# Efficient Usable Transactions in Software and Hardware

by

C. Scott Ananian

M.Sc Electrical Engineering and Computer Science,
Massachusetts Institute of Technology, 1999;
B.S.E. Electrical Engineering,
Princeton University, 1997.

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

April XX, 2007



Author .....
Department of Electrical Engineering and Computer Science
April XX, 2007

Certified by .....
Martin Rinard
Professor of Computer Science and Engineering
Thesis Supervisor

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

Name systems & algorithms

Avoid future tense

We versus I

However

Solitary "this"

Long sentences

Cancer of the semicolon

Ack joint work, prior pubs at beginning of relevent chapters

Numbered vs unnumbered sections

Weak topic ¶'s

"very" → "damn"

A number of

Note that

Oblique references

Larger figures

Efficient Usable Transactions in Software and Hardware

by

C. Scott Ananian

Submitted to the
Department of Electrical Engineering and Computer Science
April XX, 2007
In partial fulfillment of the requirements for the Degree of Doctor of Philosophy
in Electrical Engineering and Computer Science.

# Abstract

Transactions are gaining ground as a programmer-friendly means of expressing concurrency, as microarchitecture trends make it clear that parallel systems are in our future. This dissertation presents the design and implementation of three efficient and powerful transaction systems: an object-oriented software-only system, a scalable system using a custom processor extension, and a hybrid of the two systems, obtaining the benefits of both.

The software transaction system implements *strong atomicity*, which ensures that transactions are protected from the influence of non-transactional code. Previous software systems use weaker atomicity guarantees because strong atomicity is presumed to be too expensive. In this thesis strong atomicity is obtained with as little as 15% slowdown for non-transactional code. Compiler analyses can further improve the efficiency of the mechanism, which has been formally verified with the SPIN model-checker.

The low overhead of my software system allows it to be profitably combined with a hardware transaction system to provide fast execution of short and small transactions, while allowing fallback to software for large or complicated transactions. I present UTM, a hardware transactional memory system allowing unbounded virtualizable transactions, and show how a hybrid system can be obtained.

Thesis Supervisor: Martin Rinard
Title: Professor of Computer Science and Engineering

## Acknowledgments

I would like to thank my advisor Martin Rinard for his infinite patience, and for his guidance and support. I would like to thank Tom Knight for pointing me in the direction of transactional memory many years ago, and Charles Leiserson, Bradley Kuszmaul, Krste Asanović, and Sean Lie for the opportunity to collaborate on our hardware transactional memory research.

A list of thanks would not be complete without my "officemates" (real or virtual) who made my years at LCS and CSAIL so enjoyable: Maria-Cristina Marinescu (#2), Radu Rugina (#3), Darko Marinov (#5), Brian Demsky (#6), Alex Salcianu (#7), Karen Zee (#8), Patrick Lam (#10), Viktor Kuncak (#11), Amy Williams (#13), and Angelina Lee (who somehow has no number).

My mother, Cyndy Bernotas, supported me even when she was uncertain I'd ever graduate. Jessica Wong and Kathy Peter put up with my indefinite studenthood. Thank you.

C. Scott Ananian
Cambridge, Massachusetts
April 2007

# Contents

# List of Figures

*LIST OF FIGURES*

[Parallel programming] is hard.

*Teen Talk Barbie*
(apocryphal)

# Chapter 1

# Introduction

How can we fully utilize the coming generation of parallel systems? The primitives available to today's programmers are largely inadequate. ***Transactions*** have been proposed as an alternative, but current implementations are either too limited or too inefficient for general use. This dissertation presents the design and implementation of efficient and powerful transaction systems to help address the challenges posed by current trends in computing hardware.

## 1.1 The rising challenge of multicore systems

Processor technology is finally bumping against its limits: even though transistor quantities continue to grow exponentially, we are now unable to effectively harness those vast quantities of transistors to create speedier single processors. The smaller transistors yield relatively slower signal propagation times, dooming attempts to create a single synchronized processor from all of those resources. Instead, hardware manufacturers are providing tightly integrated ***multicore*** systems which integrate multiple parallel processors on one chip.

The widespread adoption of parallel systems creates problems: how can we ensure that operations occur in the appropriate order? Equivalently, how

can we ensure certain operations occur *atomically*, so that other components of the parallel system only observe data structures in well-defined states?

Atomicity in shared-memory multiprocessors is conventionally provided via mutual-exclusion ***locks*** (see, for example, [67, p. 35]). Although locks are easy to implement using test-and-set, compare-and-swap, or load-linked/store-conditional instructions, they introduce a host of difficulties. Protocols to avoid deadlock when locking multiple objects often involve acquiring the locks in a consistent linear order, making programming with locks error-prone and introducing significant overheads. The granularity of each lock must also be explicitly chosen: locks that are too fine introduce unnecessary space and time overhead, while locks that are too coarse sacrifice attainable parallelism (or may even deadlock). Every access to a shared object must hold some lock protecting that object, regardless of whether another thread is actually attempting to access the same object.

## 1.2 Advantages of transactions

***Transactions*** are an alternative means of providing concurrency control. A transaction can be thought of as a sequence of loads and stores performed as part of a program which either ***commits*** or ***aborts***. If a transaction commits, then all of the loads and stores appear to have run atomically with respect to other transactions. That is, the transaction's operations are not interleaved with those of other transactions. If a transaction aborts, then none of its stores take effect and the transaction may be restarted, with some mechanism to ensure forward progress.

need

By structuring concurrency at a high level with transactions, human programmers no longer ~~have to~~ manage the details required to ensure atomicity. Programmers have difficulty correctly understanding the global interactions between multiple threads, each holding its own set of locks, and a full mental model of the global concurrency structure must be kept in mind when writing synchronization code. The simpler "global atomicity" guaranteed

under the transactional model eliminates potential errors and simplifies the conceptual model of the system, making future modifications safer as well.

The transaction primitives presented in this thesis can exploit optimistic concurrency, provide fault tolerance, and prevent delays by using non-blocking synchronization. Although transactions can be implemented using mutual exclusion (locks), our algorithms will utilize non-blocking synchronization [48, 37, 39, 52, 27] to exploit optimistic concurrency among transactions. Non-blocking synchronization offers a number of advantages; from the system builder's perspective the foremost is fault tolerance. A process that fails or pauses while holding a lock within a critical region can prevent all other processes from ever making progress. In general it is not possible to restore locked data structures to a consistent state after such a failure. Non-blocking synchronization offers a graceful solution, as non-progress or failure of any one thread will not affect the progress or consistency of other threads or the system.

Implementing transactions using non-blocking synchronization offers performance benefits as well. When mutual exclusion is used to enforce atomicity, page faults, cache misses, context switches, I/O, and other unpredictable events may result in delays to the entire system. Non-blocking synchronization allows undelayed processes or processors to continue to make progress. Similarly, in real-time systems, the use of non-blocking synchronization can prevent *priority inversion* in the system [44], although naïve implementations may result in starvation of low-priority tasks (see chapter 7.2 for a discussion).

## 1.3 Unlimited transactions

The ***Transactional memory*** abstraction [46, 36, 65, 58, 63, 35] has been proposed as a general and flexible way to allow programs to read and modify disparate primary memory locations atomically as a single operation, much as a database transaction can atomically modify many records on disk.

***Hardware transactional memory*** (HTM) supports atomicity through architectural means, whereas ***software transactional memory*** (STM) supports atomicity through languages, compilers, and libraries. I will present both software and hardware implementations of the transaction model.

Researchers of both HTM and STM commonly express the opinion that transactions need never touch many memory locations, and hence it is reasonable to put a (small) bound on their size [36, 35].[1] For HTM implementations, they conclude that a small piece of additional hardware—typically in the form of a fixed-size content-addressable memory and supporting logic—should suffice. For STM implementations, some researchers argue additionally that transactions occur infrequently, and hence the software overhead would be dwarfed by the other processing done by an application. In contrast, this thesis will assume that transactions may be of arbitrary size and duration, and that details of the implementation should not be exposed to the programmer of the system.

My goal is to make concurrent and fault-tolerant programming easier, without incurring excessive overhead. This thesis discusses unbounded transactions because neither programmers nor compilers can easily cope when an architecture imposes a hard limit on transaction size. An implementation might be optimized for transactions below a certain size, but must still operate correctly for larger transactions. The size of transactional hardware should be an implementation parameter, like cache size or memory size, which can vary without affecting the portability of binaries.

In chapter 6.4 I will show how a fast hardware implementation for frequent short transactions can gracefully fail over to a software implementation designed to efficiently execute large long-lived transactions. The hybrid approach allows more sophisticated transaction models to be implemented,

[1] For example, [36, section 5.2] states, "Our [HTM] implementation relies on the assumption that transactions have short durations and small data sets"; while the STM described in [35] has quadratic slowdown when transactions touch many objects: performance is $O((R+W)R)$, where R and $W$ are the number of objects opened for reading and writing, respectively.

while allowing a simpler hardware transaction mechanism to provide speed in the common case.

## 1.4 Strong atomicity

Blundell, Lewis, and Martin [13] distinguish between ***strongly atomic*** transaction systems, which protect transactions from interference from "non-transactional" code, and ***weakly atomic*** transaction systems which do not afford this protection. However, all current software transaction systems[2] are weakly atomic, despite the pitfalls thus opened for the unwary programmer, because of the perceived difficulty in efficiently implementing the required protection.

Strong atomicity is clearly preferable, and Blundell et al. point out that programs written for a weakly atomic model (to run on a current software transaction system, say) may deadlock when run under strong atomicity (for example, on a hardware transaction system). The transaction systems considered in this dissertation will preserve the correct atomic behavior of transactions even in the face of unsynchronized accesses from outside the transaction.

## 1.5 Summary of contributions

In summary, this thesis makes the following contributions:

- We provide efficient implementations of *strongly atomic* transaction primitives to enable their general use. In particular, our technique imposes as little as 15% overhead on non-transactional code in a software-only system.

- The transaction primitives presented in this thesis can exploit optimistic concurrency, provide fault tolerance, and prevent delays by

[2] For example, [30].

using non-blocking synchronization.

- Unlike some previous work, this thesis will assume that transactions may be of arbitrary size and duration, and that details of the implementation should not be exposed to the programmer of the system.

- I will present both software and hardware implementations of the transaction model. The software transaction system runs real programs written in Java; I will discuss the practical implementation details encountered. The hardware transaction systems require only small changes to the processor core and cache.

- I will show how a fast hardware implementation for frequent short transactions can gracefully fail over to a software implementation designed to efficiently execute large long-lived transactions.

In the next chapter I will provide some concurrent programming examples that illustrate the limitations of current lock-based methodologies. I will also provide examples illustrating the uses (some novel) of a transaction system, and conclude with a brief caution about the current limits of transactions.

In chapter 3, I will present the design of an efficient software-only implementation of transactions. Software-only transactions can be implemented on current hardware, and can easily accommodate many different transaction and nesting models. Software transactions excel at certain long-lived transactions, where the overhead is small compared to the transaction execution time. I will conclude by presenting a microbenchmark that demonstrates the performance limits of the design.

In chapter 4, I will discuss the practical implementation of chapter 3's design. I will present details of the compiler analyses and transformations performed, as well as solutions to problems that arise when implementing Java. I will then present benchmark results using real applications, discuss

the benefits and limitations revealed, and describe how the limits could be overcome.

In chapter 5 I will consider one such limitation in depth, describing how large arrays fit into an object-oriented transaction scheme. We present a potential solution to the problem based on fast functional arrays.

In chapter 6, I will present hardware that enables very fast short transactions. The transaction model is more limited, but short committing transactions may execute with no overhead. The additional hardware is small and easily added to current processor and memory system designs.

At the end of the chapter, I present a ***hybrid*** transaction implementation that builds on the strengths of simple hardware support while allowing software fallback to support a robust and capable transaction mechanism. Unlike the extended hardware scheme, the transaction model is still easy to change and update; the hardware primarily supports fast small transactions and conflict checking in the common case. In the following chapter, we discuss some ways compilers can further optimize software and hybrid transaction systems. These opportunities may not be available to pure-hardware implementations.

In chapter 7, I discuss remaining challenges to the use of ubiquitous transactions for synchronization, and present some ideas toward solutions. Chapter 8 discusses related work, and our final chapter summarizes our findings and draws conclusions.

Name coauthors & published papers.

You have a hardware or a software problem.

*Service manual for Gestetner 3240*

# Chapter 2

# Examples of Transactional Programming

Before diving into the design of an efficient transaction system, I will motivate the transactional programming model by presenting four common scenarios that are needlessly difficult using lock-based concurrency; I will then present four novel applications that a transactional model facilitates. To ground the discussion in reality, I conclude by enumerating a few cases where transactions may *not* be the best solution.

## 2.1 Four old things you can't (easily) do with locks

Locks engender poor modularity and composability, an inability to deal gracefully with asynchronous events, and fragile and complex safety protocols that are often expressed externally to their implementations. These limitations of locks are well-known [38].

### 2.1.1 Tweak performance with localized changes

Preventing deadlocks and races requires global protocols and non-local reasoning. It is not enough to simply specify a lock of a certain granularity protecting certain data items; the order or circumstances in which the lock

may be acquired and released must also be specified globally, in the context of all locks in the system, in order to prevent deadlocks or unexpected races. This prevents the programmer from easily tuning the system using localized changes: every small change must be re-verified against the protocols specified in the whole-program context in order to prevent races and/or deadlock.

Furthermore, this whole-program protocol is not typically expressed directly in code: with common programming languages, acquire/release ordering and guarantees must be expressed externally, often as comments in the source code that easily drift out of sync with the implementation. For example, in [5] we counted the comments in the Linux filesystem layer, and found that about 15% of these relate to locking protocols; often describing global invariants of the program that are difficult to verify. Many reported kernel bugs involve races and deadlocks.

### 2.1.2 Atomically move data between thread-safe containers

Another common programming pitfall with locks is their ***non-composability***. For example, given two thread-safe container classes implemented with locks, it is impossible to safely compose the *get* function of one with the *put* function of the another to create an atomic move. We must peek inside the implementation of the containers to synthesize an appropriate locking mechanism for such an action–for example, to acquire the appropriate container, element, or other locks on *both* containers—and even then, we need to resort to some global lock ordering to guard against deadlock. Modularity must be broken in order to synthesize the appropriate composed function, if it is possible at all.

