTM searches the suspended transactions. Another performance loss is incurred when UTM suspends or resumes transaction. Fortunately, UTM can search the suspended transactions in the background. Also, transaction suspend and resume are infrequent. Therefore, the context-switching mechanism can have low overall performance overhead.

UTM can search the suspended transactions in the background since the search result is not immediately needed. A suspended transaction search can only result in aborted suspended transactions. Therefore, for both cache interventions and processor requests, a reply can be made before the suspended transaction search completes. For example, if a cache intervention misses in the cache and triggers a suspended transaction search, the processor always replies as if the line was silently dropped. This reply is sent back even if the line is eventually found in a suspended transaction.

Performing suspended transaction searches in the background requires some additional hardware. The overflow handler needs the ability to queue up requests and perhaps even perform multiple outstanding searches. The background search, however, will use some memory bandwidth. Since the search can be performed lazily with lower priority than normal memory requests, the bandwidth overhead can be reduced. Since the hardware needed is not overly complicated, it is reasonable to assume that suspended transaction searches will not increase memory request latency significantly.

The high performance overhead associated with suspending and resuming transactions, however, cannot be hidden in the background. For both suspend and resume, the overflow handler needs to access every transactional cache line in the cache. In large transactions, which are most likely to be interrupted by a context-switch, most

or all of the cache may be transactional. Therefore, in a modern processor with a cache size of several megabytes, the performance overhead may be high. Fortunately, most transactions are short so suspended transactions are unlikely.

If necessary, however, the high overhead associated with suspending and resuming transactions can be alleviated with some additional cache modifications. In the remainder of this section, I describe some optimizations that speedup transaction suspend and resume.

Optimizing context-switches

The overhead associated with suspending and resuming a transaction can be reduced with two optimizations. In the first optimization, the transactional data repopulates the cache lazily. In the second optimization, the cache tracks multiple transactions.

The first optimization repopulates the cache lazily. On transaction resume, instead of writing every transactional cache line back into the cache, the cache brings the lines in only when necessary. This lazy repopulation can be achieved using the overflow mechanism already in place. When a transaction is resumed, it marks the overflow bits for every cache set originally containing transactional data. The actual data is not actually brought into the cache at that point. Instead, all the data remains in the overflow hash table until the processor requests it. At that time, the request misses in the cache but hits a set with the overflow bit marked. Then, the normal overflow mechanism searches the overflow hash table and brings the target line into the cache.

Lazy cache repopulation does not incur an immediate high performance overhead every time a transaction is resumed. Instead, the overhead is incurred only when necessary. After normal context-switches, most memory operations miss in the cache, resulting in a period of time where the cache is warmed up. During the cache warm up period, the cache retrieves data from main memory on every cache miss. A similar warm up period exists for lazy cache repopulation.

Using the overflow bit for lazy cache repopulation is convenient but has one drawback. Cache sets that did not originally overflow are marked as overflowed after the thread is switched back. As a result, unnecessary hash table searches may be invoked.

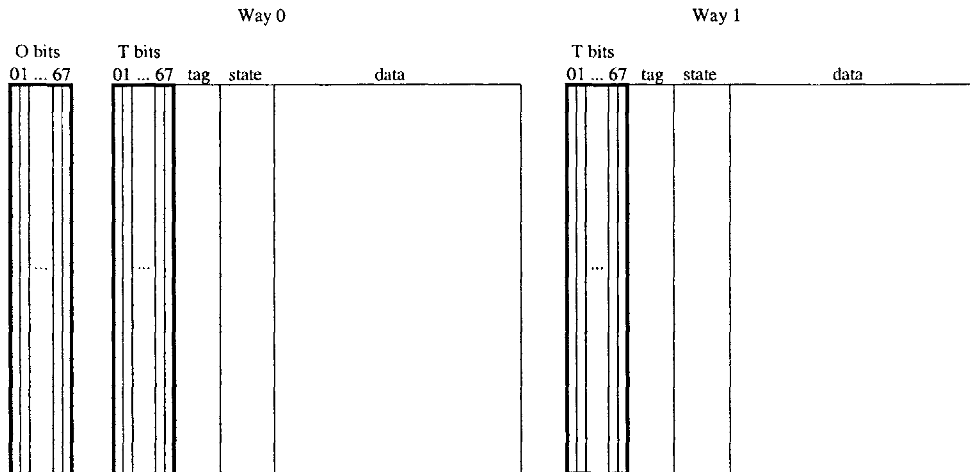


Figure 4-8: Cache modifications for optimized context-switches. An addition overflow (O) bit and transaction (T) bit is added per transaction. A 2-way set-associative cache is shown supporting 8 transactions.

Fortunately, context-switches are likely to only occur during large transactions, in which most or all cache sets have already overflowed.

The second optimization allows multiple transactions to reside in the cache at the same time. With this optimization, transactional data need not be pushed out to the overflow hash table each time a transaction is suspended. Support for multiple transactions in the cache can be accomplished by making some simple modifications to the cache. As shown in Figure 4-8, instead of only one transaction bit and one overflow bit, there is one bit per transaction. For example, the cache can have 8 transaction bits per line and 8 overflow bits per set. These additional bits distinguish which transaction each line and set belongs to. The processor simply tracks which transaction is currently running; all others are suspended. The overflow handler stores the necessary information for all 8 overflow hash tables. If a transactional cache line needs to be evicted for capacity reasons, the overflow handler simply adds the cache line into the appropriate hash table.

When more than 8 transactions exist, additional transactions are pushed into memory using the mechanism described earlier. Supporting a constant number of transactions with low overhead, however, covers most common cases.

When all suspended transactions fit in cache, the optimized mechanism suspends a

transaction by simply changing a hardware register that tracks the bit corresponding to the running transaction. Moreover, suspended transaction searches are faster as well, since the hash table search is not always necessary. If a cache intervention hits a transactional cache line belonging to a suspended transaction, the suspended transaction can be aborted immediately. Hash table searches are only necessary when an intervention misses on a suspended overflowed cache set.

Chapter 5

Evaluation

To quantitatively evaluate UTM, I implemented it in a detailed software simulator. The standard SPECjvm98 benchmark suite was converted to use UTM transactions to evaluate performance overheads and transaction characteristics. Also, some microbenchmarks were written to evaluate multiprocessor program behavior. The results show that, as expected, the overall performance overhead is low. The results confirm that transactions are indeed small and short in the common case. Therefore, the higher overhead associated with overflows is efficiently amortized over the fast common case.

In this chapter, I present the evaluation in more detail. In Section 5.1, I describe the evaluation environment and the software simulator. In Section 5.2, I show that UTM indeed has low overall performance overhead. In Section 5.3, I show that most transactions are small and short, confirming the fundamental assumption behind UTM. In Section 5.4, I evaluate the affect of overflowing transactions on memory latency. In Section 5.5, I show that UTM pipeline modifications indeed increase performance for transactional code. In Section 5.6, I use the microbenchmark to describe the behavior of multiprocessor transactional programs. I show that small and short transactions run much faster than locks and can indeed exploit concurrency hidden by locks.

UVSIM parameter	Value
Issue/graduate width	4 instructions
Reorder buffer size	48 instructions
Rename snapshots	5
Functional units	2 ALUs, 2 FPUs, 1 LSU
Int phy registers	96
Fp phy registers	96
L1 Icache	32KB, 4-way, 64 byte lines
L1 Dcache	32KB, 4-way, 64 byte lines
L1 latency	2 processor cycles
L2 Dcache	1MB, 4-way, 128 byte lines
L2 latency	8 processor cycles
RAM	SDRAM, 1/5 processor freq.
Network hop latency	10 processor cycles
Overflow hash table	128MB (1M entries)

Figure 5-1: UVSIM simulation parameters.

5.1 Evaluation environment

UTM was implemented in a detailed software simulator based on RSIM [40]. The simulator models a multiprocessor system using a directory based cache-coherency protocol. In this section, I describe the UVSIM simulation environment and how benchmark applications were compiled.

UTM was implemented in the UVSIM [57] software simulator developed at the University of Utah. UVSIM is based on URSIM [56], and URSIM is, in turn, based on RSIM [40] from Rice University. UVSIM is an event-driven simulator modeling MIPS R10000 processors in a multiprocessor configuration similar to the SGI Origin 2000/3000. All simulated processors share a single memory address space but the memory is distributed among the processors and accessed through a network. Cache-coherency in the simulated system uses a MESI invalidation-based directory protocol. The processor, bus, DRAM, and network hub are modeled in high detail and are accurate [56]. The hard disk is not modeled and the network does not model traffic or contention. The simulator parameters are shown in Figure 5-1.

The UVSIM simulator has detailed timing models but it is not a full system simulator. Therefore, it cannot run an off-the-shelf operating system such as IRIX

or Linux. Instead, UVSIM runs custom microkernel operating system that handles common system calls and other essentials such as TLB refills. Many features, however, such as dynamically linked libraries and multiprocess scheduling, are not present. Therefore, all application binaries were statically linked at compile time. The lack of context-switching support prevented any evaluation of UTM's context-switching mechanism. The UTM context-switching mechanism was not implemented in UVSIM.

Since there are no transactional benchmark suites, the SPECjvm98 [51] suite was used for this evaluation. The SPECjvm98 suite contains Java [12] applications that were compiled to use UTM transactions. Although the SPECjvm98 benchmarks are largely single-threaded, they use the thread-safe Java standard libraries that contain synchronized code. Java `synchronized` blocks were transformed into atomic blocks [7, 16] that can be implemented using either conventional locks or transactions. Technically, such a transformation changes the program semantics slightly. The effect, however, tends to be consistent with the programmer's original intent [7].

The program transformation was performed with the FLEX [8] compiler infrastructure in an IRIX [24] 6.5 environment. FLEX is a modified Java compiler which has various backends. For this evaluation, the `PreciseC` backend was used to generate `gcc`-compatible C files. Version 0.06 of the GNU Classpath [9] Java standard libraries was used. `gcc` [50] version 3.3.1 was used with the highest optimization level (-O9). The compilation was done with the `-mips4` option using the `n64` ABI. Since all standard IRIX 6.5 libraries are dynamically linked, SGI provided special statically linked libraries for compiling to UVSIM. Garbage collection was turned off since only single thread execution was performed, and garbage collection can result in unpredictable performance.