In the Jade programming language, Martin Rinard presented a partial solution using "implicitly synchronized" objects [59, p14]. Lock acquisition for each module is exposed in the module's API as an executable "access declaration." Operation composition is accomplished by creating an access declaration for the composed operation that invokes the appropriate access

declarations for the components. The runtime system orders the lock acquisitions to prevent deadlock. This process suffices for conservative mutual exclusion, but it is limited in its ability to express alternative operation orders that preserve atomicity of the composed operation while overlapping its components.

Transactions do not suffer from the composability problem [31]. Because transactions only specify the atomicity properties, not the locks required, the programmer's job is eased and implementations are free to order operations in any way that preserves atomicity.

### 2.1.3 Create a thread-safe double-ended queue

Herlihy suggests creating a thread-safe double-ended queue using locks as "sadistic homework" for computer science students [38]. Although double-ended queues are a simple data structure, creating a scalable locking protocol is a non-trivial exercise: one wants dequeue and enqueue operations to complete concurrently when the ends of the queue are "far enough" apart, while safely handling the interference in the small-queue case. In fact, the solution to this assignment was a publishable result, as Michael and Scott demonstrated in 1996 [54].

The simple "one lock" solution to the double-ended queue problem, ruled out as unscalable in the locking case, is scalable and efficient for non-blocking transactions.

### 2.1.4 Handle asynchronous exceptions

Properly handling asynchronous events is difficult with locks, because it is impossible to safely go off to handle the event while holding an arbitrary set of locks—and it is impossible to safely drop the locks. The solution implemented in the Real-Time Specification for Java and similar systems is to generally forbid asynchronous events within locked regions, allowing the programmer to explicitly specify certain points within the region at which

execution can be interrupted, dropping all locks in order to do so. Maintaining the correctness in the face of even explicitly declared interruption points is still difficult.

Transactional atomic regions handle asynchronous exceptions gracefully: the transaction is aborted to allow an event to occur.

## 2.2 Four new things transactions make easy

I present four examples in this section, illustrating how transactions can support fault tolerance and backtracking, simplify locking, and provide a more intuitive means for specifying thread-safety properties. I will first examine a destructive traversal algorithm, showing how a transaction implementation can be treated as an exception-handling mechanism. A variant of this mechanism can be used to implement backtracking search. Using a network flow example, I will then show how the transaction mechanism can be used to simplify the locking discipline required when synchronizing concurrent modifications to multiple objects. Finally, I show an existing race in the Java standard libraries (in the class `java.lang.StringBuffer`). "Transactification" of the existing class corrects this race.

### 2.2.1 Destructive traversal

Many recursive data structures can be traversed without the use of a stack using pointer reversal. This technique is widely used in garbage collectors, and was first demonstrated in this context by Schorr and Waite [62]. An implementation of a pointer-reversal traversal of a simple singly-linked list is shown in Figure 2.1.

The `traverse()` function traverses the list, visiting nodes in order and then reversing the `next` pointer. When the end of the list is reached, the reversed links are traversed to restore the list's original state.

Of course, I have chosen the simplest possible data structure here, but the technique works for trees and graphs—and the reader may mentally

```
// destructive list traversal.
void traverse(List l) {
  List last = null, t;

  /* zip through the list, reversing links */
  for (int i=0; i<2; i++) {
    do {
      if (i==0) visit(l); // visit node
      t = l.next;
      l.next = last;
      last = l;
      l = t;
    } while (l!=null);
    l = last;
    // now do again, backwards. (restoring links)
  }
}
```

Figure 2.1: Destructive linked-list traversal.

substitute their favorite hairy update on a complicated data structure.

In normal execution, the data structure is left complete and intact after the operation. But imagine that an exception or fault occurs inside the `visit()` method at some point during the traversal: an assertion fires, an exception occurs, the hardware hiccups, or a thread is killed. Control may leave the `traverse()` method, but the data structure is left in shambles. What is needed is some exception-handling procedure to restore the proper state of the list. This can be manually coded with Java's existing `try/catch` construct, but the exception-handling code must be tightly-coupled to the traversal if it is going to undo the list mutations.

Instead, I can provide a non-deterministic choice operator, `try/else`, and write the recovery code at a higher level as:

```
try {
  traverse(list);
} else { // try-else construct
  throw new Error();
}
```

The `try/else` block appears to make a non-deterministic choice between executing the `try` or the `else` clause, depending on whether the `try` would

```
int expr() {
   try {
     return sum();
   } else {
     return difference();
   }
}
int sum() {
   int a = number();
   eat(ADD);
   int b = number();
   return a+b;
}
int difference() {
   int a = number();
   eat(MINUS);
   int b = number();
   return a-b;
}
```

Figure 2.2: A simple backtracking recursive-decent parser.

succeed or not. This can be straightforwardly implemented with a transaction around the traversal, always initially attempting the `try`. Exceptions or faults cause the transaction to abort; when it does so all the heap side-effects of the `try` block disappear.

### 2.2.2 Backtracking search

Introducing an explicit `fail` statement allows us to use the same `try/else` to facilitate backtracking search. Backtracking search is used to implement practical[1] regular expressions, parsers, logic programming languages, Scrabble-playing programs [7], and (in general) any problem with multiple solutions or multiple solution techniques.

As a simple example, let us consider a recursive-decent parser such as that shown in Figure 2.2. We don't know whether to apply the `sum()` or `difference()` production until after we've parsed some common left prefix. We can use backtracking to attempt one rule (`sum`) and `fail` out of it

[1] As opposed to the limited regular expressions demonstrated in theory classes which are always neatly compiled to deterministic finite automata [22].

inside the `eat()` method, in the process undoing any data structure updates performed on this path, and then attempt the other possible production.

### 2.2.3 Network flow

Let's now turn our attention now to parallel codes, the more conventional application of transaction systems. Consider a serial program for computing network flow (see, for example, [18, Chapter 26]). The inner loop of the code pushes flow across an edge by increasing the "excess flow" on one vertex and decreasing it by the same amount on another vertex. One might see the following Java code:

```
void pushFlow(Vertex v1, Vertex v2, double flow) {
  v1.excess += flow; /* Move excess flow from v1 */
  v2.excess -= flow; /* to v2.                   */
}
```

To parallelize this code, one must preclude multiple threads from modifying the excess flow on those two vertices at the same time. Locks provide one way to enforce this mutual exclusion:

```
void pushFlow(Vertex v1, Vertex v2, double f) {
  Object lock1, lock2;
  if (v1.id < v2.id) {       /* Avoid deadlock. */
    lock1 = v1; lock2 = v2;
  } else {
    lock1 = v2; lock2 = v1;
  }
  synchronized(lock1) {
    synchronized(lock2) {
      v1.excess += f; /* Move excess flow from v1 */
      v2.excess -= f; /* to v2.                   */
    } /* unlock lock2 */
  } /* unlock lock1 */
}
```

This code is surprisingly complicated and slow compared to the original. Space for each object's lock must be reserved. To avoid deadlock, the code must acquire the locks in a consistent linear order, resulting in an unpredictable branch in the code. In the code shown, I have required the programmer to insert an `id` field into each vertex object to maintain a

total ordering. The time required to acquire the locks may be an order of magnitude larger than the time to modify the excess flow. What's more, all of this overhead is rarely needed! For a graph with thousands or millions of vertices, the number of threads operating on the graph is likely to be less than a hundred. Consequently, the chances are quite small that two different threads actually conflict. Without the locks to implement mutual exclusion, however, the program would occasionally fail.

Software transactions (and some language support) allow the programmer to parallelize the original code using an `atomic` keyword to indicate that the code block should appear to execute atomically:

```
void pushFlow(Vertex v1, Vertex v2, double flow) {
  atomic { /* Transaction begin. */
    v1.excess += flow; /* Move excess flow from v1 */
    v2.excess -= flow; /* to v2.                   */
  } /* Transaction end. */
}
```

This `atomic` region can be implemented as a transaction, and with an appropriately non-blocking implementation, it will scale better and execute faster than the locking version [5, 30, 27, 52, 36, 63]. From the programmer's point of view, I have also eliminated the convoluted locking protocol which must be observed rigorously everywhere the related fields are accessed if deadlock and races are to be avoided.

Further, I can implement `atomic` using the `try/else` exception-handling mechanism I have already introduced:

```
for (int b=0; ; b++) {
  try {
    // atomic actions
  } else {
    backOff(b);
    continue;
  }
  break; // success!
}
```

I non-deterministically choose to execute the body of the `atomic` block if and only if it will be observed by all to execute atomically. The same lin-

guistic mechanism I introduced for fault tolerance and backtracking provides atomic regions for synchronization as well.

### 2.2.4 Bug fixing

The existing ***monitor synchronization*** methodology for Java, building on such features in progenitors such as Emerald [12, 45],[2] implicitly associates an lock with each object. Data races are prevented by requiring a thread to acquire an object's lock before touching the object's shared fields. However, the lack of races is not sufficient to prevent unanticipated parallel behavior.

Flanagan and Qadeer [20] demonstrated this insufficiency with an actual bug they discovered in the Sun JDK 1.4.2 Java standard libraries. The `java.lang.StringBuffer` class, which implements a mutable string abstraction, is implemented as follows:

```
public final class StringBuffer ... {
  private char value[];
  private int count;
  ...
  public synchronized
  StringBuffer append(StringBuffer sb) {
    ...
A:  int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length)
      expandCapacity(newcount);
    // next statement may use stale len
B:  sb.getChars(0, len, value, count);
    count = newcount;
    return this;
  }
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }
}
```

The library documentation indicates that the methods of this class are meant to execute atomically, and the `synchronized` modifiers on the methods are meant to accomplish this.

[2] See chapter 8.4.

However, the `append()` method is *not* atomic. Another thread may change the length of the parameter `sb` (by adding or removing characters) between the call to `sb.length()` at label A and the call to `sb.getChars(...)` at label B. This non-atomicity may cause incorrect data to be appended to the target or a `StringIndexOutOfBoundException` to be thrown. Although the calls to `sb.length()` and `sb.getChars()` are individually atomic, they do not compose to form an atomic implementation of `append()`.

Note that replacing `synchronized` with `atomic` in this code gives us the semantics we desire: the atomicity of nested `atomic` blocks is guaranteed by the atomicity of the outermost block, ensuring that the entire operation appears atomic.

Both the network flow example and this `StringBuffer` example require synchronization of changes to more than one object. Monitor synchronization is not well-suited to this task. Atomic regions implemented with transactions can be used to simplify the locking discipline required to synchronize multi-object mutation and provide a more intuitive specification for the desired concurrent properties. Further, the `StringBuffer` example shows that simply replacing `synchronized` with `atomic` provides a alternative semantics that may in fact correct existing synchronization errors. For many Java programs, the semantics of `atomic` and `synchronized` are identical; see chapter 7.3.

## 2.3 Some things we still can't (easily) do

The transaction mechanism presented here is not a universal replacement for all synchronization. In particular, transactions cannot replace mutual exclusion required to serialize I/O, although the needed locks can certainly be built with transactions. The challenge of integrating I/O within a transactional context is discussed in chapter 7.4. However, large programs—the Linux kernel, for example—have been written such that locks are never held across context switches or I/O operations. Transactions provide a complete

solution for this limited synchronization.

Blocking producer-consumer queues, or other options that require a transaction to wait upon a condition variable, may introduce complications into a transaction system: transactions cannot immediately retry when they fail, but instead must wait for some condition to become true. Chapter 4.4.4 describes some solutions to this problem, ranging from naïve (keep retrying and aborting with exponential backoff until the condition is finally true) to clever; the clever solutions require additional transaction machinery.

Easy things should stay easy,
hard things should get easier,
and impossible things should get
hard.

*Motto for Perl 6 development*

# Chapter 3

# Designing a software transaction system

In this chapter I will detail the design of a high-performance software transaction system. I will first present our methodology for isolating likely transactions from our benchmarks. I will then describe the system goals, and use quantitative data from our benchmarks as I explain the design choices. Then I will formalize an implementation meeting those goals in the modeling language Promela. The correctness of this implementation can be model-checked using the SPIN tool; the details of this effort are in Appendix A.

After outlining the design in this chapter, chapter 4 will discuss a practical implementation.

## 3.1 Finding transactions

One of the difficulties of proposing a novel language feature is the lack of benchmarks for its evaluation. Although there is no existing body of code that uses transactions, there is a substantial body of code that uses Java (locking) synchronization. This thesis will utilize the FLEX compiler [3] to substitute `atomic` blocks (methods) for `synchronized` blocks (methods) in order to evaluate the properties Java transactions are likely to have. Note

that the semantics are not precisely compatible: the existing Java memory model allows unsynchronized updates to shared fields to be observed within a synchronized block, while such updates will never be visible to a transaction expressed with an `atomic` block.[1] Despite the differences in semantics, the automatic substitution of `atomic` for `synchronized` does, in fact, preserve the correctness of the benchmarks I examine here.

The initial results of this chapter explore the implications of exposing the transaction mechanism to user-level code through a compiler. I compiled the SPECjvm98 benchmark suite with the FLEX Java compiler, modified to turn synchronized blocks and methods into transactions, in order to investigate the properties of the transactions in such "automatically converted" code. FLEX performed method cloning to distinguish methods called from within transactions, and implemented nested locks as a single transaction around the outermost.[2] I instrumented this transformed program to produce a trace of memory references and transaction boundaries for analysis. I found both large transactions (involving millions of cache lines) and frequent transactions (up to 45 million of them).

The SPECjvm98 benchmark suite represents a variety of typical Java applications that use the capabilities of the Java standard library. Although the SPECjvm98 benchmarks are largely single-threaded, since they use the thread-safe Java standard libraries, they contain synchronized code which is transformed into transactions. Because in this evaluation I am looking at transaction properties only, the multithreaded `227_mtrt` benchmark is identical to its serialized version, `205_raytrace`. For consistency, I present only the latter.

Figure 3.1 shows the raw sizes and frequency of transactions in the transactified SPECjvm98 suite. Figure 3.2 proposes a taxonomy for Java applications with transactions, grouping the SPECjvm98 applications into quad-

[1] The proposed revision of the Java memory model [50] narrows the semantic gap, however I do not plan to treat `volatile` fields in this work. See chapter 7.3 for more details.