Three versions of the SPECjvm98 benchmark suite were compiled: `Base`, `Locks`, and `Trans`. The `Base` version uses no synchronization. The `Locks` version uses standard Java locking to implement atomic blocks. The `Trans` version uses UTM transactions to implement atomic blocks. The `Trans` version uses method cloning to flatten transactions but the same cloning was performed for all other compiled versions so that performance improvements due to the specialization would not be

improperly attributed to transactions. The three benchmark versions were built from a common code-base using method inlining in `gcc` to remove or replace all invocations of lock and transaction entry and exit code with appropriate implementations.

All SPECjvm98 runs were performed with the small input size (1). The small input size was used mainly to decrease simulation time. Since UVSIM is a detailed simulator, larger input sizes required too much time to run. Fortunately, the results from the small input size should give a good representation of large input performance since the inputs are chosen to have similar characteristics. In fact, in some cases, such as `213_javac` and `228_jack`, the larger inputs are simply iterations of the small input.

In addition to SPECjvm98 applications, some evaluation was performed using microbenchmarks. The microbenchmarks are parallel applications written in C using OpenMP [39] parallel processing primitives. They were compiled using the MIPSpro `cc` [48] version 7.3.1.3m compiler in an IRIX 6.5 environment. Since `cc` does not support inline assembly, UTM instructions were added into the assembly source before invoking the assembler. All binaries were statically linked using the static libraries provided by SGI. Two versions of each microbenchmark were compiled: a locking version and a transactional version. For the locking version, the standard SGI IRIX `__lock_test_and_set/__lock_release` [48, chap. 11] spin-lock routines were used. These locking routines are implemented with load-linked/store-conditional instructions.

5.2 Overall performance

The SPECjvm98 benchmark suite was used to evaluate overall UTM performance. SPECjvm98 contains many applications with different characteristics. Therefore, the SPECjvm98 results give a good indication of how UTM performs under a typical workload. In this section, I describe the results which confirm that UTM indeed has low overhead and, in fact, is faster than using locks in all cases.

The three versions of the benchmark were run on a single simulated processor to evaluate single-thread overhead associated with UTM transactions. The results are

Benchmark	Base time (cycles)	Locks time	Trans time	Time in trans	Time in overflow
		(% of Base time)		(% of Trans time)	
200_check	8.1M	124.0%	101.0%	32.5%	0.0085%
202_jess	75.0M	140.9%	108.0%	59.4%	0.0072%
209_db	11.8M	142.4%	105.2%	54.0%	0%
213_javac	30.7M	169.9%	114.2%	84.2%	10.5014%
222_mpegaudio	99.0M	100.3%	99.6%	0.8%	0%
228_jack	261.4M	175.3%	104.3%	32.1%	0.0056%

Figure 5-2: SPECjvm98 performance. Simulation was run on 1 simulated processor in UVSIM. The *Time in trans* and *Time in overflow* are the times spent actually running a transaction and handling overflows respectively.

shown in Figure 5-2. The `Locks` runtime is as high as 1.75x over the `Base` version for `228_jack`. On the other hand, the `Trans` runtime is less than 15% over the `Base` in all cases. Thus, performance overhead is indeed low overall. In fact, the overhead is less than that of locks in every case.

5.3 Transaction size and length

The UTM design is based on the assumption that most transactions are small and short, so overflows are infrequent. The size and length of all transactions in SPECjvm98 were measured. In this section, I describe the results which confirm that the assumption is indeed true.

The size and length distribution of the transactions in the SPECjvm98 are shown in Figure 5-3. All applications touch less than 41 cache lines on average. In addition, 99% of all transactions touch less than 68 cache lines and are less than 11K cycles long.

Although most transactions are small, some large and long transactions do exist. The largest transaction occurs in `213_javac`, touching more than 13K cache lines and running for more than 28M cycles. The existence of such transactions is evidence that some transactions will require more space than a simple hardware cache and require the ability to span context-switches.

Benchmark	xactions	Size (cache lines)			Length (cycles)		
		average	largest	99%	average	longest	99%
200_check	6,396	6.8	100	24	422.0	21,206	1,693
202_jess	82,125	10.8	336	42	582.2	41,354	2,973
209_db	14,191	7.1	155	41	470.1	36,808	2,963
213_javac	460	40.6	13,844	66	63,808.9	28,986,949	10,050
222_mpegaudio	1,044	9.9	87	67	774.4	22,648	6,064
228_jack	707,389	4.0	226	17	123.2	33,767	1,755

Figure 5-3: .SPECjvm98 transaction size and length distribution. Simulation was run on 1 simulated processor in UVSIM. The *xactions* column is the total number of transactions. The *average* columns are the average size and length of a transaction. The *largest* and *longest* columns are the largest and longest transactions respectively. The *Size 99%* column is the transaction size that 99% of all transactions are smaller than. For example, if the value in the *Size 99%* column is 10 then 99% of all transactions is less than 10 cache lines. Similarly, the *Length 99%* column is the transaction length that 99% of all transactions are shorter than.

Benchmark	Transactions		Xops		Overflowed cache sets	
	total	overflow	total	overflow	avg.	max
200_check	6,396	2	1,073K	3	0.0488%	0.0488%
202_jess	82,103	30	18,251K	72	0.0567%	0.1460%
209_db	14,191	0	2,709K	0	0%	0%
213_javac	460	3	10,684K	14,004	33.4000%	100.0000%
222_mpegaudio	1,044	0	292K	0	0%	0%
228_jack	707,388	59	29,083K	63	0.0488%	0.0488%

Figure 5-4: SPECjvm98 transaction overflow statistics. Simulation was run on 1 simulated processor in UVSIM. The *Transactions total* and *overflow* columns are the number of total transactions and overflowed transactions respectively. The *Xops total* column is the total number of transactional memory operations (xops). The *Xops overflow* is the number of xops that invoke the overflow handler such as a miss on an overflowed cache set. The *Overflowed cache sets avg.* column is the average number of cache sets overflowed per overflowed transaction. The *Overflowed cache sets max* column is the largest number of cache sets overflowed. The number of cache sets is given as a percentage of the total number of sets.

Even in 213_javac, however, 99% of all transactions are small enough (< 66 cache lines) and short enough (< 11K cycles) to fit in a hardware cache and within a process time slice. This supposition is verified by the results in Figure 5-4 showing the number of overflowing transactions, memory operations, and cache sets. The results indicate that very few transactions require more space than the processor cache. In all cases, far less than 1% of all transactions overflow. The 228_jack application has the most overflowing transactions and it only has 59.

Similarly, few transactional operations invoke the overflow handler. In the worst case, 213_javac, only 14K (< 0.2%) out of over 10M transactional operations actually invoke the overflow handler. This result is expected since UTM only searches the overflow hash table when the memory request misses on an overflowed cache. Even though all cache sets have overflowed in 213_javac (the *Overflowed cache sets max* is 100%), the probability of a miss is still low since the structure is a cache. Therefore, the overflow search rate follows the cache miss rate which is generally low.

Since all SPECjvm98 tests were run on a single simulated processor, there were no cache interventions from other processors. Therefore, the frequency of hash table searches caused by interventions is not known. UTM only searches the overflow hash table when an intervention indexes into an overflow cache set but misses in the cache. Therefore, the *Overflowed cache sets* columns in Figure 5-4 is a good indication of the likelihood of such an intervention. In all but one case, less than 0.2% of all cache sets overflowed, indicating that searches caused by interventions are unlikely.

In 213_javac, however, every cache set overflowed. This result implies every intervention that misses in the cache requires an overflow hash table search. Such a situation can potentially decrease performance dramatically but the actual performance loss is highly dependent on the activity of other processors. It is unlikely, however, that incoming interventions will repeatedly require hash table searches. When an intervention misses the cache and UTM finds the requested line in the hash table, the transaction is immediately aborted preventing future searches. The transaction continues running only when the requested line is not found in the hash table. Missing the hash table, however, only occurs if the target line was dropped silently. Since

Benchmark	Base	Trans			
	all	all	non-overflow	overflow	commit
200_check	4.53	4.41	4.28	161.33	150
202_jess	4.37	4.69	4.68	153.72	240
209_db	4.40	4.28	4.15	-	-
213_javac	4.68	4.95	4.64	192.17	339,066
222_mpegaudio	3.61	3.62	4.58	-	-
228_jack	4.67	4.68	4.30	104.24	107

Figure 5-5: SPECjvm98 memory access latency. Simulation was run on 1 simulated processor in UVSIM. All columns are times given in average number of processor cycles for the memory system to service the memory request. The *Base all* column is the time for all memory operations with in the no-synchronization version of the application. All other columns are for the transactional version of the application. The *non-overflow* and *overflow* columns are the times for all transactional memory operations that do not overflow and those that do overflow respectively. The *commit* column is the average number of processor cycles it takes to commit an overflowed transaction.

silent drops are not permitted for transactional cache lines, repeated overflow searches from interventions are unlikely.

5.4 Memory latency overhead

Since overflows are handled in main memory, the performance overhead can be significantly higher than operations in the cache. The overflow memory latency was measured in the SPECjvm98 applications. In this section, I present the results which show that overflow latency is high but the overhead is indeed efficiently amortized over the fast common case.

To evaluate the overhead of overflowed transactional operations at a finer granularity, SPECjvm98 memory access latency was measured. Figure 5-5 shows the latency for `Base` and `Trans` versions of SPECjvm98. The latency is measured from when the memory request enters the cache pipeline to when the memory system replies. As expected, the memory access latency in the `Trans` version does not differ significantly from that of the `Base` version. For `200_check` and `209_db`, the average latency is even less than the `Base` version.

The latency for memory requests requiring overflow handling is much higher than

the average non-overflow case, since the request is not serviced until after the hash table is searched. In the `213_javac` case, average overflowing memory requests take more 40x longer than the average non-overflow case. As expected, however, this high overhead is rare, and thus is amortized over the fast non-overflow case. In `213_javac`, only 0.13% of all transactional memory operations require overflow handling (from Figure 5-4). With such a low rate, overflow handling must complete in less than 360 cycles on average to maintain an average latency of 1.1x over the non-overflow case (4.64 cycles). UTM overflow hash table searches only require 192.17 cycles on average in `213_javac`. Therefore, UTM tolerates high overflow overhead since the overhead is rarely incurred.

Transaction commit time is high especially in `213_javac`. Most of the `213_javac` application is enclosed in one large transaction which takes 1M cycles to commit. Such large commits, however, are uncommon. In `213_javac`, the transaction is so large only because it contains most of the application. Therefore, the 1M cycle commit is still efficiently amortized over the 35M cycle total runtime. The overall result is less than 3% overhead, confirming that UTM does indeed efficiently amortize the high overflow overhead over the fast common case.

5.5 Effect of pipeline modifications

UTM adds additional physical registers and an additional rename table snapshot to the processor core. The design assumes that these modifications will increase performance during transaction execution while not adversely affecting performance during normal execution. Overall SPECjvm98 runtime was measured for UTM with and without the modifications. In this section, I present the result which confirms that the modifications do indeed increase transactional performance.

The SPECjvm98 applications were run on UTM configured with two different parameter sets: an `old` parameter set and a `new` parameter set. The `new` configuration matches the parameters from Figure 5-1. The `old` configuration matches the original MIPS R10000 parameters before UTM changes, and contains 4 rename table

Benchmark	Base		Trans	
	old	new	old	new
200_check	8.0M	100.36%	8.3M	98.69%
202_jess	74.8M	100.25%	86.7M	92.76%
209_db	11.7M	100.41%	12.7M	97.21%
213_javac	30.9M	99.56%	44.1M	79.19%
222_mpegaudio	99.0M	100.78%	97.9M	100.70%
228_jack	260.2M	100.69%	272.1M	100.05%

Figure 5-6: Processor modification effects on SPECjvm98 performance. Simulation was run on 1 simulated processor in UVSIM. The *Base* and *Trans* columns are runtimes using the `Base` and `Trans` version of the applications respectively. The *old* and *new* columns are the runtimes using the `old` and `new` processor configurations respectively. The *old* runtime is given as the number of processor cycles. The *new* runtime is given as a percentage of the *old* runtime.

snapshots, 64 integer physical registers, and 64 floating point physical registers.

The pipeline modifications have negligible performance impact on normal non-transactional codes as shown in the `Base` results in Figure 5-6. All `new` applications run within 1% of the `old` configuration. Although one might expect the extra physical registers and rename snapshot to increase performance, in fact, the additional hardware reduces performance in most cases. `213_javac` is the only application that runs faster after the pipeline modifications. All other applications suffer from a slight performance decrease. The fact that the reorder buffer and cache are not scaled accordingly is the most likely cause of the performance decrease. Since the performance difference is negligible, however, there is no need to compensate for this effect.

The pipeline modifications improve performance significantly for transactional codes as shown by the `Trans` results in Figure 5-6. The register snapshot uses physical registers otherwise used for speculation. Therefore, replacing those registers increases performance as expected. The performance increase is especially high for applications with large transactions. The `new` version of `213_javac`, in particular, runs more than 20% faster than the `old` version. In fact, the `old` version of `213_javac` has 25x more instruction decode stalls caused by insufficient physical registers. A similar performance increase is also seen in the `200_check`, `202_jess`, and `209_db` as well. `213_javac`, however, is affected the most since it has the largest transactions.

5.6 Parallel transactional program behavior

To gain insight into parallel transactional program behavior, four parallel microbencharmks were implemented: `NodePush`, `Counter`, `BinaryTree`, and `LinkedList`. The `NodePush` application shows that small transactions with low contention run faster than with locks, as expected. The `Counter` application shows that small transactions are likely to complete even without backoff in high contention. The `BinaryTree` application shows that transactions can indeed exploit concurrent hidden by conventional locks. The `LinkedList` application shows that conventional locking can outperform transactions in some situations where simple backoff is insufficient. In this section, I present these results quantitatively.

5.6.1 The `NodePush` microbenchmark

I implemented the `NodePush` parallel microbenchmark application to examine program behavior for small transactions with low contention. Such transactions represent the expected common case in highly parallel applications. The results show that, as expected, UTM executes small transactions efficiently. In fact, the overhead is much lower than that of locks in all cases.

The `NodePush` application has a large random fully-connected graph containing 10,000 nodes. Like the example from Chapter 2, each processor repeatedly pushes flow from one node to another within the graph. One push operation is performed per iteration of the microbenchmark. The push operation is performed atomically on random nodes. This type of activity is typical in graph algorithms such as the parallel push-relabel maximum-flow algorithm [3, 10]. The locking version of `NodePush` uses one spin-lock per node. Locks are acquired in ascending order to prevent deadlock. The transactional version performs the push in a transaction with randomized exponential backoff.

The results confirm that UTM overhead is indeed low in the common case. The performance results are shown in Figure 5-7. Both versions exhibit a similar performance trend since both fine-grained locks and transactions exploit the available

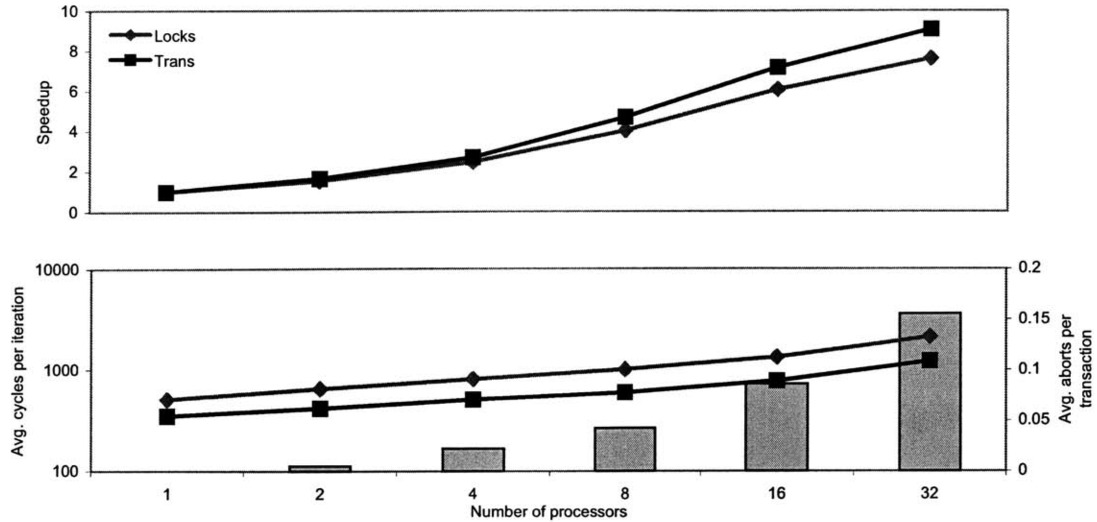


Figure 5-7: `NodePush` microbenchmark performance. The Locks and Trans lines show the runtime for the locking and the transactional version respectively on the left y-axis. The Aborts bars show the number of aborts for the transactional version on the right y-axis.

concurrency. Both versions allow processors to independently work on different nodes at the same time. The locking overhead, however, is much higher than that of transactions in every case. For 32 processors, the locking version is almost 2x slower than the transactional version per iteration. Moreover, the speed up gained by the transactional version is higher as well. For 32 processors, the transactional version shows a 9.1x speedup, whereas the locking version shows only a 7.6x speedup.

5.6.2 The `Counter` microbenchmark

I implemented the `Counter` parallel microbenchmark application to examine program behavior for small transactions with high contention. The results show that the extremely low overhead of small transactions enables them to almost always complete even when contention is high.

The `Counter` microbenchmark has one shared variable that each processor atomically increments repeatedly. One atomic increment is performed per iteration of the microbenchmark. Each atomic block is only a few instructions long and every processor attempts to read and modify the same shared location repeatedly. The lock-

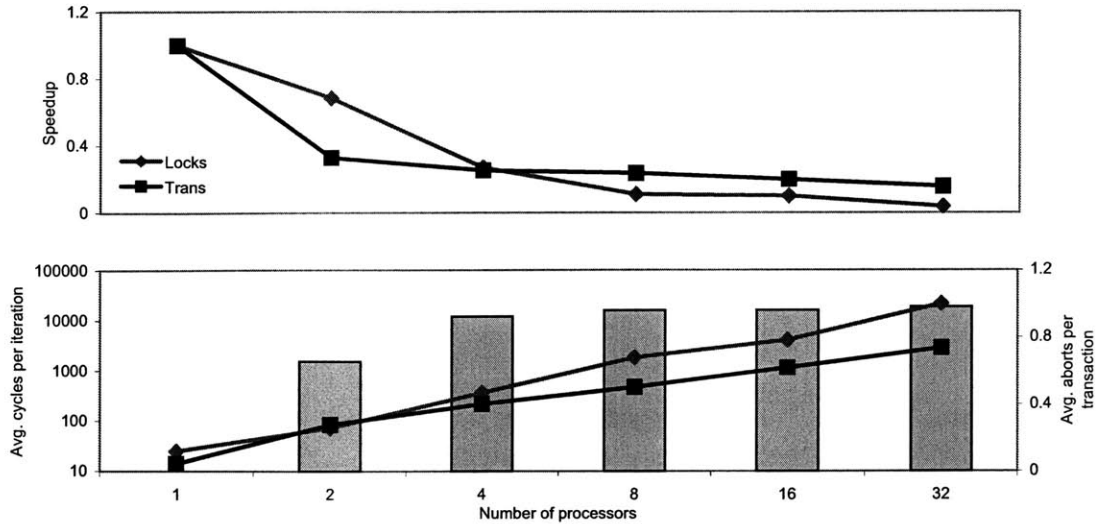


Figure 5-8: `Counter` microbenchmark performance. The Locks and Trans lines show the runtime for the locking and the transactional version respectively on the left y-axis. The Aborts bars show the number of aborts for the transactional version on the right y-axis.

ing version uses a global spin-lock protecting the shared variable. The transactional version performs the increment in a transaction with no backoff.

Since all processors are accessing the same memory location, both the locking and transactional versions scale poorly as expected. Transactions, however, still perform better than locks. The performance results are shown in Figure 5-8. The locking version performs worse than transactions because the LL/SC sequence in the spin-lock routines causes many cache interventions even when the lock cannot be obtained. On the other hand, the transactional version scales much better, despite having no backoff policy. When a transaction obtains a cache line, it will likely execute a few more instructions before receiving an intervention since the network latency is relatively high. Starting at 8 processors, there is 1 average abort per iteration. This result implies that after obtaining the cache line, each transaction has enough time to commit and start the next iteration. Once the next iteration starts, however, the transaction is aborted by an incoming intervention. Then the transaction restarts and waits for the cache line. Once it obtains the cache line, the cycle continues. Therefore, small transactions make progress even when contention is high.