[2] See chapter 4.2 for more details on this transformation.

| program | total memory ops | transactions | transactional memory ops | biggest transaction |
|---|---|---|---|---|
| 201_compress | 2,981,777,890 | 2,272 | <0.1% | 2,302 |
| 202_jess | 405,153,255 | 4,892,829 | 9.1% | 7,092 |
| 205_raytrace | 420,005,763 | 4,177 | 1.7% | 7,149,099 |
| 209_db | 848,082,597 | 45,222,742 | 23.0% | 498,349 |
| 213_javac | 472,416,129 | 668 | 99.9% | 118,041,685 |
| 222_mpegaudio | 2,620,818,169 | 2,991 | <0.1% | 2,281 |
| 228_jack | 187,029,744 | 12,017,041 | 34.2% | 14,266 |

Figure 3.1: Transactification of SPECjvm98 benchmark suite: resulting transaction counts and sizes, compared to total number of memory operations (loads and stores). These are full input size runs.

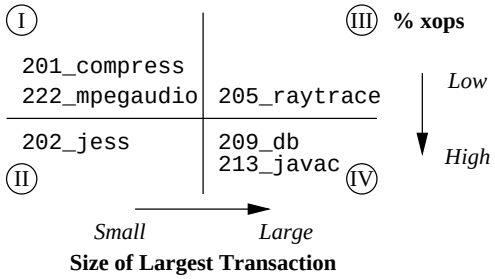



Figure 3.2: Classification of SPECjvm98 benchmarks into quadrants based on transaction properties.

rants based on the number and size of the transactions that they perform. Applications in Quads II and IV require an efficient transaction implementation, because they contain many transactional operations. Quads III and IV contain at least some very large transactions, which pose difficulties for currently-proposed hardware transactional memory schemes. We will now examine the benchmarks in each quadrant to determine why its program logic caused it to be classified in that quadrant.

Quad I applications perform few (up to about 2000) small transactions. These applications include 201_compress, an implementation of gzip compression, and 222_mpegaudio, an MP3 decoder. Both of these applications perform inherently serial tasks. They perform quite well with locks, and would likely execute with acceptable performance even with a naïve software implementation of transactions, as long as the impact on non-transactional operations was minimal.

Quad II applications perform a large number of small transactions. The expert system 202_jess falls in this category, as do small input sizes of 209_db, a database. These benchmarks perform at least an order of magnitude more transactions than Quad I applications, and all of the transactions are small enough to comfortably fit the known hardware transactional memory schemes [36, etc], if one were to be implemented.

Quad III includes 205_raytrace, a ray-tracing renderer. A small number of transactions are performed, but they may grow very large. Existing bounded hardware transactional schemes will not suffice. The large transactions may account for a large percentage of total memory operations, which may make software schemes impractical.

Finally, Quad IV applications such as 209_db and the 213_javac Java compiler application perform a large number of transactional memory operations with at least a few large transactions.

The 213_javac Java compiler application and the large input size of the 209_db benchmark illustrate that some programs contain *extremely* large transactions. When 213_javac is run on its full input set, it contains 4

very large transactions, each of which contains over 118 million transactional memory operations. Closer examination reveals that the method `Javac.compile()`, which implements the entire compilation process, is marked as synchronized: the programmer has explicitly requested that the entire compilation occur atomically.

The large transactions in Quad III and IV may be, as in this case, a result of overly-coarse–grained locking, but our goal is to relieve the programmer from the burden of specifying correct atomic regions of smaller granularity. Performance may benefit from narrowing the atomic regions, but execution with coarse regions should be possible and not prohibitively slow.

The 209_db benchmark suffers from a different problem: at one point the benchmark atomically scans an index vector and removes an element, creating a potentially large transaction if the index is large. The size of this index is correlated in these benchmarks with the input size, but it need not be: a large input could still result in a small index, and (to some degree) vice-versa.

A similar situation arises in the `java.lang.StringBuffer` code shown in chapter 2.2.4: a call to the synchronized `sb.getChars()` method means that the size of the transaction for this method will grow like the length of the parameter `sb`. In other words, the transaction can be made arbitrarily large by increasing the length of `sb`; or, equivalently, there is no bound on transaction size without a bound on the size of the string `sb`.

Any scheme that allows the programmer free reign over specifying desired transaction and/or atomicity properties will inevitably result in some applications in each of these categories. Existing hardware transactional memory schemes only efficiently handle relatively short-lived and small transactions (Quad I or II), although for these they are very efficient. Object-based transaction systems can asymptotically approach that efficiency for very long-lived transactions; the existence of such is shown in Figure 3.3, which plots the distribution of transaction sizes in SPECjvm98 on a semi-log

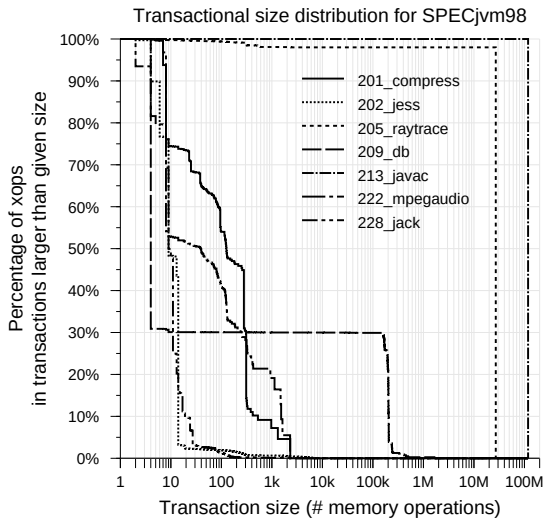



Figure 3.3: Distribution of transaction size in the SPECjvm98 benchmark suite. Note that the x-axis uses a logarithmic scale.

scale.

These initial results indicate that real applications can be transactified with modest effort, yielding significant gains in concurrency. In other work [5] we have shown that a factor of 4 increase in concurrency can be obtained by doing nothing more than converting locks to transactions. Since the transactified applications may contain large transactions, proposed hardware support for transactions is inadequate.

## 3.2 Designing efficient transactions

In this section I briefly describe some desired properties of the software transaction system of this thesis. Where possible I will justify these desiderata using quantitative data obtained from analyses of the SpecJVM98 benchmarks, which I implemented using the Flex Java compiler framework.

### 3.2.1 Weak vs. strong atomicity

As previously discussed in chapter 1.4, ***strongly atomic*** transaction systems protect transactions from interference from "non-transactional" code, while ***weakly atomic*** transaction systems do not afford this protection. Consider unsynchronized code directly altering the length field of `StringBuffer` (chapter 2.2.4). In a weakly atomic system this will cause the `atomic` `StringBuffer.append()` method to appear non-atomic. Current software transaction systems implement only weak atomicity because of the assumed expense of implementing strong atomicity. The system we will design in this chapter will achieve strong atomicity without adding excessive overhead, so that correct operation is assured even despite concurrent non-transactional operations on locations involved in a transaction.

### 3.2.2 Object-oriented vs. flat TM

This transaction system, unlike most current proposals [30, 36] (including the hardware system presented in chapter 6), uses an "object-oriented" design. Much contemporary research is focused on implementing a flat (transactional) memory abstraction in software, primarily because this solves some of the issues I will present in chapter 5. However, the object-oriented approach offers several benefits:

**Efficient execution of long-running transactions.** As discussed briefly in Chapter 8.2 and 8.3, flat word-oriented transaction schemes require overhead proportional to the number of words read/written in the transaction, even if these locations had been accessed before inside the transaction. Object-oriented schemes impose a cost proportional to the number of *objects* touched by the transaction — but once the cost of "opening" those objects is paid, the transaction can continue to work indefinitely upon those objects without paying any further penalty. Object-oriented schemes are thus seen to be more efficient for *long-running transactions*.

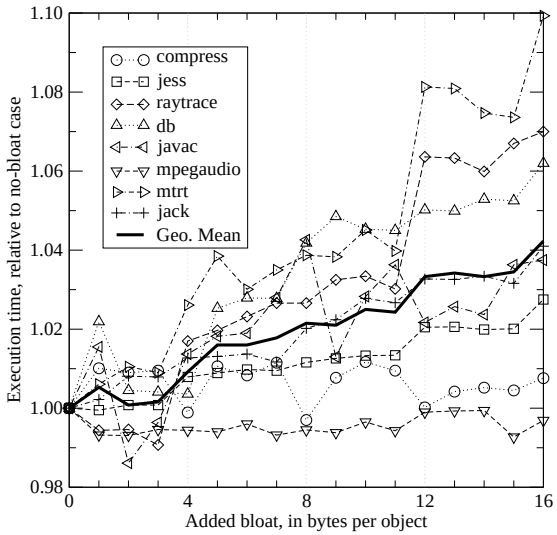



Figure 3.4: Application slowdown with increasing object bloat for the SPECjvm98 benchmark applications.

**Preservation of optimization opportunities.** Furthermore, transaction-local objects can be identified (statically or dynamically) and creation/updates to these objects can be done without any transaction tax at all. Word oriented schemes discard the high-level information required to implement these optimizations.

I believe that the problems with previous object-oriented schemes can be solved while preserving the inherent benefits of an object-oriented approach, and the current thesis presents one such solution.

### 3.2.3 Tolerable limits for object expansion

I will need to add some additional information to each object to track transaction state. I measured the slowdown caused by various amounts of object "bloat" to determine reasonable bounds on the size of this extra information. Figure 3.4 presents these results for the SPECjvm98 applications; I determined that two words (eight bytes) of additional storage per object

| program | transactional memory ops | transactional stores % |
|---|---|---|
| 201_compress | 50,029 | 26.2% |
| 202_jess | 36,701,037 | 0.6% |
| 205_raytrace | 7,294,648 | 23.2% |
| 209_db | 195,374,420 | 6.3% |
| 213_javac | 472,134,289 | 22.9% |
| 222_mpegaudio | 41,422 | 18.6% |
| 228_jack | 63,912,386 | 17.0% |

Figure 3.5: Comparison of loads and stores inside transactions for the SPECjvm98 benchmark suite, full input runs.

would not impact performance unreasonably. This amount of bloat causes a geometric mean of 2% slowdown on these benchmarks.

### 3.2.4 Designing for reads vs. writes

I also measured the number and types of reads and writes for the SpecJVM98 benchmarks. Figure 3.5 shows that transactional reads typically outnumber transactional writes by at least 4 to 1; in some cases reads outnumber writes by over 100 to 1.[3] It is worthwhile, therefore, to make reads more efficient than writes. In particular, since the flag-overwrite technique discussed in chapter 3.2.5 requires us to allocate additional memory to store the "real" value of the field, I wish to avoid this process for transactional reads, reserving the extra allocation effort for transactional writes.

### 3.2.5 The big idea: waving FLAGs

I would like non-transactional code to execute with minimal overhead, but transactions should still appear atomic to non-transactional code. My basic

[3] The typical ratio roughly matches the 3:1 average observed in Hennessy and Patterson [32, pp. 105, 379].

mechanism is loosely based on the distributed shared memory implementation of Scales and Gharachorloo [61]. I pick a special "flag" value, and "cross-out" locations currently involved in a transaction by overwriting them with the flag value. Reading or attempting to overwrite a flagged value will indicate to non-transactional code that exceptional processing is necessary; all other non-transactional operations proceed as usual.

Note that this technique explicitly allows safe access to fields involved in a transaction from non-transactional code, which is another design goal of the system. This eases "transactification" of legacy code (but see chapter 7.3!).

## 3.3 Specifying the basic mechanism

I now present an algorithm that has these desired properties. My algorithms will be completely non-blocking, which allows good scaling and proper fault-tolerant behavior; one faulty or slow processor cannot hold up the remaining good processors.

I will implement the synchronization required by my algorithm using load-linked/store-conditional instructions. I require a particular variant of these instructions that allows the location of the load-linked to be different from the target of the store-conditional; this variant is supported on many chips in the PowerPC processor family, although it has been deprecated in version 2.02 of the PowerPC architecture standard.[4] This disjoint location

[4] Version 2.02 of the PowerPC architecture standard says, "If a reservation exists but the storage location specified by the `stwcx.` is not the same as the location specified by the *Load And Reserve* instruction that established the reservation... it is undefined whether [the operand is] stored into the word in storage addressed by [the specified effective address]" and states that the condition code indicating a successful store is also undefined in this circumstance [42, p 25]. The user manual for the MPC7447/7457 ("G4") PowerPC chips states, however, "The `stwcx.` instruction does not check the reservation for a matching address. The `stwcx.` instruction is only required to determine whether a reservation exists. The `stwcx.` instruction performs a store word operation only if the reservation exists," [21, Chapter 3.3.3.6] which is the behavior we require. I believe

capability is essential to allow us to keep a finger on one location while modifying another: a poor man's "Double Compare And Swap" instruction.

I will describe my algorithms in the Promela modeling language [41], which I used to allow mechanical model checking of the race-safety and correctness of the design. Portions of the model have been abbreviated for this presentation; the full Promela model is given in Appendix A, along with a brief primer on Promela syntax and semantics.

### 3.3.1 Object structures

Figure 3.6 illustrates the basic data structures of my software transaction implementation. Objects are extended with two additional fields. The first field, `versions`, points to a singly-linked list of object versions. Each one contains field values corresponding to a committed, aborted, or in-progress transaction, identified by its `owner` field. There is a single unique transaction object for each transaction.

The other added field, `readers`, points to a singly-linked list of transactions that have read from this object. Committed and aborted transactions are pruned from this list. The `readers` field is used to ensure that a transaction does not operate with out-of-date values if the object is later written non-transactionally.

There is a special flag value, here denoted by `FLAG`. It should be an uncommon value, i.e. not a small positive or negative integer constant, nor zero. In my implementation, I have chosen the byte `0xCA` to be our flag value, repeated as necessary to fill out the width of the appropriate type. The semantic value of an object field is the value in the original object

version 1.10 of the PowerPC Architecture specification required this behavior, although I have not been able to confirm this. The Cell architecture specification follows version 2.02 of the PowerPC specification, although it adds cache-line reservation operations that can also be used to implement our algorithm for reasonably-sized objects aligned within cache lines; see [69] for an implementation of Java I created that observes the appropriate alignments.

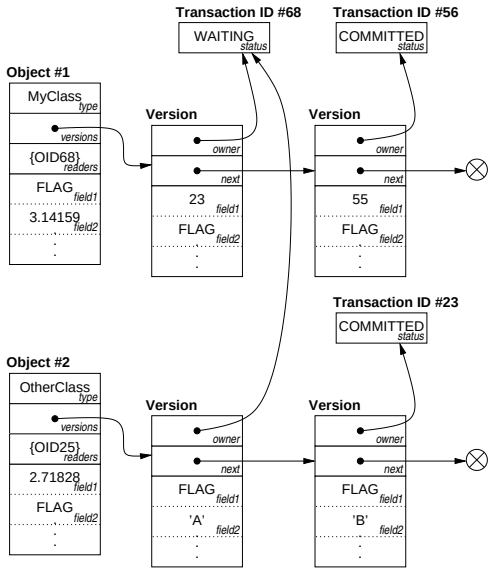



Figure 3.6: Implementing software transactions with version lists. A transaction object consists of a single field *status*, which indicates if it has COMMITTED, ABORTED, or is WAITING. Each object contains two extra fields: *readers*, a singly-linked list of transactions that have read this object; and *versions* a linked list of version objects. If an object field is FLAG, then the value for the field is obtained from the appropriate linked version object.

```
#define FLAG 202 /* special value to represent 'not here' */

typedef Object {
  byte version;
  byte readerList; /* we do LL and CAS operations on this field */
  pid fieldLock[NUM_FIELDS]; /* we do LL operations on fields */
  byte field[NUM_FIELDS];
};
typedef VersiOn { /* 'Version' misspelled because SPIN #define's it. */
  byte owner;
  byte next;
  byte field[NUM_FIELDS];
};
typedef ReaderList {
  byte transid;
  byte next;
};
mtype = { waiting, committed, aborted };
typedef TransID {
  mtype status;
};
```

Figure 3.7: Declaring objects with version lists in Promela. Note that we are using the byte datatype to encode pointers. The fieldLock field assists in the implementation of the load-linked/store-conditional pair of operations in Promela.

structure, *unless that value is FLAG* , in which case the field's value is the value of the field in the first committed transaction in the object's version list. A "false flag" occurs when the application wishes to "really" store the value FLAG in a field; this is handled by creating a fully-committed version attached to the object and storing FLAG in that version as well as in the object field.

Figure 3.7 declares these object structures in Promela.

### 3.3.2 Operations

I support transactional read/write and non-transactional read/write as well as transaction begin, transaction abort, and transaction commit. Transaction begin simply involves the creation of a new transaction identifier object. Transaction commit and abort are simply compare-and-swap operations that

atomically set the transaction object's status field appropriately if and only if it was previously in the WAITING state. The simplicity of commit and abort are appealing: my algorithm requires no complicated processing, delay, roll-back or validate procedure to commit or abort a transaction.

Note that we could support non-transactional read and write (that is, reads and writes that take place outside of any transaction) by creating a new very short transaction that encloses only the single read or write. Non-transactional accesses to objects can be very frequent, however, so I provide more efficient implementations with the same semantics.

I will present the operations one-by-one.

### Non-transactional read

The readNT function does a non-transactional read of field f from object o, putting the result in $v$. In the common case, the only overhead is to check that the read value is not FLAG. However, if the value read *is* FLAG, we copy back the field value from the most-recently committed transaction (aborting all other transactions) and try again. The copy-back procedure will notify us if this is a "false flag", in which case the value of this field really is FLAG. We pass the kill_writers constant to the copy-back procedure to indicate that only transactional writers need be aborted, not transactional readers. All possible races are confined to the copy-back procedure, which I will describe on page 48.

The top of Figure 3.8 specifies the non-transactional read operation in Promela.

### Non-transactional write

The writeNT function does a non-transactional write of new value nval to field f of object o. For correctness, we need to ensure that the reader list is empty before we do the write. I implement this with a load-linked/store-conditional pair, which is modelled in Promela slightly differently, ensuring

```
inline readNT(o, f, v) {
  do
  :: v = object[o].field[f];
     if
     :: (v!=FLAG) -> break /* done! */
     :: else
     fi;
     copyBackField(o, f, kill_writers, _st);
     if
     :: (_st==false_flag) ->
        v = FLAG;
        break
     :: else
     fi
  od
}

inline writeNT(o, f, nval) {
  if
  :: (nval != FLAG) ->
     do
     :: atomic {
          if /* this is a LL(readerList)/SC(field) */
          :: (object[o].readerList == NIL) ->
             object[o].fieldLock[f] = _thread_id;
             object[o].field[f] = nval;
             break /* success! */
          :: else
          fi
        }
        /* unsuccessful SC */
        copyBackField(o, f, kill_all, _st)
     od
  :: else -> /* create false flag */
     /* implement this as a short *transactional* write. */
     /* start a new transaction, write FLAG, commit the */
     /* transaction; repeat until successful. */
     /* Implementation elided. */
     ...
  fi;
}
```

Figure 3.8: Promela specification of non-transactional read and write operations.

that our write only succeeds so long as the reader list remains empty.[5] If it is not empty, we call the copy-back procedure (as in `readNT`), passing the constant `kill_all` to indicate that both transactional readers and writers should be aborted during the copy-back. The copy-back procedure leaves the reader list empty.

If the value to be written is actually the `FLAG` value, things get a little bit trickier. This case does not occur often, and so the simplest correct implementation is to treat this non-transactional write as a short transactional write, creating a new transaction for this one write, and attempting to commit it immediately after the write. This is slow, but adequate for this uncommon case.

The bottom of Figure 3.8 specifies the non-transactional write operation in Promela.

### Field Copy-Back

Figure 3.9 presents the field copy-back routine. We create a new version owned by a pre-aborted transaction which serves as a reservation on the head of the version list. We then write to the object field with a load-linked/store-conditional pair if and only if our version is still at the head of the versions list.[6] This addresses the major race possible in this routine.

### Transactional Read

A transactional read is split into two parts. Before the read, we must ensure that our transaction is on the reader list for the object. This is straightforward to do in a non-blocking manner as long as we always add ourselves to the head of the list. We must also walk the versions list and abort any uncommitted transaction other than our own. These steps can be combined

[5] Note that a standard CAS would not suffice, as the load-linked targets a different location than the store-conditional.

[6] Note again that a CAS does not suffice.

```
inline copyBackField(o, f, mode, st) {
  _nonceV=NIL; _ver = NIL; _r = NIL; st = success;
  /* try to abort each version.  when abort fails, we've got a committed version. */
  do
  :: _ver = object[o].version;
     if
     :: (_ver==NIL) ->
        st = saw_race; break /* someone's done the copyback for us */
     :: else
     fi;
      /* move owner to local var to avoid races (owner set to NIL behind our back) */
     _tmp_tid=version[_ver].owner;
     tryToAbort(_tmp_tid);
     if
     :: (_tmp_tid==NIL || transid[_tmp_tid].status==committed) ->
        break /* found a committed version */
     :: else
     fi;
     /* link out an aborted version */
     assert(transid[_tmp_tid].status==aborted);
     CAS_Version(object[o].version, _ver, version[_ver].next, _);
  od;
  /* okay, link in our nonce.  this will prevent others from doing the copyback. */
  if
  :: (st==success) ->
     assert (_ver!=NIL);
     allocVersion(_retval, _nonceV, aborted_tid, _ver);
     CAS_Version(object[o].version, _ver, _nonceV, _cas_stat);
     if
     :: (!_cas_stat) ->
        st = saw_race_cleanup
     :: else
     fi
  :: else
  fi;
  /* check that no one's beaten us to the copy back */
  if
  :: (st==success) ->
     if
     :: (object[o].field[f]==FLAG) ->
        _val = version[_ver].field[f];
        if
        :: (_val==FLAG) -> /* false flag... */
           st = false_flag /* ...no copy back needed */
        :: else -> /* not a false flag */
           d_step { /* LL/SC */
             if
             :: (object[o].version == _nonceV) ->
                object[o].fieldLock[f] = _thread_id;
                object[o].field[f] = _val;
             :: else /* hmm, fail.  Must retry. */
                st = saw_race_cleanup /* need to clean up nonce */
             fi
           }
        fi
     :: else /* may arrive here because of readT, which doesn't set _val=FLAG*/
        st = saw_race_cleanup /* need to clean up nonce */
     fi
  :: else /* !success */
  fi;
  /* always kill readers, whether successful or not.  This ensures that we make progress if
   * called from writeNT after a readNT sets readerList non-null without changing FLAG to
   * _val (see immediately above; st will equal saw_race_cleanup in this scenario). */
  if
  :: (mode == kill_all) ->
     killAllReaders(o, _r); /* see Appendix for details */
  :: else /* no more killing needed. */
  fi;
  /* done */
}
```

Too small
Make 2 pages?

Figure 3.9: The field copy-back routine.

and hoisted so that they are done once before the first read from an object and not repeated.

At read time, we initially read from the original object. If the value read is not FLAG, we use it. Otherwise, we look up the version object associated with our transaction (this will typically be at the head of the version list) and read the appropriate value from that version. Note that the initial read-and-check can be omitted if we know that we have already written to this field inside this transaction.

The top of Figure 3.10 specifies the transactional read operation in Promela.

### Transactional Write

Again, writes are split in two. Once for each object, we must traverse the version list, aborting other versions and locating or creating a version corresponding to our transaction. We must also traverse the reader list, aborting all transactions on the list except ourself. This is shown in the `ensureWriter` routine in Figure 3.11.

Once for each field we intend to write, we must perform a copy-through: copy the object's field value into all the versions and then write FLAG to the object's field. We use load-linked/store-conditional to update versions only if the object's field has not already been set to FLAG behind our backs by another copy-through. The `checkWriteField` routine is shown in Figure 3.12.

Then, for each write, we simply write to the identified version, as shown at the bottom of Figure 3.10.

## 3.4 Performance limits for non-transactional code

The software transaction system outlined in this section depends on the insertion of simple read and write checks in non-transactional code. The performance of read and write barriers have been well-studied in the garbage

```
inline readT(tid, o, f, ver, result) {
  do
  ::
     /* we should always either be on the readerList or
      * aborted here */
     result = object[o].field[f];
     if
     :: (result==FLAG) ->
        if
        :: (ver!=NIL) ->
           result = version[ver].field[f];
           break /* done! */
        :: else ->
           findVersion(tid, o, ver);
           if
           :: (ver==NIL) ->/*use val from committed vers.*/
              assert (_r!=NIL);
              result = version[_r].field[f];/*false flag?*/
              moveVersion(_r, NIL);
              break /* done */
           :: else /* try, try, again */
           fi
        fi
     :: else -> break /* done! */
     fi
  od
}

inline writeT(ver, f, nval) {
  /* easy enough: */
  version[ver].field[f] = nval;
}
```

Figure 3.10: Promela specification of transactional read and write operations.

```
/* per-object, before write. */
inline ensureWriter(tid, o, ver) {
  assert(tid!=NIL);
  ver = NIL; _r = NIL; _rr = NIL;
  do
  :: assert (ver==NIL);
     findVersion(tid, o, ver);
     if
     :: (ver!=NIL) -> break /* found a writable version for us */
     :: (ver==NIL && _r==NIL) ->
        /* create and link a fully-committed root version, then
         * use this as our base. */
        allocVersion(_retval, _r, NIL, NIL);
        CAS_Version(object[o].version, NIL, _r, _cas_stat)
     :: else ->
        _cas_stat = true
     fi;
     if
     :: (_cas_stat) ->
        /* so far, so good. */
        assert (_r!=NIL);
        assert (version[_r].owner==NIL ||
                transid[version[_r].owner].status==committed);
        /* okay, make new version for this transaction. */
        assert (ver==NIL);
        allocVersion(_retval, ver, tid, _r);
        /* want copy of committed version _r.  No race because
         * we never write to a committed versions. */
        version[ver].field[0] = version[_r].field[0];
        version[ver].field[1] = version[_r].field[1];
        assert(NUM_FIELDS==2); /* else ought to initialize more fields */
        CAS_Version(object[o].version, _r, ver, _cas_stat);
        moveVersion(_r, NIL); /* free _r */
        if
        :: (_cas_stat) ->
           /* kill all readers (except ourself) */
           /* note that all changes have to be made from the front of the
            * list, so we unlink ourself and then re-add us. */
           do
           :: moveReaderList(_r, object[o].readerList);
              if
              :: (_r==NIL) -> break
              :: (_r!=NIL && readerlist[_r].transid!=tid)->
                 tryToAbort(readerlist[_r].transid)
              :: else
              fi;
              /* link out this reader */
              CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _)
           od;
           /* okay, all pre-existing readers dead & gone. */
           assert(_r==NIL);
           /* link us back in. */
           ensureReaderList(tid, o);
           break
        :: else
        fi;
        /* try again */
     :: else
     fi;
     /* try again from the top */
     moveVersion(ver, NIL)
  od;
  /* done! */
  assert (_r==NIL);
}
```

Figure 3.11: The per-object version-setup routine for transactional writes.

```
/* per-field, before write. */
inline checkWriteField(o, f) {
  _r = NIL; _rr = NIL;
  do
  ::
     /* set write flag, if not already set */
     _val = object[o].field[f];
     if
     :: (_val==FLAG) ->
        break; /* done! */
     :: else
     fi;
     /* okay, need to set write flag. */
     moveVersion(_rr, object[o].version);
     moveVersion(_r, _rr);
     assert (_r!=NIL);
     do
     :: (_r==NIL) -> break /* done */
     :: else ->
        object[o].fieldLock[f] = _thread_id;
        if
        /* this next check ensures that concurrent copythroughs don't stomp on each other's versions,
         * because the field will become FLAG before any other version will be written. */
        :: (object[o].field[f]==_val) ->
           if
           :: (object[o].version==_rr) ->
              atomic {
                if
                :: (object[o].fieldLock[f]==_thread_id) ->
                   version[_r].field[f] = _val;
                :: else -> break /* abort */
                fi
              }
           :: else -> break /* abort */
           fi
        :: else -> break /* abort */
        fi;
        moveVersion(_r, version[_r].next) /* on to next */
     od;
     if
     :: (_r==NIL) ->
        /* field has been successfully copied to all versions */
        atomic {
          if
          :: (object[o].version==_rr) ->
             assert(object[o].field[f]==_val ||
                    /* we can race with another copythrough and that's okay; the locking strategy
                     * above ensures that we're all writing the same values to all the versions
                     * and not overwriting anything. */
                    object[o].field[f]==FLAG);
             object[o].fieldLock[f]=_thread_id;
             object[o].field[f] = FLAG;
             break; /* success!  done! */
          :: else
          fi
        }
     :: else
     fi
     /* retry */
  od;
  /* clean up */
  moveVersion(_r, NIL);
  moveVersion(_rr, NIL);
}
```

Figure 3.12: The per-field copy-through routine for transactional writes.

collection community, but our checks are slightly different: in particular, they are checks on the *contents* of a memory cell, rather than on its address. This introduces a more direct dependency which may affect performance. Further, our write check involves a LL/SC instruction pair, which may behave differently from the standard loads and stores used in barriers.