5.6.3 The BinaryTree microbenchmark

I implemented the BinaryTree parallel microbenchmark application to examine program behavior for larger highly concurrent data structures. The results show that even fine-grained locking cannot always exploit the available parallelism in such data structures. On the other hand, transactions allow efficient parallel access with little programming effort.

The BinaryTree microbenchmark has a large binary search tree containing 1,000 elements. The binary tree is balanced at the beginning of each run. Each processor accesses a random element in the tree repeatedly. On each processor, 95% of the accesses are lookup operations. The remaining 5% of the accesses are deletions and insertions. Each lookup, deletion, and insertion operation is performed using the standard binary search tree algorithm [5, chap. 12] atomically. Such data structures which are read often but rarely modified are common in applications such as databases. Two locking versions of BinaryTree were implemented: a global locking version and a fine-grained locking version. The global locking version uses one global spin-lock that is acquired on each lookup, deletion, and insertion. The fine-grained locking version has one spin-lock per element in the tree. The locks are acquired in hand-over-hand fashion when the processor traverses the tree. The transactional version performs each operation in a transaction with randomized exponential backoff.

Since the elements are chosen at random, the time to complete each operation may vary drastically depending on the location of the element in the tree. Therefore, for the purpose of comparison, traversing one node in the tree is considered one *iteration* of the microbenchmark. One operation may take several iterations to complete.

Since lookup operations only perform reads, they are inherently concurrent. Therefore, most accesses to the tree can be performed in parallel. Locks cannot effectively exploit this parallelism as shown in Figure 5-9. As expected, the global locking version performs poorly since all operations are mutually exclusive. Similar performance, however, is seen even with fine-grained locking. For 8 processors, the fine-grained locking version performs better than the global locking version but both scale poorly.

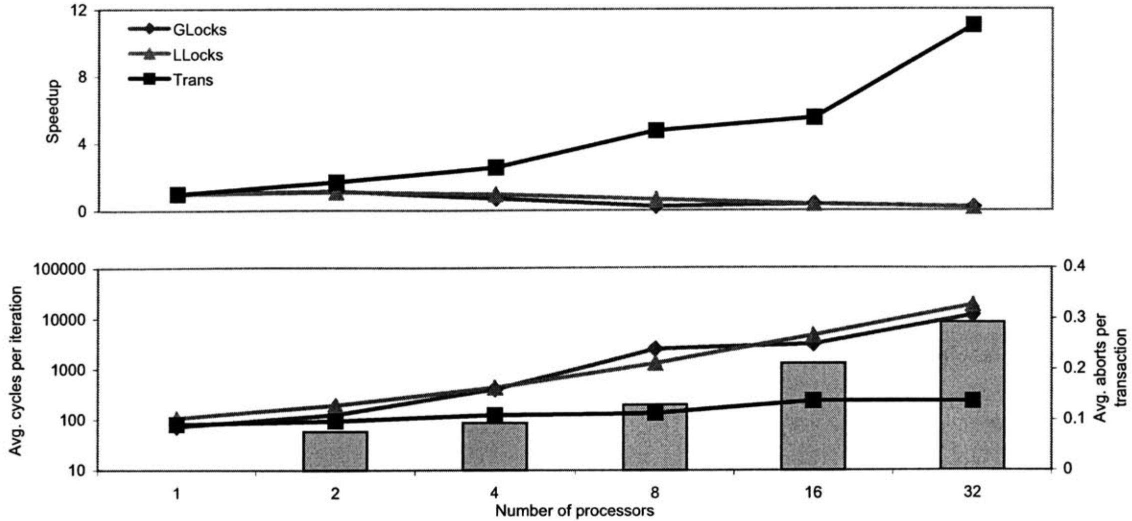


Figure 5-9: `BinaryTree` microbenchmark performance. The GLocks and LLocks lines show the runtime for the global locking and fine-grained locking versions respectively on the left y-axis. The Trans line shows the runtime for the transactional version on the left y-axis. The Aborts bars show the number of aborts for the transactional version on the right y-axis.

Both locking versions show a 5x slowdown for 32 processors.

Poor fine-grained locking performance is caused by contention near the root of the tree. Although most of the target elements are near the bottom of the tree, each operation must traverse through the top nodes to get to the bottom. In most cases, the operations only read the top nodes on the way down the tree. Therefore, in theory, most accesses to the top nodes can be performed in parallel. Locks, however, cannot effectively exploit this type of parallelism. Each time a processor reads the root node, for example, the node lock must be acquired to prevent another processor from writing to it at the same time. While that processor holds the lock, all other processors cannot even read from the node. Therefore, the bottleneck at the top of the tree prevents even fine-grained locking from scaling as desired.

On the other hand, transactions are better suited to exploit the parallelism available in `BinaryTree`. The transactional version shows almost an 11x speedup for 32 processors, whereas both locking versions show a slowdown. The locking bottleneck does not occur since many transactions can concurrently read from the same node

at the same time. Conflicts only occur when a processor writes to a node as part of a deletion or insertion operation. The target nodes for these operations are likely to be at the bottom of the tree since they are chosen at random. Therefore, conflicts are unlikely. Even for 32 processors, there are less than 0.3 aborts per successful transaction.

In addition to the poor lock performance, adding fine-grained locks into `BinaryTree` was not a trivial programming task. In all the other microbenchmarks, adding locks was not difficult since coarse-grained locking was mostly used. Coarse-grained locks are simply acquired at the start each atomic region and released at the end of the region. On the other hand, adding fine-grained locks often increases program complexity significantly. In `BinaryTree`, adding fine-grained locking increases the code size by 16%. Although programming complexity is a qualitative measure, this increase gives an indication of the additional programming effort required to perform tasks such as deadlock avoidance.

The fine-grained locking used in `BinaryTree` is not optimal. It is possible to exploit multiple reader concurrency by using more sophisticated techniques. For example, Kung and Lehman [31] designed a concurrent binary search tree by making copies of tree sections and redirecting searches by manipulating pointers. The technique is similar to lock-free and wait-free synchronization [18, 19, 33]. The additions to the data structure, however, are not straightforward. This work shows the high tradeoff between performance and programming ease that exists with conventional locks. The binary search tree is a simple data structure that inherently has high concurrency. Trying to exploit this concurrency with conventional synchronization is possible but is difficult. With transactions, on the other hand, the programmer can exploit this concurrency easily.

5.6.4 The `LinkedList` microbenchmark

I implemented the `LinkedList` parallel microbenchmark application to examine program behavior for moderately sized transactions with varying contention. The results show that when contention is low, transactions are efficient as expected. As contention

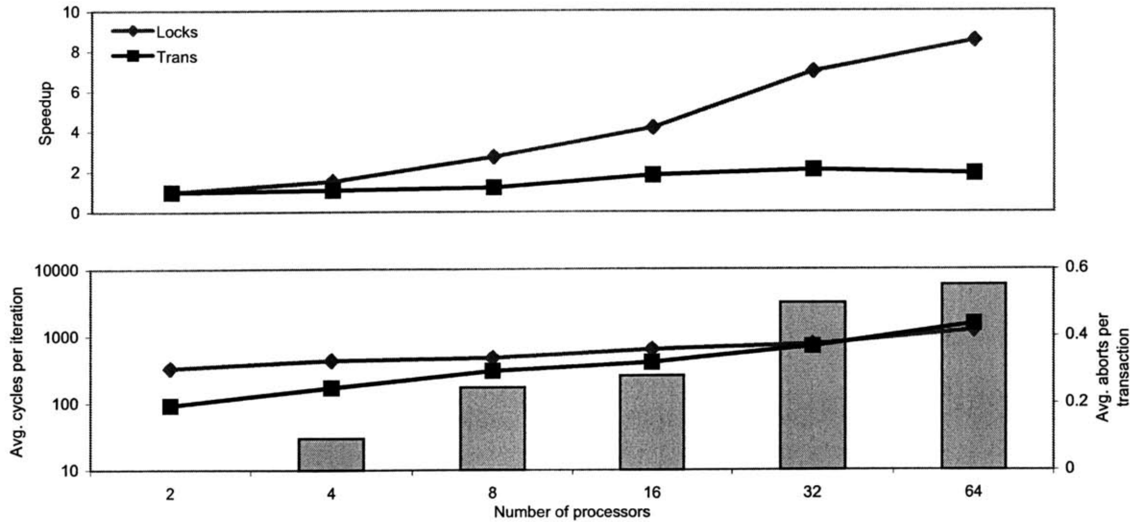


Figure 5-10: `LinkedList` microbenchmark performance. The Locks and Trans lines show the runtime for the locking and the transactional version respectively on the left y-axis. The Aborts bars show the number of aborts for the transactional version on the right y-axis.

increases, however, the efficiency is highly dependent on the backoff policy. In some cases, simple backoff policies are insufficient.

The `LinkedList` application has an equal number of consumer and producer processors. Each consumer has a doubly-linked list which starts with 50 elements. Each consumer repeatedly takes items from the head of its list while each producer repeatedly adds items to the tail of a random list. The locking version uses one spin-lock per list, and the entire list is locked when adding or removing items. The transactional version performs each list addition and deletion in a transaction with randomized exponential backoff.

Transactions perform better than conventional locks for low processor counts as shown in Figure 5-10. This result is expected since transactions have lower overhead and allow concurrent accesses to the head and the tail of each linked list. For 2 processors, the locking version actually requires 3.5x more cycles per iteration than the transactional version.

As the number of processors increases, producers conflict more often. The contention increases even though the number of lists is scaled with the number of proces-

sors. A list becomes backed-up when two producers access it simultaneously. While the two processors contend for the list, the probability that another processor attempts an access increases.

When `LinkedList` was first implemented, no backoff policy was used. As a result, livelock prevented program completion for more than 2 processors. To solve the livelock problem, `LinkedList` was implemented with randomized exponential backoff.

The backoff policy solved livelock but results in poor performance for high processor counts. At high contention, transactions abort often and spend much time in backoff. On the other hand, the locking version wastes no time since processors simply wait for locks to be freed before proceeding. Therefore, after 32 processors, the transactional version performs worse than the locking version. Moreover, for 64 processors, the locking version shows an 8x speedup whereas the transactional version shows only a 2x speedup. Poor transactional performance is caused by the increasingly high number of aborts (more than one abort for every two transactions) starting at 32 processors. These results suggest that a simple backoff policy is insufficient in situations with high contention.