In this section we use a simple counter microbenchmark to evaluate the "best worst case" non-transactional performance of our transaction system. It is an idealized "best case" in that we won't benchmark the effects of "false flags" or other forays into the transactional code path, nor will we account for double-word or sub-word writes, cache effects due to code duplication, or other details of a real implementation. These effects we will investigate with full benchmarks in the chapter 4.5. We will also feel free to optimize down to the assembly level to determine our fundamental performance limits. However, the tight read/write dependency of a counter increment makes it a "worst case" benchmark: in general, modern compilers are very good at separating reads from writes in "real" code to mask load latency, but our particular microbenchmark can not be reasonably unrolled to accomplish this separation. Our conclusions about the fundamental costs of read and write costs should thus be tempered by the knowledge that some of these costs are in practice be masked by the same standard compiler techniques used to mitigate memory-access latencies.

Figure 3.13 presents the basic structure of the counter microbenchmark. The `read()` and `write()` methods will have appropriate definitions inlined for each variant of the benchmark. Note that the `readerList` and `field` fields of the `struct oobj` are marked `volatile` to prevent the C compiler from optimizing away the accesses we are interested in benchmarking; without these declarations, gcc 4.1.2 will optimize away the entire benchmark loop and replace it with a direct addition of `REPETITIONS`.

Figure 3.14 shows the baseline `read()` implementation, which inlines to a single `lwz` instruction on PowerPC, along with the C implementation of the read check necessary for non-transactional code under our transaction

```
typedef int32_t field_t;

struct oobj {
  struct version *version;
  struct readerList * volatile readerList;
  volatile field_t field[NUM_FIELDS];
};
void do_bench(struct oobj *obj) {
  int i;
  for (i=0; i<REPETITIONS; i++) {
    field_t v = read(obj, 0);
    v++;
    write(obj, 0, v);
  }
}
```

Figure 3.13: Counter microbenchmark to evaluate read- and write-check overhead for non-transactional code.

```
#if !defined(WITH_READ_CHECKS)

static inline field_t read(struct oobj *obj, int idx) {
  field_t val = obj->field[idx];
  return val;
}

#else

static inline field_t read(struct oobj *obj, int idx) {
  field_t val = obj->field[idx];
  if (unlikely(val==FLAG))
    return unusualRead(obj,idx);
  else return val;
}

#endif
```

Figure 3.14: C implementation of read checks for counter microbenchmark.

```
#if !defined(WITH_WRITE_CHECKS)

static inline void write(struct oobj *obj, int idx, field_t val) {
  obj->field[idx] = val;
}

#else

static inline void write(struct oobj *obj, int idx, field_t val) {
  if (unlikely(val==FLAG))
    unusualWrite(obj,idx,val); // never called
  else {
    do {
      if (unlikely(NULL != LL(&(obj->readerList)))) {
        unusualWrite(obj,idx,val); // never called
        break;
      }
    } while (unlikely(!SC(&(obj->field[idx]), val)));
  }
}

#endif
```

Figure 3.15: C implementation of write checks for counter microbenchmark.

system. We use the `unlikely()` macro, implemented using the `gcc` extension `__builtin_expect()`, to apply the appropriate static prediction bits indicating that `FLAG` is expected to be an unusual value. In our microbenchmark, this prediction will always correct. The C compiler must still save whatever registers are necessary to allow the call to `unusualRead()`, although this function is never actually called during the microbenchmark. The `unusualRead` function would be expected to perform the remainder of the `readNT` algorithm from Figure 3.8, including looking up and copying back the most recently-committed value for this field.

Figure 3.15 shows the equivalent implementation pairs for a non-transactional write. The basic implementation inlines to a single `stw` instruction. The write-check version at the bottom must perform three tests. First, it must ensure that the value to be written is not `FLAG`; if it is we must use the "false flag" mechanism to write it, modeled here by a call to `unusualWrite()`. This

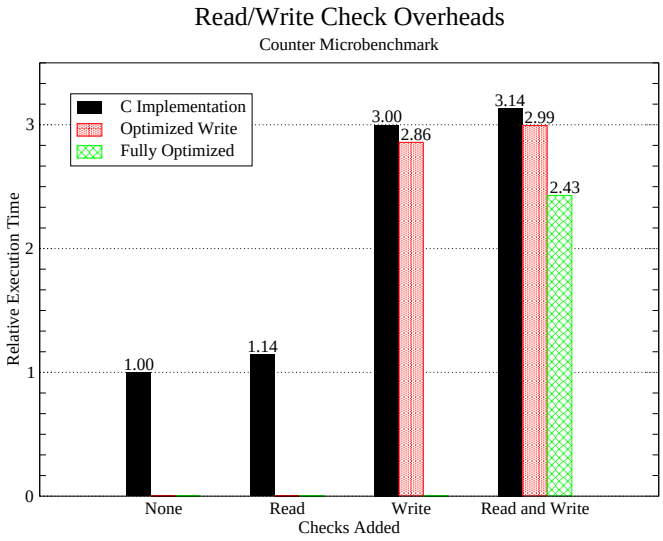



Figure 3.16: Time overhead of read checks, write checks, and both read and write checks for non-transactional code, in both a pure C implementation and optimized assembly.

check can be optimized away when writing a constant,[7] but the `volatile` declarations ensure that our benchmark will always perform the check. Second, we must perform a load-linked of the object's `readerList` to check that there are not any current transactional readers of this object. Last we perform a store-conditional of the value we wish to write and check that it was successful. Unlike the other two tests, this check will fail a number of times[8] during the benchmark run due to context switches that may occur between the load-linked and the store-conditional instructions.

The performance of this benchmark is shown in Figure 3.16. Read checks, even in this worst case where the value read is demanded imme-

[7] Hennessy and Patterson indicate that 35% of integer instructions use an immediate operand; their methodology does not allow breaking out the percentage of stores specifically [32, p. 78].

[8] Experimentally, about 3,600 times in the 1 billion repetitions of the write during the counter microbenchmark.

diately, only add 14% overhead. Write checks are more costly, due to the load-linked and store-conditional pair they add 200% overhead to the benchmark. Combining read and write checks yields the expected 214% overhead; our microbenchmark was carefully constructed to avoid the opportunities for instruction-level parallelism one might expect in real-world applications.

Examination of the assembly code generated for this simple benchmark seems to indicate substantial scope for improvement. The inline assembly mechanism of gcc/C provides no inherent support for instructions like the PowerPC's "store-conditional" (`stwcx.`) that leave their results in a condition code register. Figure 3.17 shows the assembly emitted for the write-check version of the microbenchmark, with the cumbersome mechanism required to move the condition code to a register so that it can then be retested. We can easily hand-code a better write-check mechanism, shown in the right-hand column of the figure.[9] The optimized store-conditional test reduces write-check overhead to 186% and combined read- and write-check overhead to 199%, as shown in Figure 3.16.

Further performance improvement is possible by optimizing the benchmark with both read and write checks as a whole. Figure 3.18 shows an optimized assembly version of our `do_bench()` function. I've primarily rescheduled the code here to separate dependent instructions as much as possible, but I've also ensured repeatable instruction cache alignment,[10] replaced our canonical `FLAG` value with `0xFFFFCACA`, which can be represented by the PowerPC's 16-bit signed immediate instruction field (eliminating the need for a register), and combined the flag checks in the read and write

[9] This version has a stub where a function call to `unusualWrite` would usually go; the correct code here would branch to a small thunk that will perform the frame operations and register saves necessary to adhere to the C calling convention; gcc makes it difficult to ensure that this thunk is "near enough" for a direct branch (within a 24-bit signed displacement) without duplicating it needlessly.

[10] The MPC7447 (G4) PowerPC has a 32-byte cache line, although fetches occur in 16-byte (4 instruction) chunks.

```
do_bench:
  mflr 0
  stwu 1,-16(1)
  lis 5,0x3b9a
  li 8,0
  ori 5,5,51712
  addi 6,3,4
  addi 7,3,8
  stw 0,20(1)
  b .L4
.L22:
  mr 10,6
  mr 11,7
.L5:
  lwarx 0,0,10
  cmpwi 7,0,0
  bne- 7,.L19
  stwcx. 9,0,11
  li 0,0
  bne- 0f
  li 0,1
0:
  cmpwi 7,0,0
  beq- 7,.L5
  addi 8,8,1
  cmpw 7,8,5
  beq- 7,.L21
.L4:
  lwz 9,8(3)
  cmpwi 7,9,-13623
  addi 9,9,1
  bne+ 7,.L22
.L21:
  lwz 0,8(3)

  cmpw 7,0,8
  bne- 7,.L23
  lwz 0,20(1)
  addi 1,1,16
  mtlr 0
  blr
```

```
do_bench:
  mflr 0
  stwu 1,-16(1)
  addi 10,3,8
  addi 11,3,4
  stw 0,20(1)
  lis 0,0x3b9a
  ori 0,0,51712
  mtctr 0
  b .L4

.L5:
0:
  lwarx 0,0,11
  cmpwi 0,0
  beq+ 1f
  b . # stub for unusualWrite branch
1:stwcx. 9,0,10
  bne 0b

  bdz .L14

.L4:
  lwz 9,8(3)
  cmpwi 7,9,-13623
  addi 9,9,1
  bne+ 7,.L5
.L14:
  lwz 0,8(3)
  xoris 9,0,0xc465
  cmpwi 7,9,-13824
  bne 7,.L15
  lwz 0,20(1)
  addi 1,1,16
  mtlr 0
  blr
```

Figure 3.17: PowerPC assembly for counter microbenchmark with write checks. The right-hand column is generated from the C implementation; the left-hand column has an optimized version of the write check inlined into the code using the `asm` keyword. Italicized code is essentially unchanged; the optimized section is shown in boldface.

```
    # repetition count is in the count register to start
    b 0f
    .balign 32 # align to 32-byte cache-line boundary
    nop
    nop
    nop
    nop
    nop
    nop
0:  lwarx 5,0,0
    lwz 6, 0(9)
1:  ori 8, 6, 3
    addi 7, 6, 1
    cmpwi 1, 5,0
    cmpwi 2, 8, 0xFFFFCACB
    bne- 1, 0b # stub: unusualWrite
    beq- 2, 0b # stub: unusualRead or unusualWrite
    stwcx. 7,0,9
    lwarx 5,0,0
    lwz 6, 0(9)
    bne- 0, 1b
    bdnz 1b
```

Figure 3.18: Optimized PowerPC assembly for counter microbenchmark with both read and write checks.

routines.[11] The read-and-write check overhead is improved to 143% with these optimizations, due mostly to the irreducible dependency between the load-linked and store-conditional instructions.

Hennessy and Patterson's measurements indicate that load and store instructions comprise respectively 26% and 9% of dynamic instruction counts in SPECint92 on a RISC microarchitecture (DLX) [32, p. 105]. As a rough estimate, using dynamic instruction count as a proxy for instruction time, the 14% read overhead and 186% write overhead measured in this microbenchmark thus translate into 20% net overhead for non-transactional code.[12] If we conservatively assume that most of the improvement in Figure 3.18's optimized checks should be credited to reduced write-check over-

[11] Again, the branches to `unusualRead()` and `unusualWrite()` are difficult to express in this hybrid of C and assembly, but small stubs would be written to save registers and perform a branch with the appropriate calling conventions.

[12] $0.65 + (1.14 \times 0.26) + (2.86 \times 0.09) = 1.20$

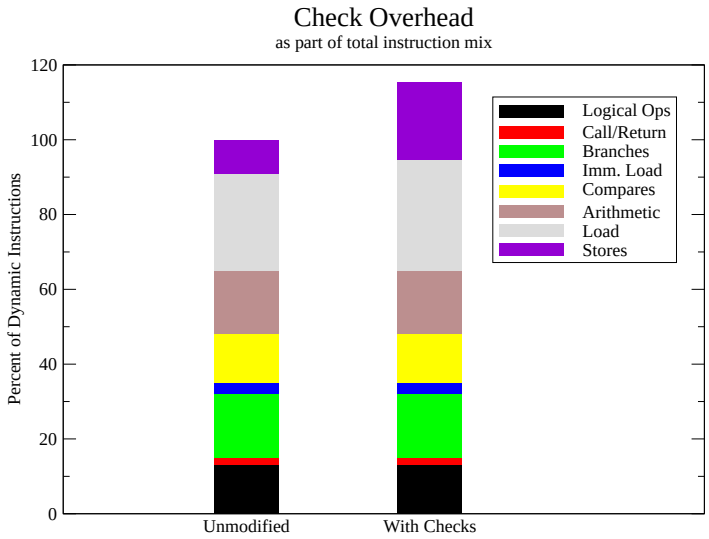



Figure 3.19: Check overhead as percentage of total dynamic instruction count. The bar on the left is the DLX instruction mix for SPECint92 from [32, p.105]. On the right is the same instruction mix after scaling the loads and stores according to Figure 3.16.

head, giving the same 14% read overhead but only 129% write overhead, the same metric gives only 15% net overhead, as shown in Figure 3.19. This should be considered a rough lower bound for the time overhead, possible with aggressive optimization and scheduling.[13]

## Implementation

There are a number of details that are not yet covered by our model, for example; how are fields of different byte lengths handled? How does transactional Java code interact with the garbage collector, or with native code in the runtime system? The next chapter addresses the real-world implementation details of our software transaction system.

[13] Note that this calculation combines over- and underestimation, making this number only a rough bound. The use of dynamic instruction count, rather than dynamic instruction time, likely underestimates the contribution of loads and stores to the run time. However, scheduling, prefetching, and store buffers can pipeline the cost of the loads and stores, which might cause our overhead estimate to exceed the actual cost. Chapter 4.5 will present actual overhead measurements of real Java applications.

In theory, there is no difference between theory and practice.

*Jan L. A. van de Snepscheut*

# Chapter 4

# Implementing efficient software transactions

In the previous chapter I described the design of an efficient software transaction system. In this chapter I will discuss the challenges involved in translating this design to a practical Java compiler, the implementation techniques I used, and the performance achieved. I will conclude by listing some limitations of the implementation, and describing how they may be overcome in a production-quality compiler.

## 4.1 The FLEX compiler infrastructure

In 1998 I began implementing the FLEX compiler infrastructure for Java [3], which I will use to evaluate the transaction system implementations in this thesis. FLEX has been used in over 20 published papers, on a wide range of topics.

FLEX is a whole-program static compiler for Java, with a runtime system built around the GNU Classpath implementation of the Java standard libraries [23]. FLEX takes as input Java bytecode, generated by any Java 1.0-1.6 compiler from user code and the GNU Classpath 0.08 library implementation, and emits either assembly code for MIPS, Sparc, or StrongARM,

or else "portable assembly language" written in gcc 3.4's variant of C. The emitted code is compiled and linked against FLEX's runtime system and the GNU Classpath 0.08 native libraries to produce a stand-alone binary for the target system. The FLEX compiler and analysis code is written in Java, while the FLEX runtime system is written in C.

### Using C as a target

The experiments in this thesis were conducted on either the x86 or PowerPC architectures, for which FLEX does not have a native assembly backend. The "Precise C" backend was thus used, so named because, aside from emitting C code, it's original purpose was to investigate the possibility of precise garbage collection whist emitting high-level code. To that end, the backend contains code to maintain a separate stack for values of non-primitive types, and to push and pop all live variables to this stack at gc points. Since FLEX's low-level IR may create derived pointers that point inside objects, for example during loop optimizations, the Precise C backend will also reconstruct the derived pointers after the gc point, in case the objects involved have moved.

Experiments showed that this mechanism had minimal impact on performance, because variables pushed onto the explicitly managed "object stack" were then dead across the call from the perspective of the underlying C compiler; in effect our explicit stack management replaced the implicit stack save/restore which the C compiler would otherwise have performed to maintain its calling convention.

Java exceptions presented another difficulty when implementing a C backend. The mechanisms used in our assembly backends (separate return addresses for "returning" an exception, either derived by rule from the normal return address or stored in a sorted table keyed by the normal return address) can not be implemented in C. FLEX supports two alternate translations: the first uses `setjmp` and `longjmp` to branch directly from the site where the exception is thrown to an appropriate handler, and the

second returns a pair value from every function call. The `setjmp` method only incurs cost when a new exception-handling region (`try/catch` block) is entered, or when an exception is thrown, but `setjmp` and `longjmp` are rather expensive. The pair-return method returns a C `struct` consisting of the "real" return value along with an exception value.[1] When the function returns, the caller must first test the exception value; if it is non-`NULL`, then an exception has been thrown and the caller must handle if (if it can) or re-throw it.

The mandatory test in the pair-return method adds overhead to every function call, but it is minimal. Experiments on x86 indicated that a pair-return call cost 2ms for both the "normal return" and "exception thrown" case, while the `setjmp` method cost only 1ms for the "normal return" but 73ms for "exception thrown". Thus `setjmp` is dramatically slower for applications that use exceptions, such as the SPECjvm98 benchmark `202_jess`.

The experiments in this thesis all use pair-return to implement exceptions, and conservative garbage collection, so the "Precise C" stack is not required.

## 4.2 Transforming synchronization

Aside from implementing the read and write mechanisms of the software transaction design presented in the previous chapter, a number of other transformations and analyses of our Java benchmarks must be performed: transactions must be synthesized from the benchmark's monitor synchronization, methods must be cloned according to the context (whether inside a transaction or not) of their call site, analyses must be performed in order to reduce redundant checks in the transformed code, and some minor desugaring is done to ease implementation.

```
synchronized {
    ...
    x = o.f;
    ...
    o.f = y;
    ...
    z = foo(a, b, c);
    ...
}
```

⇒

```
t = CommitRecord.newTransaction();
while(true) {
  try {
    ...
    v = ensureReader(o, t);
    x = readT(o, FIELD_F, &v, t);
    ...
    v = ensureWriter(o, t);
    checkWriteField(o, FIELD_F);
    writeT(o, FIELD_F, y, v); /* v.f = y */
    ...
    z = foo$$withtrans(t, a, b, c);
    ...
    t.commitTransaction();
    break;
  } catch (TransactionAbortException tex) {
    t = t.retryTransaction();
  }
}
```

Figure 4.1: Software transaction transformation. The code on the left is the original Java source; the transformed source is on the right.

### 4.2.1 Method transformation

We automatically create transactions in our benchmarks from Java synchronized blocks and methods. Figure 4.1 illustrates the transformation applied.

Entering a top-level synchronized block creates a new transaction object and starts a new exception handler context. Exiting the block causes a call to the transaction's commit method. If the commit, or any other operation inside the block, throws `TransactionAbortException` indicating that the transaction must be aborted, we recreate the transaction object and loop to retry the transaction's operations. The `retryTransaction` method performs backoff; it may also perform livelock detection or other transaction management functions.

Inside the transaction context, reads and writes must be transformed. Before the read, the `ensureReader` algorithm must be invoked (see chapter 3.3.2 and Appendix A). This need only be done once for each object read in this transaction; chapter 4.2.2 described how these checks are hoisted

[1] Methods declared as `void` return only the exception value, of course.

and combined to reduce their number. The ensureReader routine returns a pointer to a ***version object*** containing this transaction's field values for the object. The pointer may be NULL, however, if this transaction has not written the object. The actual read is done via the readT algorithm described in Figure 3.10 and chapter 3.3.2; it may update the cached version object for the given object (for example, if the object has been written within this transaction since the point at which ensureReader was invoked).[2]

Before each write, ensureWriter and checkWriteField must be executed. Like ensureReader, ensureWriter (Figure 3.11) need only be performed once for each object written in the transaction, and is hoisted and combined in the same way. The checkWriteField algorithm (Figure 3.12) need only be executed once for each object field written; if the same field in the same object is written multiple times, the subsequent checkWriteField invocations can be eliminated. The actual write is performed via writeT (Figure 3.10), which is a simple store to the version object.

Nested transactions are implemented via subsumption; that is, nested synchronized blocks inside a transaction context are ignored: the inner transaction is subsumed by the outermost.

Method invocations inside transaction context must be transformed, since read and write operations are implemented differently depending on whether or not the current execution is inside a transaction. We create a transactional clone from each method, named with a $$withtrans suffix, which is invoked when we are executing inside a transaction. We pass the current transaction as the first parameter of the cloned method.[3]

Non-transactional code inside a method must be transformed to use the readNT and writeNT mechanisms to perform object reads and writes

[2]The FLEX infrastructure cannot create pointers to temporaries, or alternately return multiple values from function calls, so the version object in my current implementation cannot be updated by readT. This impairs our ability to hoist and combine ensureReader and requires us to make redundant calls.

[3]Note that cloning must take virtual dispatch into account: cloning a method in a superclass must also create appropriate cloned subclass implementations.

(Figure 3.8); the implementation is similar to that shown in Figure 3.14 and Figure 3.15.

### 4.2.2 Analyses

Our performance will be improved if we can identify objects or fields that are guaranteed not to be concurrently mutated, and replace our checks and read/write protocols with direct accesses. We perform two analyses of this sort; chapter 4.6 contains additional analyses which we have not implemented.

Our first analysis classifies all fields in the program according to their use in transactional and non-transactional regions. This analysis is encapsulated in a FLEX class named `GlobalFieldOracle`. Fields that can never be written within a transaction do not need the special `readNT` mechanism from non-transactional code; their values can be loaded directly without testing for `FLAG`. The only way that the field can have the `FLAG` value, if it is not written within a transaction, is if it is a false flag, in which case its value really is `FLAG`. Similarly, fields that can never be read or written within a transaction do not need to use the `writeNT` mechanism; they can just store directly to the field.[4] Thus `GlobalFieldOracle` can make non-transactional code more efficient by eliminating the overhead of the software transaction system in some cases.

While `GlobalFieldOracle` targets non-transactional code, the `CheckOracle` analysis classes make transactional code more efficient by removing unneeded read and write checks. As mentioned above, the `ensureReader` and `ensureWriter` checks need only be invoked once on each unique object read/written by the transaction. Similarly, `checkWriteField` need only be performed once on each object field written by the transaction. The `CheckOracle` classes hoist each check to its immediate dominator iff this new location for the check is postdominated by the current location and

[4] Note that its not enough for the field not to be written; we must also protect against write-after-write conflicts.

the new location is dominated by the definition of the variable referenced in the check. This ensures that all paths to the original location of the check must pass through the new location, and that the object involved in the check is still defined appropriately at the new check location, since the analysis is performed on a single-assignment intermediate representation [4]. The check-hoisting process is repeated until no checks can be moved higher, then we eliminate any check that is dominated by an identical check.

### 4.2.3 Desugaring

We also desugar some Java idioms to ease implementation. Java specifies a `clone()` method of `java.lang.Object`, from which every object is derived. The top-level `clone()` method is somewhat magical, as it creates an exact copy of its subclass, including all its fields, which are otherwise hidden from methods of a superclass. The transactional translation of this method is also somewhat baroque: it would have to mark all fields of the original method as "read" by the transaction, and then construct a versioned object written by the transaction. As an end-run around this complexity, we desugar `clone()` into individual implementations in appropriate classes, each of which reads all fields of the class, and constructs a new object bypassing its constructors, and then individually sets all the fields to their new values. Arrays get a similar `clone()` implementation that reads and writes the elements of the array. These new implementations can then be transformed by the transaction pass like any other piece of code that reads and writes fields.

The FLEX infrastructure also contains a mechanism for fast initialization of arrays; this mechanism is desugared into individual array set operations so that it, too, can be uniformly transformed by the transaction pass.

## 4.3 Runtime system implementation

The FLEX runtime system was extended to support transactions. The runtime uses the standard Java Native Interface (JNI) [66] for native methods

(written in C) called from Java. Certain parts of the transaction system were written as native methods invoked via JNI, but others interact with the runtime system at a much lower level.

The runtime system is parameterized to allow a large number of memory allocation strategies and collectors, but for the experiments reported here I used the Boehm-Demers-Weiser conservative garbage collector [14].[5] Ideally the garbage collector would know enough about the transaction implementation to be able to automatically prune unneeded committed or aborted versions from the object's version list during collection. However, we relied on opportunistically nulling out tail pointers during version list traversal (for example, while looking for the last committed version), which may result in incomplete collections (and more memory usage than strictly necessary).

Each transaction had a `CommitRecord` object storing its state, whether `COMMITTED`, `ABORTED`, or still `WAITING`. Most `CommitRecord` methods were written in Java, including object creation, exponential backoff on retry, and throwing the appropriate `TransactionAbortException` on abort. Only the crucial `commit()` and `abort()` operations, which atomically set the transaction status iff it is still `WAITING`, were written in C, as JNI methods.

### 4.3.1 Implementing the JNI

No Java transaction implementation is complete without some mechanism for executing native methods, that is, C code. One cannot banish all native methods, since the Java standard library is built on them. If one cannot make native code observe the `readNT` and `writeNT` protocols, one must ensure that the native code never reads any object written in a transaction, or writes any object read or written in a transaction.[6] This is very inconvenient.

[5] The use of a conservative collector means that the mechanism described in chapter 4.1 to enable precise garbage collection was unneeded and thus disabled for these experiments.

[6] If one can implement `writeNT` but not `readNT`, one need only ensure that native code does not write fields that are also written inside a transaction, as discussed in chapter 4.2.2.

```
struct JNINativeInterface {
  ...
  /* Calling instance methods */
  jmethodID (*GetMethodID)
    (JNIEnv *env, jclass clazz, const char *name, const char *sig);
  jobject (*CallObjectMethod)
    (JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ...
  jboolean (*CallBooleanMethod)
    (JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ...
  jbyte (*CallByteMethod)
    (JNIEnv *env, jobject obj, jmethodID methodID, ...);
  ...
  /* Accessing fields of objects */
  jfieldID (*GetFieldID)
    (JNIEnv *env, jclass clazz, const char *name, const char *sig);
  jobject (*GetObjectField)
    (JNIEnv *env, jobject obj, jfieldID fieldID);
  jboolean (*GetBooleanField)
    (JNIEnv *env, jobject obj, jfieldID fieldID);
  ...
  void (*SetObjectField)
    (JNIEnv *env, jobject obj, jfieldID fieldID, jobject value);
  void (*SetBooleanField)
    (JNIEnv *env, jobject obj, jfieldID fieldID, jboolean value);
  ...
  /* Array Operations */
  jobject (*GetObjectArrayElement)
    (JNIEnv *env, jobjectArray array, jsize index);
  void (*SetObjectArrayElement)
    (JNIEnv *env, jobjectArray array, jsize index, jobject value);
  ...
  jboolean* (*GetBooleanArrayElements)
    (JNIEnv *env, jbooleanArray array, jboolean *isCopy);
  jbyte* (*GetByteArrayElements)
    (JNIEnv *env, jbyteArray array, jboolean *isCopy);
  ...
};
```

Figure 4.2: A portion of the Java Native Interface for interacting with the Java runtime from C native methods [66]. There are function variants for all of the basic Java types: boolean, byte, char, short, int, long, float, double, and Object. Note that the `jobject` and `j*Array` types are not direct references to the heap objects; they are opaque wrappers that preserve the garbage collector's invariants and protect the runtime system's private implementation.

Our solution ensures not only that (separately compiled) native code properly uses the `readNT` and `writeNT` protocols outside a transaction, but also that reads and writes inside a transaction are handled properly. This allows the use of "safe" native methods (those without external side effects) inside a transaction.

We are able to do this because FLEX uses the Java Native Inteface [66] to interact with native code. The JNI, a portion of which is shown in Figure 4.2, abstracts field accesses and Java method invocations from native code, so we can substitute implementations that behave appropriately whether in a transaction or not.

We distinguish between three classes of native methods. The first are native methods that are "safe" in a transactional context. That is, they have no external side effects (which would need to be undone if the transaction aborted), and will behave correctly in a transaction if all reads, writes, and method calls are simply replaced by the appropriate transactional protocol. The second are methods that have a different implementation in a transactional context. One example would be the native methods that implement file I/O; in a transactional context these should interface with a underlying transactional file system and arrange for the I/O operations to commit iff the current transaction commits. Another example is the `Object.wait()` method; the appropriate implementation is described in chapter 4.4.4. The last class of native methods are those that are inherently impossible in a transactional context, usually due to irreversible external I/O (chapter 7.4 discusses I/O interactions in more detail). A compiler configuration file identifies the safe native methods.