The `LinkedList` application illustrates the importance of a good backoff policy in optimistic transaction systems which have no forward-progress guarantees, such as UTM. In contrast, lock-free and wait-free synchronization has strong forward-progress guarantees. Perhaps such a guarantee is needed to achieve reasonable performance in high contention. This problem is one of UTM's main challenges.

Chapter 6

Design Alternatives

Many design alternatives were considered during the UTM design process. Firstly, an integrated hardware-software approach to unbounded transactions was considered. The integrated approach uses hardware to run small and short transactions, but falls back to software to run large and long transactions. The results indicate that such an integrated approach has impractically high performance overhead. A linear overflow data structure was also considered. The linear data structure is simple but its poor performance limits its use in practice. A software register snapshot mechanism was also considered. The software mechanism requires no hardware changes but has performance and compatibility problems. Lastly, a nontransactional mechanism called NITs (nested independent transactions) with clean semantics was also considered. The resulting semantics are better than UTM's nontransactional semantics but the implementation requires widespread hardware modification.

In this chapter, I describe these alternatives and highlight their design tradeoffs. In Section 6.1, I outline the integrated hardware-software system. In Section 6.2, I describe the linear overflow data structure. In Section 6.3, I outline the software register snapshot. In Section 6.4, I describe the NIT semantics and hardware mechanism.

6.1 Integrated hardware-software approach

Previous hardware transaction systems impose transaction size and length limitations that generally do not exist in software transaction systems. Unfortunately, software systems generally have much higher overhead than hardware systems. Therefore, it is natural to consider whether an integrated hardware-software design can achieve the best of both worlds by running small transaction in hardware, and large transactions in software. Since both hardware and software designs have previously been studied, they only need to be integrated. In this section, I describe such an integrated design called HSTM (hardware-software transactional memory). I conclude, however, that HSTM incurs high performance overhead and cannot be easily integrated into existing systems.

Base hardware and software transactions

HSTM integrates the hardware design by Herlihy and Moss [22, 23] and the software design by Ananian and Rinard [2]. I call the base hardware system HTM (hardware transactional memory) and the based software system STM (software transactional memory). HTM is simply UTM without overflow and context-switching support. HTM is used since it has low performance overhead for small transactions. STM is the FLEX software transaction system. STM is used since it is freely available and is currently one of the most efficient software transaction designs [2].

STM is an object-based system where the primary data structure is an object as the Java programming language. Figure 6-1 shows the STM object structure. Each object has a `versions` and `readers` pointer, in addition to its normal fields. `versions` points to committed versions of the object, and possibly a transactional version. The transactional version stores uncommitted speculative state. Each transactional version is linked to a transaction tag trough the `owner` pointer. The transaction tag is specific to each transaction. `readers` points to a list of running transactional readers of the object.

When a transaction writes an object field, STM copies the object if a copy does

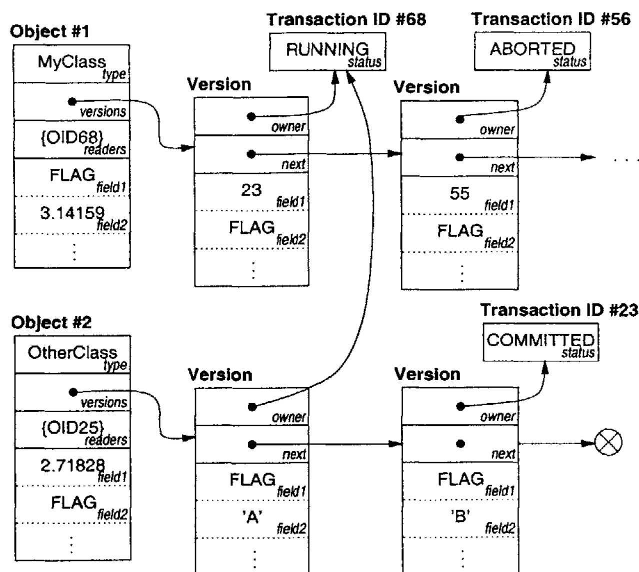


Figure 6-1: STM object structure. Each object has, in addition to its normal fields, a `versions` pointer and `readers` pointer. `versions` points to committed versions of the object and possibly a transactional version. `readers` points to a list of pending transactional readers of the object. The actual readers list is not shown. Each version is linked to a transaction tag through the `owner` pointer.

not already exist. The copy serves as the backup data in case the transaction is aborted. Then STM sets the original object field to some predetermined constant FLAG. The FLAG value indicates to other transactions that the field is being written transactionally. Then STM creates a new transactional version if a transactional version does not already exist. Then STM makes the modification to the transactional version. Lastly, STM adds the transaction to the object's readers list.

Transactional reads, on the other hand, are much simpler. When a transaction reads a field, STM simply adds the transaction to the object's readers list.

When a transaction commits or aborts, STM changes the transaction tag accordingly. This change switches the status of all linked transactional versions.

To detect conflicts, STM performs a check on each transactional read and write operation. On each write, STM checks if the object has any readers in the readers list. If there are readers, STM aborts them all by traversing the readers list. On each read, STM checks if the field is marked FLAG. If the value is FLAG, another running transaction may be writing to the field. Therefore, STM traverses the versions list and

aborts any running transactions. The value may also be FLAG, however, if a committed transaction previously wrote to the field. In this case, the reading transaction reads the most up-to-date value from the committed version and updates the original object.

STM is implemented in the compiler so STM can take advantage of compiler analysis to optimize many of the checks. For example, if a transaction reads from a field it previously wrote, it can simply read the value directly from the transactional version without checking the field in the original version. Such optimizations decrease STM performance overhead for long transactions [2] since long transactions tend to access the same fields often.

The HSTM design

The HSTM design is straight-forward. HSTM simply attempts to run all transactions in hardware. If the hardware transaction fails because of a size or length limitation, HSTM attempts to run the transaction in software. To make such a switch possible, however, some changes to HTM are made.

Both hardware and software transactions may run on the same system at the same time. Therefore, HSTM must detect conflicts between both hardware and software transactions. Fortunately, if a hardware transaction conflicts with another hardware transaction, the normal HTM mechanism detects the conflict. Similarly, the normal STM mechanism detects software-software conflicts.

Therefore, HSTM must add the ability to detect conflicts between hardware transactions and software transactions. Fortunately, if a software transaction touches a location owned by a running hardware transaction, the software transaction triggers a cache intervention and the normal HTM mechanism detects the conflict. On the other hand, if a hardware transaction touches a location owned by a running software transaction, the conflict is not be detected by the normal HTM or STM mechanism.

HSTM performs additional software checks to detect such conflicts as shown in Figure 6-2. On each hardware transactional load, HSTM checks if the loaded value is FLAG. If the value is FLAG, a running software transaction may have written to that field. Therefore, HSTM aborts the hardware transaction. On each hardware

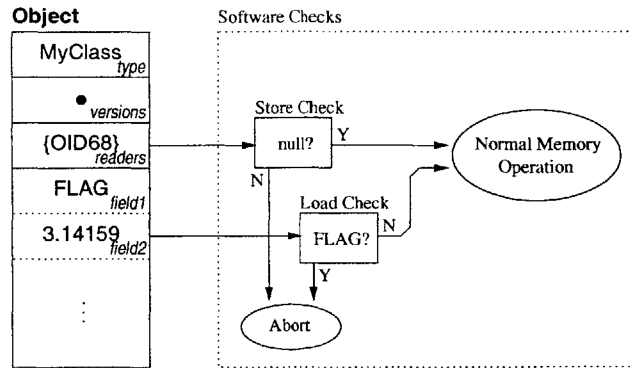


Figure 6-2: Additional checks performed on HTM operations. These checks are performed in software. On each load, if the field being read is flagged, the hardware transaction is aborted. Otherwise, the hardware transaction proceeds normally. On each store, if the readers list of the object not empty, the hardware transaction is aborted. Otherwise, the hardware transaction proceeds normally.

transactional store, HSTM checks if the object's **readers** pointer is `NULL`. If the pointer is not `NULL`, a running transaction may have read that field. Therefore, HSTM aborts the hardware transaction.

Lastly, since these additional checks are performed in software, a hardware transaction must be able to abort itself in software if a conflict is detected. Therefore, an `xABORT` instruction is added to the ISA.

Evaluation

To evaluate HSTM performance overhead, I implemented a partial HSTM system in UVSIM and a microbenchmark. The HTM component was fully implemented in UVSIM as part of the UTM implementation. The STM component was not fully implemented since STM functionality was not available in the FLEX compiler at this time. Therefore, I hand-coded the STM structure into the microbenchmark. The goal is only to evaluate common case performance overhead. Therefore, I only implemented the subset of STM used in common case single-processor execution. Similarly, the software checks for hardware transactions were hand-coded into the microbenchmark as well.

A microbenchmark similar to `NodePush` from Chapter 5 was used as simulation

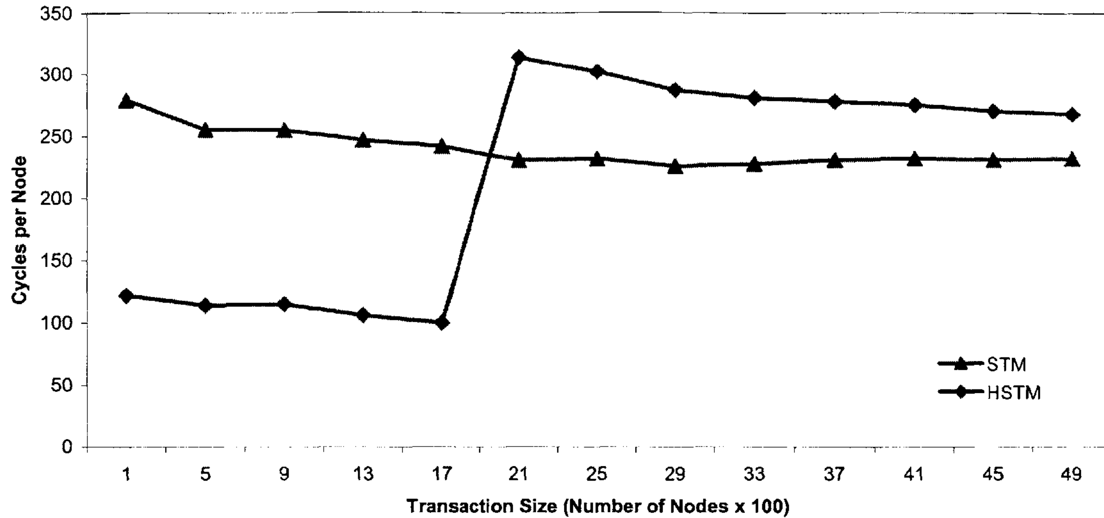


Figure 6-3: HSTM performance for various transaction sizes.

workload. The `NodePush` microbenchmark accesses a variable number of nodes to simulate transactions of variable size. Nodes are accessed in sequential memory order.