For "safe" native methods, the FLEX compiler creates a thunk that stores the transaction object in the JNI environment structure (`JNIEnv`) which is passed to every JNI method. When that native method invokes the field or array operations in the JNI (for example, `Get*Field`, `Set*Field`, `Get*ArrayElements`, etc.) the runtime checks the environment structure to determine whether to use the appropriate transactional or non-transactional

```
/* Framework for use of preprocessor specialization. */
/* (from readwrite.c) */

#define VALUETYPE jboolean
#define VALUENAME Boolean
#include "transact/readwrite-impl.c"
#define ARRAY
#include "transact/readwrite-impl.c"
#undef ARRAY
#undef VALUENAME
#undef VALUETYPE

/* ... etc ... */

#define VALUETYPE jdouble
#define VALUENAME Double
#include "transact/readwrite-impl.c"
#define ARRAY
#include "transact/readwrite-impl.c"
#undef ARRAY
#undef VALUENAME
#undef VALUETYPE

#define VALUETYPE struct oobj *
#define VALUENAME Object
#include "transact/readwrite-impl.c"
#define ARRAY
#include "transact/readwrite-impl.c"
#undef ARRAY
#undef VALUENAME
#undef VALUETYPE
```

Figure 4.3: Specializing transaction primitives by field size and object type: `readwrite.c`. This figure shows how `readwrite.c` specializes the code in `readwrite-impl.c` (Figure 4.4) through the use of multiple `#include` statements.

read or write protocol. When Java methods are invoked, via the `Call*Method` family of methods, in a transactional context the runtime looks up the `$$withtrans` suffixed version of the method (see chapter 4.2.1) and invokes it, passing the transaction object from the environment as the first parameter.

For unsafe native methods, FLEX generates a call to a `$$withtrans`-suffixed JNI method, passing the transaction as the first parameter as is done for pure Java calls inside a transaction context. The implementer is then responsible for correct transactional behavior.

```
/* Implementing generic functions. */
/* (from readwrite-impl.c) */
#include "transact/preproc.h" /* Defines 'T()' and 'TA'() macros. */

VALUETYPE TA(EXACT_readNT)(struct oobj *obj, unsigned offset);

VALUETYPE TA(EXACT_readNT)(struct oobj *obj, unsigned offset) {
  do {
    VALUETYPE f = *(VALUETYPE*)(FIELDBASE(obj) + offset);
    if (likely(f!=T(TRANS_FLAG))) return f;
    if (unlikely(SAW_FALSE_FLAG ==
                 TA(copyBackField)(obj, offset, KILL_WRITERS)))
      return T(TRANS_FLAG); // "false" transaction: field really is FLAG.
    // okay, we've done a copy-back now.  retry.
  } while(1);
}

/* ... undefine macros at end of readwrite-impl.c ... */
```

Figure 4.4: Specializing transaction primitives by field size and object type: `readwrite-impl.c`. In this snippet the `readNT` algorithm is implemented using macros (defined in Figure 4.5, `preproc.h`) to implement generic field types and naming conventions. Note the use of the macros `likely` and `unlikely`, which communicate static branch prediction information to the C compiler.

```
/* Preprocessor specialization macros */
/* (from preproc.h) */
#define x1(x,v) x2(x,v)
#define x2(x,v) x ## _ ## v

#if defined(ARRAY)
# define A(x) x1(x,Array)
# define OBJ_OR_ARRAY(x,y) (y)
#else
# define A(x) x
# define OBJ_OR_ARRAY(x,y) (x)
#endif

#define T(x) x1(x,VALUENAME)
#define TA(x) T(A(x))

#define FIELDBASE(obj) \
         OBJ_OR_ARRAY(obj->field_start,((struct aarray *)obj)->element_start)
```

Figure 4.5: Specializing transaction primitives by field size and object type: `preproc.h`. In this portion of the file we define the specialization macros used in Figure 4.4.

### 4.3.2 Preprocessor specialization

Part of the challenge in engineering a practical implementation is properly accounting for the wide range of data types in a real programming language. Java contains 8 primitive types in addition to types deriving from `java.lang.Object`, that are themselves divided into array and non-array types.[7]

Figures 4.3, 4.4, and 4.5 demonstrate how field size and object type specialization was implemented in the FLEX transaction runtime. The main header and source files did nothing but repeatedly `#include` an `-impl`-suffixed version of the file with `VALUETYPE`, `VALUENAME`, and `ARRAY` defined to range over the possible primitive and array types. This allows us to compactly name and define specialized functions, accounting for array/object (among other) differences. For example, the `FIELDBASE` macro (defined in Figure 4.4, used in Figure 4.3) allows us to treat object fields and array elements uniformly, even though the first array element starts at a different offset from the first field (since array data starts with an immutable length field).

Chapter 4.4.3 will discuss some of the other challenges involved in synthesizing atomic operations on subword and multi-word datatypes.

## 4.4 Limitations

The present implementation contains a number of non-fundamental limitations with well-understood solutions that, nonetheless, would add additional implementation complexity. These involve static fields, coarse granularity of LL/SC instructions, sub-word and multi-word fields, and condition variables (`Object.wait()`). In addition, there are limitations related to the

[7] Our flag value was carefully chosen so as not to be a NaN for either of the floating-point types, since comparisons against NaN can be surprising, and to consist of a repeated byte value so that a new object can be initialized with all fields `FLAG` without knowledge of the exact types and locations of each field.

```
class C {
  static int f;
  ...
  void foo() {
    ... = C.f;
    C.f = ...;
  }
}
```

⇒

```
class C$$static {
  int f;
}
class C {
  final static C$$static $static = new C$$static();
  ...
  void foo() {
    ... = C.$static.f;
    C.$static.f = ...;
  }
}
```

Figure 4.6: Static field transformation. Static fields of class `C` have been hoisted into a new class `C$$static`. Since the new field `$static` is write-once during initialization, it is safe to read/write directly, unlike the old field `f`.

manipulation of large objects (usually arrays); we will put off to chapter 5 a full discussion of the large object problem and its solutions.

### 4.4.1 Static fields

Static fields are not encapsulated in an object as other fields are; they are really a form of controlled-visibility global variable. The present implementation ignores static fields when performing the synchronization transformation; reads and writes to static fields are always direct. In theory this could cause races, but in our benchmarks most static fields are written only during initialization. An analysis that proved that writes only occurred during a single-threaded class initialization phase of execution could prove that these direct accesses are safe in most cases. Proper handling of static fields that do not meet this condition is straight-forward: new singleton classes would be created encapsulating these fields, and access would be via an extra indirection. Figure 4.6 illustrates this transformation.

### 4.4.2 Coarse-grained LL/SC

In our description of the software transaction algorithms we have so far neglected to state the size of the reservation created by the platform load-

linked instruction. The implicit assumption is that the reservation is roughly word-sized: our algorithm uses load-linked to watch the pointer-valued `readerList` (figures 3.8, 3.11) and `version` (figures 3.9, 3.11, 3.12) fields, and to perform the compare-and-swap operation on the transaction status field necessary for aborting and committing transactions. However, implementations of load-linked and store-conditional on most architectures usually place a reservation on a specific cache line, which spans many words. In the case of our MPC7447 (G4) PowerPC processor, a reservation (and cache line) is 32 bytes.

In our present implementation we have not taken any special action to account for this. In general, a larger-than-expected reservation may cause false conflicts between independent transactions, and this can lead to deadlock or livelock if contention is not managed properly. Our implementation uses randomized[8] exponential backoff, which ensures that every transaction eventually has an opportunity to execute in isolation, which will moot any false conflict.

Other mechanisms to resolve contention can also be used to address false conflicts. In an extreme case, a contention manager could invoke a copying collector to relocate objects involved in a false conflict, although this is likely merited only when the conflicts are extremely frequent or when long-lived transactions make the copying cost less than the cost of serializing the transactions involved.

Alternatively, false conflicts can be managed with careful allocation policy. The allocator in the present implementation ensures 8-byte alignment of objects. As we have done in prior work [69], the Java allocator can be modified to align objects to cache line (32-byte) boundaries, to ensure there are no false conflicts between objects. Some of the wasted space can be reclaimed by using a smaller alignment for objects known not to escape from their thread, or by packing multiple objects into a cache line iff they are

[8] The randomization is via the varying latency of the Unix `sleep` syscall; stronger randomization could be implemented if it mattered.

known never to participate in transactions that might conflict.

### 4.4.3 Handling subword and multi-word reads/writes

The granularity of the store-conditional instruction is more problematic. Although we have specified our algorithms assuming that a store-conditional of any field is possible, in practice store-conditional operations are only provided for a restricted set of data types: the PowerPC architecture defines only 32-bit and 64-bit stores, and 32-bit implementations (like our G4) don't implement the 64-bit variant. We need to carefully consider how to safely implement our algorithms with only a word-sized store-conditional.

First let us consider the case where the field size is larger than a word; for a Java implementation this corresponds to the `double` and `long` types. Within a transaction we can simply write two `int`s (for example) to implement the store of a `long`; the transactional mechanism already ensures that the multiple writes will appear atomic.

This leaves large writes outside of a transaction. One solution is to decompose these fields into multiple smaller fields as we did in the transactional case; we don't give up strong atomicity but we allow other readers (both transactional and non-transactional) to see the partial writes. The Java memory model actually permits this behavior (Java Language Specification chapter 17.7, "Non-atomic Treatment of double and long" [24]). The specification continues, "VM implementers are encouraged to avoid splitting their 64-bit values where possible." We can easily implement this behavior as an alternative: non-transactional writes of doubles and longs can be converted to small transactions encompassing nothing but the multiple word-size writes.

Now let us consider sub-word-sized writes. Chapter 17.6 of the Java Language Specification specifically says:

> One implementation consideration for Java virtual machines is that every field and array element is considered distinct; updates

> to one field or element must not interact with reads or updates of any other field or element. In particular, two threads that update adjacent elements of a byte array separately must not interfere or interact and do not need synchronization to ensure sequential consistency.
>
> Some processors do not provide the ability to write to a single byte. It would be illegal to implement byte array updates on such a processor by simply reading an entire word, updating the appropriate byte, and then writing the entire word back to memory. This problem is sometimes known as *word tearing*, and on processors that cannot easily update a single byte in isolation some other approach will be required.

Again, writes within a transaction are straight-forward: reading an entire word, updating it, and rewriting will appear atomic due to the transaction mechanism, and no observable word tearing will occur. A minor difficulty is the possibility of "false" conflicts between updates to adjacent fields of an object; these are handled as described in chapter 4.4.2.

The final case is then non-transactional writes to sub-word values, which occur frequently in real programs during string manipulations. We could address this by using a short transaction to do the read-modify-write of the word surrounding the sub-word field, but this is likely too expensive, considering the number of sub-word manipulations in most programs. A better algorithm to address this problem is presented in Figure 4.7; compare this to the earlier expression of the `writeNT` algorithm in Figure 3.15. The revised algorithm steals the lower two bits of the `readerList` pointer to serve as subword reservations, ensuring that racing writes to different subwords in the same word are prevented. The cost of this algorithm is mostly extra masking when the readerList is read, and an extra store (to set the `readerList` reservation) on every sub-word write.[9]

[9] It would be nice if we could avoid updating the `readerList` pointer on every write

```
void writeNT(struct oobj *obj, int byte_idx, small_field_t val) {
  if (unlikely(val==FLAG))
    unusualWrite(obj,byte_idx,val); // do a short transactional write
  else {
    do {
      int r = (int) LL(&(obj->readerList));
      if (unlikely((r & ~3) != 0)) {
        // readerList is not "NULL"; have to kill readers, etc.
        unusualWrite(obj,idx,val);
        return;
      } else if (r == (byte_idx & 3)) {
        // okay to write to this subword
        word_t w = obj->word[byte_idx>>2];
        if (SC(&(obj->word[byte_idx>>2]), combine(w, val, byte_idx)))
          return;
      } else
        // last write was to some other subword; switch to this one.
        SC(&(obj->readerList), byte_idx & 3);
    } while (1);
  }
}
```

Figure 4.7: Non-transactional write to a small (sub-word) field. Compare to Figure 3.15. The `byte_idx` value is the offset of the field within the objects, in bytes, and the `combine` function does the appropriate shifting and masking to combine the new sub-word value with the read value of the surrounding word.

Our present implementation does not specially handle sub-word-sized writes, resulting in programmer-visible word tearing.

### 4.4.4 Condition variables

A condition variable is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. When the predicate becomes true, a thread signals the condition (`Object.notify()` and `Object.notifyAll()` in Java); other threads can suspend themselves waiting for the signal (`Object.wait()`). In order to prevent races between notification and waiting, condition variables are

when using common access patterns, like stepping linearly through the array, but I don't see a way to do that.

```
public class Drop {
    //Message sent from producer to consumer.
    private String message;
    //True if consumer should wait for producer to send message, false
    //if producer should wait for consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        //Wait until message is available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = true;
        //Notify producer that status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        //Wait until message has been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = false;
        //Store message.
        this.message = message;
        //Notify consumer that status has changed.
        notifyAll();
    }
}
```

Figure 4.8: Drop box example illustrating the use of condition variables in Java, from [70].

usually associated with a mutex (the object monitor, in Java) and `wait` and `notify` require the lock to be held when they are called. The `wait` operation atomically releases the lock and waits for the variable; when notification is received `wait` re-acquires the lock before resuming. An example of the use of these operations in Java is provided in Figure 4.8.

Implementing these semantics properly in a transactional context is challenging; a better solution to the fundamental problem is a mechanism like Harris and Fraser's ***guarded transactions*** [30]. However, a reasonable implementation is possible.

The JDK1.6 specification for the `Object.wait()` method says:

> A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops...

This simplifies our implementation of the notification methods, since we needn't wait until the transaction doing the notify is sure to commit; it is always safe to go ahead and do the notify immediately; if we end up getting aborted we'll just redo the notify later when we retry. So we acquire a special mutex (either per-object as with standard Java or global; the lock is not held long), do the notification, and release the mutex.