Using this evaluation environment, I found that HSTM hardware transactions incur a 2.2x slowdown over standard HTM transactions. The slowdown is caused by the additional software checks in every hardware transaction.

The 2.2x slowdown, however, is still much faster than the slowdown from STM. Figure 6-3 shows the performance of HSTM compared to pure software transactions. For transactions sizes less than 2,100 nodes, all transactions fit in hardware and run almost 3x faster than software transactions. Starting at 2,100 nodes, however, transactions no longer fit in hardware and the high STM overhead is incurred.

Design tradeoffs

Although, in principle, HSTM runs small and short transactions fast in hardware, in practice, even hardware transactions incur a high (2.2x) performance overhead. The overhead is caused by the additional software checks. Thus, one might argue that performing the checks in hardware can decrease the overhead. Even hardware checks, however, will not alleviate all of the overhead since the checks require addition

memory references.

HSTM also incurs a memory overhead as well. Each object requires the added `readers` and `versions` pointers even when running hardware transactions. The additional fields are necessary since another processor may be operating on the object in software. Since HSTM runs most transactions in hardware, however, the additional memory overhead is unused most of the time.

Lastly, HSTM does not support legacy function calls since HSTM must be compiled into any code running within a transaction. In fact, all designs that rely on the compiler for transactions do not support legacy function calls. Therefore, such designs are difficult to integrate into existing systems.

The main advantage of HSTM is that it can support unbounded transactions with minimal hardware modification. The overhead and incompatibility, however, outweigh this advantage. Moreover, implementing unbounded transactions completely in hardware is not overly complicated, as shown by the UTM design. Therefore, HSTM's integrated approach is not as attractive as UTM's hardware-only approach.

6.2 Linear overflow data structure

A key assumption in the UTM design is that overflows are infrequent enough that high performance overhead from overflows can be amortized efficiently over the fast common case. Therefore, it may be reasonable to use the simplest overflow data structure even if overhead is extremely high. When UTM was first designed, UTM used a simple unsorted array as the overflow data structure. In this section, I show that the simple unsorted array leads to unreasonable performance overhead.

The unsorted array is much simpler than a hash table. The unsorted array does not need any additional data such as the validity bits or the linked list pointers. The overflow handler simply needs to track the start and the end of the array.

Inserting an overflowed cache line into the unsorted array requires only constant time since the new line is simply added to the end of the array. A lookup, however, requires a linear search through all the overflowed lines. The search time is linearly

Benchmark	Base time (cycles)	Locks time	Trans time	Time in trans	Time in overflow
		(% of Base time)		(% of Trans time)	
200_check	8.1M	124.0%	101.7%	32.9%	0.0037%
202_jess	75.0M	140.9%	107.1%	59.3%	0.0076%
209_db	11.7M	142.4%	105.1%	53.9%	0
213_javac	30.7M	169.9%	**1469.5%**	**99.0%**	**93.0%**
222_mpegaudio	99.0M	100.3%	99.6%	0.8%	0
228_jack	261.4M	175.3%	104.4%	32.2%	0.0026%

Figure 6-4: SPECjvm98 performance with linear overflow data structure. Simulation was run on 1 simulated processor in UVSIM with an input size of 1. The *Time in trans* and *Time in overflow* are the times spent actually running a transaction and handling overflows respectively.

proportional to the number of overflowed cache lines. When only a few lines overflow, the search overhead is acceptable. When the number of overflowed lines is high, however, the linear search can incur an impractically high performance overhead.

The results from running SPECjvm98 with an unsorted array show that the unsorted array insufficient to handle all cases. The performance results are shown in Figure 6-4. In all but one case, the overall overhead incurred by UTM is less than 8% over the base version and is much lower than locking overhead. The 213_javac application, however, incurs a 15x slowdown over the base case. The 213_javac locking version shows only a 1.7x slowdown. Almost all of the 15x slowdown is in the overflow handling. In fact, 99% of the total runtime is spent in a transaction and 93% of that time is spent handling overflows. Therefore, 92% of the entire runtime is spent handling overflows, leaving only 8% for actual program execution. This result explains the perceived 15x slowdown.

Design tradeoffs

The main advantage of the unsorted array is its simplicity. This advantage, unfortunately, is overshadowed by the unbounded search time. The transaction can grow so large that the search time cannot be efficiently amortized over the fast common case. Such performance loss is seen in 213_javac. On the other hand, hash table search

times are mostly constant since collisions are rare. For this reason, the UTM overflow data structure was changed to a hash table.

Further inspection of the `213_javac` application reveals that the entire application is essentially run within one large transaction. In fact, the main method `Javac.compile()`, which implements the entire compilation process, is marked as synchronized by the programmer. Therefore, the `213_javac` application is clearly not intended for parallel execution. One might argue that it is reasonable to incur a high performance overhead for such programs since they clearly need to be rewritten if any parallel performance is desired. Such an approach, however, makes parallel programming more difficult and is therefore undesirable.

6.3 Software register snapshot

UTM requires a register snapshot to restore the processor state after an abort. The snapshot, however, can be performed in software instead of hardware. Although the UTM hardware snapshot only requires minor changes to the processor core, a software snapshot requires no changes at all. Further, a software snapshot can be completely abstracted away from the programmer so that the easy-to-program environment is maintained. This abstraction is maintained by the compiler which inserts additional code into the application to save and restore the registers. In this section, I sketch the necessary compiler modifications to implement a software snapshot. I contend, however, that software snapshots have high performance overhead and do not facilitate easy integration into existing system.

The software snapshot can save all the register values before starting each transaction. Before each `xBEGIN`, the compiler simply inserts code that stores the contents of each register to main memory. If the transaction aborts, the saved values can be restored by the abort handler. Unfortunately, performing the entire snapshot at `xBEGIN` can incur a high performance overhead.

Alternatively, the software snapshot can take advantage of the registers routinely saved as part of subroutine calls. The subroutine linkage only needs to be modified

slightly to accommodate the register snapshot.

In typical calling conventions, registers are divided into callee-saved, caller-saved, and parameters. Caller-saved registers are those that the caller needs after calling the subroutine, but the caller is responsible for saving and restoring their values. Therefore, the callee (or subroutine) can simply overwrite the caller-saved registers if necessary. Callee-saved registers are those that the subroutine is responsible for saving and restoring. Thus, a subroutine can only overwrite callee-saved registers after saving them. The subroutine is also responsible for restoring them. Parameter registers are those used to pass subroutine parameters. Parameter registers are generally not saved or restored. The subroutine may overwrite them if necessary and their values are lost. Therefore, the parameter registers are the only registers not saved by the normal calling convention.

The software snapshot can leverage the fact that subroutine linkage already saves most of the registers. The calling convention can be modified to save the parameter registers, thus saving all registers.

The calling convention, however, saves registers on the stack and the stack pointer is changed on each subroutine call. Therefore, to restore the saved registers, the abort handler must restore the saved values as it *backs out* of the procedures. The abort handler effectively *returns* from each subroutine restoring all the registers along the way. When the abort handler backs out to the subroutine in which the transaction was started, all the registers are restored.

To accomplish the procedure *back out*, a different abort handler is associated with each subroutine. The abort handler can restore all the necessary registers for that subroutine with instructions similar to the subroutine's return code. On each subroutine call, the associated abort handler is set in the processor and the old abort handler is saved. Therefore, when an abort occurs, the processor immediately jumps to the abort handler associated with the running subroutine. The abort handler restores all the registers and then jumps to the previously saved abort handler. Thus, each abort handler associated with every subroutine is executed until finally the initial abort handler is called. The initial abort handler can then perform tasks such

as backoff and retry.

Since registers are saved only on subroutine boundaries, transactions must start and end on subroutine boundaries as well. Otherwise, the necessary register state cannot be restored. This requirement, however, can be hidden from the programmer. The programmer can still use UTM transaction semantics since the complier can simply transform each `xBEGIN/xEND` block into a subroutine-like code block.

Design tradeoffs

Although UTM's hardware snapshot mechanism is not overly complicated, it still requires modification to the processor pipeline. Therefore, a software snapshot is attractive since no hardware modifications are necessary.

Since the compiler inserts software snapshot instructions, compiler analysis can minimize the number of saved registers. Only those registers used by the program need to be saved. A hardware snapshot, on the other hand, must save all visible architectural registers even if many of them are not used.

Unfortunately, software register snapshots inevitably incur additional performance overhead. In the simple approach, where every register is saved at `xBEGIN`, the overhead can be extremely high especially for small transactions. In the subroutine linkage approach, the overhead is reduced but eliminated. There is overhead associated with saving parameters and abort hander addresses. In addition, transactions do not all naturally fall on subroutine boundaries. Therefore, treating each transaction as a transaction incurs additional overhead as well.

Lastly, using subroutine linkage for the register snapshot does not support legacy function calls. All the code executed within a transaction must be compiled to save the appropriate registers. Therefore, a software snapshot scheme cannot be integrated easily into an existing system.

Although the software snapshot alleviates the need to change the processor core, its high overhead and incompatibility make it undesirable for the UTM system. Further, the hardware changes necessary to support the hardware snapshot are not overly complicated since they use many of the existing mechanisms in the processor. For

these reasons, UTM uses a hardware register snapshot instead of software.

6.4 Nested independent transactions

Tasks such as debugging or logging within a transaction require nontransactional operations. As described in Section 4.7, nontransactional instructions can be achieved without any change to the memory system which includes the cache-coherency protocol, network messages, and directory controllers. Unfortunately, the resulting semantics are awkward since it exposes the cache line size and nontransactional regions can be interrupted by aborts. Therefore, it is natural to consider whether better semantics can be achieved by modifying the memory system. In this section, I describe a mechanism called NITs (nested independent transactions) designed to achieve that goal. NITs have much cleaner semantics but require significant hardware modification. I argue that since nontransactional operations are used rarely, the necessary hardware changes are not justified.

Semantics

There are two additional instructions in the ISA that define the scope of the nested independent transaction:

`nitBEGIN`: Marks the start of a NIT.

`nitEND`: Marks the end of a NIT.

All instructions between the `nitBEGIN` and `nitEND` are considered part of the NIT. As the name suggests, a NIT must be nested within a normal transaction but is independent from it. Since it is nested within a parent transaction, when the NIT is still running, it is considered part of the parent. Thus, a conflict causes both the parent and the NIT to abort simultaneously. Since the NIT is independent, however, the NIT commits independent of the parent. Thus, when the NIT commits, all of its memory updates are made globally visible even though the parent transaction is still running.

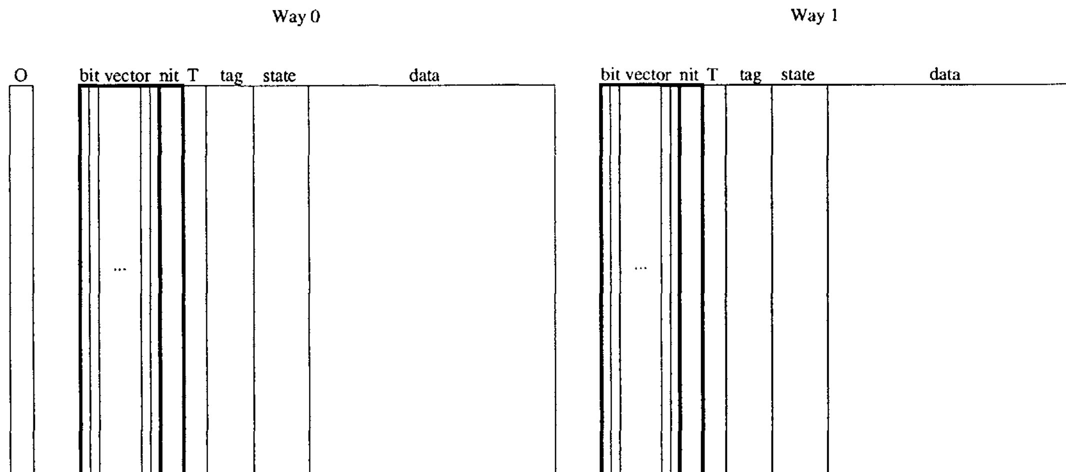


Figure 6-5: Cache modifications for NIT implementation. An additional NIT bit and bit vector are added per cache line. The bit vector contains one bit per data byte in the cache line. A 2-way set-associative cache is shown.

When the NIT commits, every *byte* in memory that was modified by the NIT is made globally visible. NITs cannot abort the parent transaction so a NIT can access a memory location previously accessed by the parent. If the NIT changes such a memory location, the change is also be made globally visible once the NIT commits.

NITs can contain nested transactions. Both NITs (`nitBEGIN`/`nitEND`) and normal transactions (`xBEGIN`/`xEND`) can be nested within a NIT. All nested transactions are simply subsumed into the outermost NIT. Thus, semantically, there are only two actual levels of independent transaction nesting: the normal transaction and the nested independent transaction. All other nested transactions are subsumed into one or the other.

NIT semantics do not suffer from the problems associated with the UTM non-transactional semantics presented in Section 4.7. Since NITs operate at a byte granularity, the cache-line size is not exposed. In addition, NIT memory operations are not globally visible until the NIT commits. Therefore, aborting a NIT cannot result in inconsistent data structures.

Modifications to the cache

NIT semantics require the underlying hardware to operate at a byte granularity. Therefore, some modifications to the cache and memory system are necessary. In the cache, a NIT bit and a bit vector are added to each cache line as shown in Figure 6-5. The bit vector contains one additional bit per byte in the cache line. For a 128 byte cache line, the bit vector contains 128 bits. Like the rest of the cache line, the NIT bit and bit vector are written to the overflow data structure if the line is evicted.

The additional NIT bit and bit vector are used to track the cache lines and the individual bytes touched by the NIT. The NIT bit marks the cache lines that are part of the running NIT. The bit vector marks the modified bytes in a cache line if the entire cache line cannot be committed.

A normal transactional operation not within a NIT marks the target cache line as before by setting the T bit. Similarly, a NIT operation marks the target cache line by setting the NIT bit. If the target cache line is transactional, however, the NIT operation marks the bit vector corresponding to the modified bytes in the cache line. The bit vector is only set when the cache line is transactional since the T bit indicates the cache line was previously accessed by the parent transaction. Modified data from the parent transaction cannot be committed with the NIT. Therefore, the bit vector indicates those bytes can be committed.

If a cache intervention hits a line with the NIT bit set, the transaction and the NIT are aborted. Both the parent transaction and the NIT are aborted simultaneously by clearing all the T bits, NIT bits, and bit vectors in the cache. As in a normal abort, modified T or NIT cache lines are invalidated.

When a NIT commits, the NIT bits are simply cleared. The bit vectors, however, are left unchanged. If the NIT cache line was not part of the parent transaction, clearing the NIT makes the entire line globally visible. If the NIT cache line was part of the parent transaction, clearing the NIT bit does not divorce the line from the parent transaction since the T bit is still set. Therefore, only the bytes marked in the bit vector can be made globally visible since the uncommitted parent transaction

may have modified the other bytes. To ensure that only the NIT modified bytes are made globally visible, the bit vector is left unchanged when the NIT commits. Thus, a cache line with the NIT bit unset but a nonzero bit vector contains partially committed data.

These lines remain partially committed until the parent transaction commits. Once the parent transaction commits, the bit vectors are cleared and all the bytes are made globally visible.

Before the parent transaction commits, however, only those bytes marked in the bit vectors are globally visible. Therefore, a cache intervention that hits such a line requires a partial line reply to send only the committed bytes back to the request. Then the requestor obtains the remainder of the line from main memory. Partial line replies are not supported in normal cache-coherency protocols so modifications to the memory system are necessary.

Modifications to the memory system

To support operations with byte granularity, modifications are required to the cache-coherency protocol, network messages, and directory controllers. I outline one of the many ways to implement these changes.

Firstly, network messages must support partial memory lines. For simplicity, the network message length can remain constant. The message is simply modified to include a bit vector that indicates which bytes are valid. The network message bit vector can be the same bit vector from the partially committed cache line. If a network component receives a message with an incomplete bit vector, the hardware must treat the data as only partial.

Next, the directory controllers must accept partial line replies in response to a memory request. If a cache intervention hits a partially committed line, the partial line is sent to its home node. Once the home node receives the partial line, it retrieves the remainder of the line from memory. Then the home node sends the reconstructed complete line to the original requestor.

Partial line replies require no more network latency than requests for silently

dropped exclusive lines. In both cases, once the cache intervention is received, two network hops are necessary before the original requestor receives the line. For a silently dropped exclusive line, the cache intervention misses in the cache and a miss reply is sent to the home node. Once the home node receives the miss reply, it retrieves the line from main memory and sends it to the original requestor. Similarly, for a partial line reply, after receiving the cache intervention, a message containing the partial line is sent to the home node. Once the home node receives the partial line, it retrieves the rest of the line from memory and sends the complete line to the original requestor.

Lastly, the memory system must also support partial line write-backs to write partially committed data back to main memory. Partial line write-backs are required if the parent transaction writes to a line partially committed line. After the home node receives a partial line write-back, it simply merges the partial line with the rest of the line in memory. The write-back is necessary since the main memory must contain the most up-to-date consistent value while the cache contains the speculative transactional values. The partial line write-back is similar to writing back exclusive cache lines when first accessed by a transaction.

Design tradeoffs

The UTM nontransactional semantics given in Section 4.7 rely on the programmer to ensure that nontransactional writes do not hit transactional cache lines. UTM nontransactional semantics do not requires any modifications to the processor, cache, and memory system. Nontransactional operations simply flow through the pipeline like normal memory operations.

Unfortunately, since implementation parameters such as cache line size are normally abstracted away from the programmer, UTM nontransactional semantics are awkward and not portable. For example, nontransactional code that runs error-free on a system with 128-byte cache lines may cause endless aborts on a system with 64-byte cache lines.

Another problem with UTM nontransactional semantics is that nontransactional

code may be interrupted at any point if the transaction is aborted. From the programmer's point of view, handling such interruptions are difficult. The nontransactional code may be in the middle of writing to a data structure before being aborted. In that case, the data structure may be left in an inconsistent state after the abort, since nontransactional instructions always commit.

Since nontransactional operations are expected to be used infrequently, the awkward semantics may not be entirely unreasonable. A fundamental goal of UTM, however, is to make programming easier. Therefore, these awkward semantics are undesirable.

The NIT is a realistic alternative with much cleaner semantics. Since internal bookkeeping is done at the byte level, there are no semantic problems associated with writing to transactional cache lines. Moreover, since NITs are aborted with the parent transaction, they also appear atomic. Therefore, NITs cannot leave data structures in an inconsistent state.

The cleaner semantics do not come without a cost. To support NIT semantics, the cache, network, and cache-coherency protocol require modification. Although most changes are conceptually simple, they are not trivial to implement. Thus, these modifications increase the cost of integration into an existing system.

Supporting NIT semantics requires many changes to the memory system but NIT were still designed with implementation in mind. The goal of NITs is to support cleaner semantics with minimal hardware modification. Therefore, NIT semantics still has two significant restrictions. Firstly, NITs provide only one additional level of real transactional nesting. Ideally, it should be possible to perform a NIT within a NIT, and commit them independently. Secondly, NITs provide only a single abort. Ideally, if a conflict occurred on a NIT memory location, only the NIT should be aborted and retried.

Unfortunately, supporting either multiple nesting levels or separate aborts requires significantly more hardware. In addition, these ideal features do not appear necessary for NITs to be useful. After all, NITs are designed to be used only for rare tasks such as logging.

NITs represent a reasonable alternative to UTM nontransactional instructions. NIT semantics, however, require many changes to the memory system. Since ease of integration is a fundamental goal of UTM, the high hardware overhead prevents NITs from being part of the UTM design. Better semantics for a rarely used feature does not justify widespread hardware modification. If future network protocols support fine-grained accesses, however, NITs are a viable alternative to UTM nontransactional instructions.

Chapter 7

Conclusions

UTM solves many of the problems of previous hardware transaction systems, but UTM has some limitations of its own. UTM context-switches are restrictive and transactions are limited by the size of physical memory. Further, UTM has no forward-progress guarantee and I/O operations are not supported. In this concluding chapter, I describe these limitations in detail. I argue, however, that UTM is still more practical than previous designs, since its limitations are far less restrictive. I contend that UTM is a significant improvement over previous hardware transaction systems.

Firstly, context-switch support in UTM is limited. A transactional thread can be switched off of a processor, thereby suspending the transaction. To resume the thread, however, it must be switched back onto the same processor. In multiprocessor operating systems, threads are often moved from one processor to another. Moving a transactional thread forces UTM to abort the suspended transaction. The transaction cannot be resumed on another processor because the underlying mechanism uses the cache-coherency protocol to track transactional data. Thus, all transactional data is coupled to the processor, not the thread. If the suspended transaction is resumed on another processor, all the directory information for each transactional line needs to be changed to point to the new processor. Otherwise, conflicts are not detected correctly. Since changing all the directory information is not feasible, UTM does not support moving transactions between processors.

Fortunately, threads are generally not often moved from processor to processor. Further, the operating system can be designed to perform thread moves only when necessary. Therefore, even though UTM's context-switching mechanism is limited, the overall effect may still be minimal.

Secondly, the UTM transaction size is limited by the size of physical memory. Since the overflow handler uses physical addresses, the size of the overflow hash table cannot exceed the size of physical memory. If the overflow hash table were virtually addressed, it is possible to swap parts of it out to disk when necessary. Unfortunately, using virtual addresses in the overflow mechanism is not viable since UTM uses the outermost cache to store transactional data. In modern processors, the outermost cache is physically addressed, thus restricting UTM to physical addresses.

Although this size restriction appears to contradict the unbounded claim of UTM, in practice the physical memory capacity is generally sufficient. The size of physical memory is usually more than 3 orders of magnitude larger than that of cache. Therefore, UTM is a significant improvement over previous hardware designs. Further, all transactional systems must have some resource limitation. Thus, it is only reasonable to expect an unbounded system to provide enough resources so the programmer does not need to worry about the limitation. Fortunately, the physical memory in modern multiprocessor systems should be large enough to accomplish that goal.

Another problem imposed by the physical address restriction is that the overflow hash table requires a large sequential block of physical memory. Further, the block of memory must be reserved, since any transaction may overflow. Since physical are used, the reserved space cannot be swapped out to disk if additional memory is needed. Thus, there is a constant memory overhead that cannot be avoided even when transactions do not overflow at all.

Fortunately, UTM allows the operating system to scale the size of the overflow hash table depending on the needs of the transaction. Thus, the operating system can start all transactions with a small hash table and only increase the size if necessary. Since the transaction aborts once the overflow hash table is full, there is a performance loss each time the hash table size is changed.

Another source of performance loss is UTM's lack of a forward-progress guarantee. UTM relies on the programmer to use backoff to resolve conflicts. When contention is high, as seen in the `LinkedList` microbenchmark in Section 5.6, the performance loss of backoff can actually be higher than that of spin-locks. In fact, backoff does not provide any deterministic bound on conflict resolution time.

In contrast, a forward-progress guarantee can be achieved by using techniques such as fallback to mutual exclusion (used in SLE [43]), timestamp-based conflict resolution (used in TLR [44] and TLS [29, 42, 49, 53]), or lock-free/wait-free synchronization [18, 19, 33]. Unfortunately, all of these techniques have problems. Falling back to mutual exclusion requires the programmer to write deadlock-free locking code, thus defeating UTM's goal of programming ease. Using timestamps to resolve conflicts precludes a non-blocking protocol since the oldest thread can block all other threads. Implementing lock-free or wait-free concurrent objects requires considerable programming effort and complicates the resulting program significantly. Therefore, although forward-progress is desirable under some circumstances, it may come with a high cost.

Lastly, UTM does not handle I/O during transaction execution. UTM assumes that all memory operations within a transaction can be rolled back by simply clearing the speculative values in the cache. Many memory operations, however, such as memory mapped I/O, are not cached at all. These operations cannot be rolled back. Once they complete, their effects may immediately be visible. For example, a write to the display buffer may immediately result in a change on the monitor. These operations cannot be run speculatively and inherently require mutual exclusion.

UTM's lack of I/O support during a transaction prevents UTM from being useful in all situations. For example, this limitation may restrict the use of UTM transaction within I/O device driver code. In general, however, many situations require atomicity without invoking I/O. For these situations, UTM provides the easy-to-use, low-overhead transaction primitive.

Although UTM has many limitations, UTM is still a useful system in practice. UTM is the first hardware transaction system to not impose a restrictive transaction

size or length limitation. UTM's unbounded approach enables clean transaction semantics not found in previous hardware transaction designs. Moreover, UTM can be implemented straightforwardly in a modern system with only minor hardware modifications. Since UTM is implemented in hardware, UTM provides the programmer with an easy-to-use transaction primitive with low performance overhead. Although large transactions can potentially incur high performance overhead, experimental results confirm that the high overhead can be efficiently amortized over the fast common case. For these reasons, I contend that UTM represents a significant advance in the design of practical hardware transactional memory systems. UTM achieves easy-to-use transactions without sacrificing practicality, performance, or ease of implementation.

Bibliography

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*, pages 280–298, Ann Arbor, Michigan, June 1984. ACM Press.

[2] C. Scott Ananian and Martin Rinard. Efficient software transactions for object-oriented languages. MIT Computer Science and Artificial Intelligence Laboratory, Unpublished, 2003.

[3] Richard J. Anderson and Joao C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *Proceedings of the 4th annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, San Diego, California, 1992. ACM Press.

[4] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Transactions on Database Systems (TODS)*, 5(2):139–156, June 1980.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms 2nd Edition*. The MIT Press, 2001.

[6] Digital Equipment Corporation. *VAX MACRO and Instruction Set Reference Manual*, November 1996.

[7] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 338–349, Montreal, Canada, June 1998. ACM Press.

[8] The FLEX compiler project. http://flex-compiler.csail.mit.edu.

[9] The GNU Classpath project. Free Software Foundation, Inc., http://classpath.org.

[10] Andrew V. Goldberg and Robert E. Targan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, October 1988.

[11] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA)*, pages 124–131, Stockholm, Sweden, June 1983. ACM Press.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison Wesley, 2000.

[13] Jim Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154, September 1981.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[15] 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. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, Munchen, Germany, June 2004. ACM Press.

[16] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, Anaheim, California, October 2003. ACM Press.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach 2nd Edition.* Morgan Kaufmann, 1996.

[18] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[19] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, November 1993.

[20] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529. IEEE Computer Society, 2003.

[21] Maurice Herlihy, Victor Luchangco, Mark Moir, and Willia m N. Scherer, III. Software transactional memory for dynamic-sized data structur es. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, Boston, Massachusetts, July 2003. ACM Press.

[22] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, San Diego, California, May 1993. ACM Press.

[23] Maurice P. Herlihy and J. Eliot B. Moss. Transactional support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, December 1992.

[24] The IRIX operating system. Silicon Graphics, Inc., http://www.sgi.com/software/irix.

[25] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, California, November 1987.

[26] Thomas F. Knight Jr. An architecture for mostly functional languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP)*, pages 105–112. ACM Press, 1986.

[27] Thomas F. Knight Jr. System and method for parallel processing with mostly functional languages. U.S. Patent 4,825,360, April 25 1989.

[28] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language 2nd Edition.* Prentice Hall, 1988.

[29] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, September 1999.

[30] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(4):579–601, October 1988.

[31] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems (TODS)*, 5(3):354–382, September 1980.

[32] H. T. Kung and John T. Robinson. On optimistic methods of concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, June 1981.

[33] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[34] James Laudon and Daniel Lenoski. System overview of the SGI Origin 200/2000 product line. In *Proceedings of the Spring 1997 42nd IEEE International Computer Conference (COMPCON)*, pages 150–156. IEEE Computer Society, February 1997.

[35] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 148–159, Seattle, Washington, May 1990. ACM Press.

[36] Jos F. Martnez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–29, San Jose, California, October 2002. ACM Press.

[37] MIPS Technologies, Inc., Mountain View, California. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, October 1996.

[38] MIPS Technologies, Inc., Mountain View, California. *MIPS64 Architecture For Programmers Volume II: The MIPS64 Instruction Set*, August 2002.

[39] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface Version 2.0*, March 2002.

[40] Vijay Pai, Parthasarathy Ranganathan, and Sarita Adve. RSIM reference manual, version 1.0. Technical Report 9705, Rice University, August 1997.

[41] Mark S. Papamarcos and Janak H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*, pages 348–354, Ann Arbor, Michigan, June 1984. ACM Press.

[42] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, San Diego, California, June 2003. ACM Press.

[43] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 294–305, Austin, Texas, December 2001.

[44] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, San Jose, California, October 2002. ACM Press.

[45] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)*, pages 439–448. IEEE Computer Society, June 2002.

[46] Peter Rundberg and Per Stenstrm. Speculative lock reordering: Optimistic out-of-order execution of critical sections. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, page 11a, Nice, France, April 2003. IEEE Computer Society.

[47] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, Ottawa, Ontario, Canada, August 1995. ACM Press.

[48] Silicon Graphics, Inc., Mountain View, California. *C Language Reference Manual*, 2003.

[49] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th Annual International Symposium on Computer Architecture (ISCA)*, pages 414–425, S. Margherita Ligure, Italy, June 1995. ACM Press.

[50] Richard M. Stallman. *Using the GNU Compiler Collection.* Free Software Foundation, Boston, Massachusetts, December 2002.

[51] The Standard Performance Evaluation Corporation (SPEC). JVM Client 98 (SPECjvm98). http://www.spec.org/jvm98, 1998.

[52] Richard E. Stearns and Daniel J. Rosenkrantz. Distributed database concurrency controls using before-values. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 74–83. ACM Press, 1981.

[53] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, Vancouver, Canada, June 2000. ACM Press.

[54] Stephen A. Ward and Robert H. Halstead Jr. *Computation Structures.* The MIT Press, 1990.

[55] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[56] Lixin Zhang. URSIM reference manual version 1.0. Technical Report UUCS-00-015, University of Utah, August 2000.

[57] Lixin Zhang. UVSIM reference manual version 0.1. Technical report, University of Utah, 2003.