The `wait` method, however, is a commit point, since it releases and reacquires the monitor lock. Thus it splits the transaction containing it into two.[10] The fundamental mechanism is straight-forward. We first acquire our special mutex, then attempt to commit the transaction, and check the result. If we don't commit successfully, we unlock the mutex and throw the usual `TransactionAbortException`. If we have committed successfully, we

[10] Which arguably calls for a special type annotation on methods that may call `wait()`, so that users are not misled into thinking a block is atomic that may invoke something that calls `wait()`.

can safely wait (releasing our mutex atomically). When we're woken, we create a new transaction, release our mutex, and continue. The mutex prevents notification from occurring between the commit and the wait.[11]

Implementation in a practical system introduces some difficulties. As described in chapter 4.2, the transaction object is passed as the first argument when calling methods within a transaction context. Since `Object.wait()` commits the initial transaction and begins a new one, we must either factor out a continuation after the call to `Object.wait()` which can be invoked with the new transaction after the wait is complete, or else provide a means to update the active transaction pointer and the retry point. One way to update the active transaction pointer is to use indirection: instead of keeping a direct pointer to the transaction object, we can keep a pointer to a cell containing the pointer; the cell can then be updated to point to the new transaction by `Object.wait()`. Alternatively, an extra return value can be added to methods called within a transaction, so that method can return an updated transaction object as well as their usual return value. Either of these transformations can be applied selectively to only those methods that could possibly invoke, directly or indirectly, `Object.wait()`. The more difficult challenge is to update the location at which we'll resume for another attempt when the transaction is aborted; this retry point should be immediately following the `wait()`. The retry mechanism becomes much more complicated, and some means to reconstruct the appropriate call-stack is necessary.

Note that for code like the example given in Figure 4.8 the "correct" resumption behavior is actually identical to restarting the transaction from

[11] Note in the drop box example in Figure 4.8 that the standard monitor lock is present to prevent (for example) `put()` from setting `empty` to false and doing the `notifyAll()` *after* `take()` has seen that `empty` is true but *before* the `wait()` has occurred (hence the notification would be lost). With the transaction transformation, the lost notification is prevented because put's write to `empty` will abort the `take` transaction if it occurs before `take` is committed (causing the `wait()` to be retried), and the special mutex will prevent the notification from occurring between the commit and the `wait`.

the original start point (ie, the beginning of the `put()` or `take()` method), assuming that the `put()` or `take()` isn't in a subsumed nested transaction. It is probably worth identifying this situation through compile-time analysis, as it simplifies the resumption transformation considerably.

In our present implementation we implement correct `notify` semantics, but we do not allow `wait` to be a commit point. Instead we take the special mutex and sleep on the condition variable without attempting to commit the transaction, and resume executing in the same transaction when we awake. This is sufficient for the limited uses of `wait` in our chosen benchmarks.

## 4.5 Performance

In this section we will evaluate the performance of our implementation. All experiments were performed on a 1GHz MPC7447 (G4) PowerPC processor, with 512kB unified L2 cache and 512MB of main memory, running Ubuntu 6.10 on a Linux 2.6.17 kernel. The limitations noted in the previous section counsel that the numbers we obtain ought to be considered guidelines to potential performance, not fundamental limits. Previous work has adequately demonstrated the scalability benefits of non-blocking synchronization; I will concentrate in this thesis on single-threaded efficiency measures.

We will be examining the SPECjvm98 benchmark suite, a collection of practical Java programs including an expert system, a simple database, a Java compiler, an audio encoder, and a parser generator. To clarify our measurements, our baseline for all of our comparisons will be a version of the benchmarks compiled with the same method cloning and desugaring (chapter 4.2) that is done for the transaction transformation.

In chapter 3.4 we examined a microbenchmark to discover the lower limits on our non-transactional overhead. These are the fundamental costs of strong atomicity: the penalty we must pay even if we are not using transactions at all.[12] Figure 4.9 shows equivalent measurements on full Java

[12] Although, of course, in practice one would turn off the transaction transformation

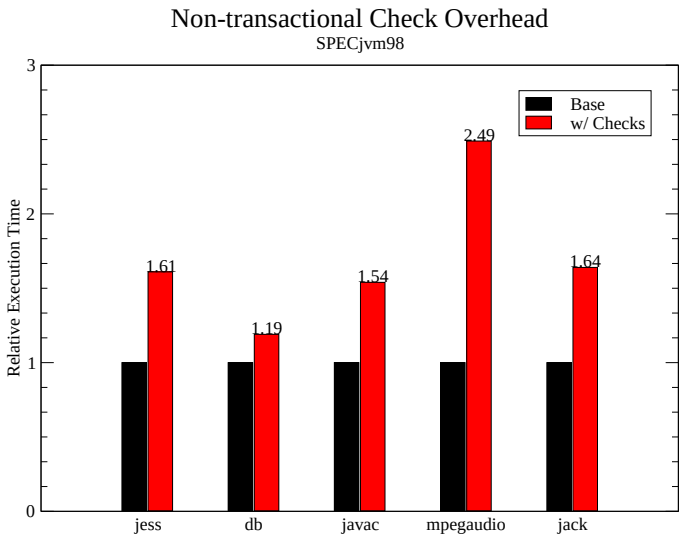



Figure 4.9: Non-transactional check overhead for SPECjvm98. Both benchmarks in each pair were compiled with the transformations in chapter 4.2 (method cloning, desugaring) and all synchronization was removed. The version "with checks" had all reads and writes transformed using the `readNT` and `writeNT` protocols, while the base version performed direct field access.

| Benchmark | reads | false flags read | writes | false flags written |
|---|---|---|---|---|
| jess | 25,583,878 | 0 (0%) | 703,303 | 0 (0%) |
| db | 11,078,177 | 0 (0%) | 912,965 | 0 (0%) |
| javac | 10,315 | 0 (0%) | 2,312 | 0 (0%) |
| mpegaudio | 239,928,276 | 2,056 (< 0.001%) | 42,164,416 | 9,132 (.02%) |
| jack | 9,882,846 | 0 (0%) | 4,595,774 | 0 (0%) |

Figure 4.10: Number of false flags read/written in SPECjvm98 benchmarks. To compile this table the applications were run with input size "10".

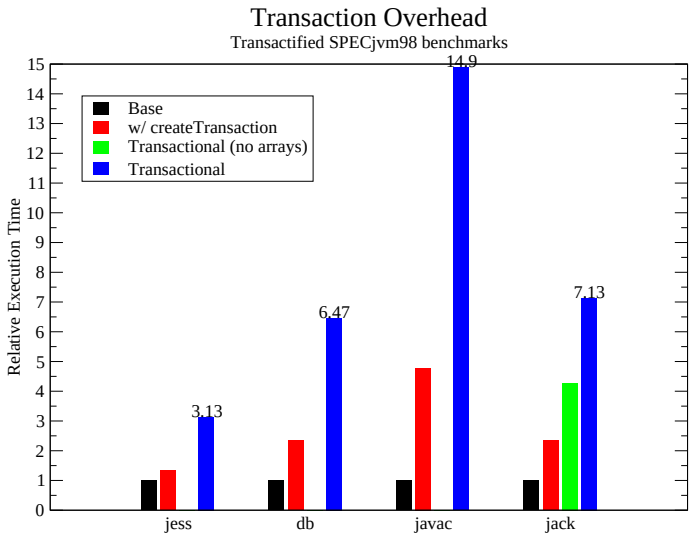



Figure 4.11: Transaction overhead for SPECjvm98. The leftmost bar is compiled with method cloning and desugaring, all transactions removed, and direct reads and writes. The next bar adds creation of commit records at the start of each transaction and the CAS commit operation at the end. The next bar uses the transactional and non-transactional read and write protocols for non-array objects. The final bar is fully transactional.

applications. We have removed all the synchronization in these benchmarks (safe because they are single-threaded) leaving only the costs of the `readNT` and `writeNT` protocols.

Our earlier microbenchmark hinted that we might expect non-transactional overheads as low as 15%, although 20% was the best we could obtain with a C implementation. In fact, we see a 19% overhead on the `209_db` benchmark, although the other benchmarks show that an overhead of around 60% is more typical. Close inspection of the assembly code emitted by the C compiler for these benchmarks indicates that the compiler's limitations are the primary reason our performance falls short of mark set by the microbenchmark. As discussed in chapter 4.2, it is hard to obtain maximum performance with a C implementation (such as we are using here) due to expressive limits, and the emitted assembly indicates that the compiler is not optimally acting on the hints we can give it. In particular, we want as little extraneous code as possible on the fast path through the common case, but the compiler adds register spills and moves here that are necessary only in unusual cases. Further, because the fundamental load-linked and store-conditional instructions are inside inline assembly blocks and thus opaque to the compiler, many opportunities for code motion and scheduling are being missed.

The `222_mpegaudio` benchmark shows another potential liability: it incurred a 149% overhead for the non-transactional code. The "false flag" counts shown in Figure 4.10 hint at the problem: this benchmark is the only one to encounter false flags. The hidden portion of the iceberg may partly consist of inefficiencies in our current implementation of byte-sized reads and writes, but since false flags cause creation of new version objects, most of the cost is probably a reflection of the "large object problem", which we will discuss further in a moment.

Figure 4.11 demonstrates the performance of transactional code, broken

completely if static or dynamic analysis indicated a program is single-threaded, for example if no `Thread` objects are every created.

down into the components of the final cost. We see that creation of transaction objects at the start of every atomic region is a considerable fraction of the total cost. If transactions are going to be frequent and small, this cost must be kept in line.

Transactions that mutate arrays also account for a large amount of the total cost. In the extreme case, the SPECjvm98 benchmark `201_compress` allocates a 1MB array and writes compressed data into it. The design of our transaction system necessitates an object copy for every unique object written to inside a transaction; if the objects are large (such as the arrays in `201_compress` and `222_mpegaudio`), then this cost becomes excessive. The next chapter will discuss this problem at length and propose a solution based on functional array data structures.

Although our transaction design ought to be able to perform direct read and writes on version objects in long-lived transactions, these performance numbers indicate that we are not achieving our goal.

## 4.6 Additional optimizations

In this section we will discuss some additional optimizations that could improve our performance on transactional code; they have not been implemented in the present compiler. The improvements include version passing, escape analysis, object immutability analysis, and fast allocation.

As described in chapter 4.2, the basic transaction transformation inserts a call to `ensureReader/ensureWriter` once per object prior to a read/write of a field in the object. These methods return a version object (the "current version") which is then passed to the `readT/writeT` methods. A straightforward improvement would avoid redundant calls to `ensureReader/ensureWriter` by passing the "current version" of the various parameters when a method is invoked. A strictness analysis should be performed, to avoid marking fields as read/written that are not guaranteed to be read/written by the transaction, and the "current version" added as

an extra argument for the strict parameters of the method. This allows the caller to combine redundant `ensureReader`/`ensureWriter` invocations for the argument.[13]

Escape analysis [68] is a standard technique to reduce unnecessary synchronization in Java programs. In a typical application, monitor locks on objects that do not escape their thread are eliminated; these lock eliminations can translate to transaction eliminations as well. Further, escape analysis can identify individual objects that do not escape their method, thread, or call context, and replace the `readT`/`writeT` or `readNT`/`writeNT` protocol with direct access.

Another orthogonal analysis can identify object immutability [72]. Immutable objects can be read directly and their field values can be safely cached across transaction boundaries. Further, the initialization of immutable objects typically can be done with direct writes as well.

Finally, a substantial portion of the cost of small transactions is spent allocating the version object, especially for the conservative collector used in the present implementation. Fast allocators could alleviate this bottleneck [6]. Alternatively, a free list of versions could be hung from objects involved in many transactions: as soon as a version commits, the previous version can be added to the free list for the use of the next transaction.

[13] One could also consider adding "current version" arguments even for non-strict parameters, with the caveat that the caller pass in NULL if it had not already read/written the non-strict parameter. However, all variables used in a method or the method's callees would then be potentially eligible for version passing; some more sophisticated heuristic would be needed to determine for which objects version passing is worthwhile.

Well, you go in and you ask for some toothpaste—the small size—and the man brings you the large size. You tell him you wanted the small size but he says the large size is the small size. I always thought the large size was the largest size, but he says that the family size, the economy size and the giant size are all larger than the large size—that the large size is the smallest size there is.

*Charade* (1963)

# Chapter 5

# Arrays and large objects

The software transaction system implemented in the previous chapter clones objects on transactional writes, so that the previous state of the object can be restored if the transaction aborts. Figure 5.1 shows the object size distribution of transactional writes for SPECjvm98, and indicates that over 10% of writes may be to large objects. As we've seen in chapter 4.5, the copying cost can become excessive.

The solution I will propose will represent objects as ***functional arrays***. O'Neill and Burton [56] give a fairly inclusive overview of such algorithms; I've chosen Tyng-Ruey Chuang's version [15] of *shallow binding*, which uses randomized cuts to the version tree to limit the cost of a read to $O(n)$ in the worst case. Single-threaded accesses to the array are $O(1)$. Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang's scheme is attractive because it limits the worst-case cost of an abort, with very little added complexity.

In this chapter I will recast the transaction system design of chapter 3 as a "small object protocol," then show how to extend it to a "large object protocol," in the process addressing the large object performance problems. The large object protocol will use a lock-free variant of Chuang's algorithm, which I will present in chapter 5.4.

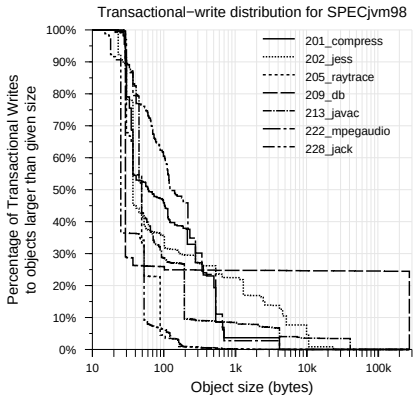



Figure 5.1: Proportion of transactional writes to objects equal to or smaller than a given size.

## 5.1 Basic operations on functional arrays

Let us begin by reviewing the basic operations on functional arrays. Functional arrays are *persistent*; that is, after an element is updated both the new and the old contents of the array are available for use. Since arrays are simply maps from integers (indexes) to values; any functional map datatype (for example, a functional balanced tree) can be used to implement functional arrays.

However, the distinguishing characteristic of an imperative array is its time complexity: $O(1)$ access or update of any element. Implementing functional arrays with a functional balanced tree yields $O(\lg n)$ worst-case access or update.[1]

For concreteness, functional arrays have the following three operations defined:

- FA-CREATE($n$): Return an array $A$ of size $n$. The contents of the array are initialized to zero.

[1] I will return to a discussion of operational complexity in chapter 5.4.

- FA-UPDATE$(A, i, v)$: Return an array $A'$ that is functionally identical to array $A$ except that FA-READ$(A', i) = v$. Array $A$ is not destroyed and can be accessed further.

- FA-READ$(A, i)$: Return $A(i)$ (that is, the value of the $i$th element of array $A$).

We allow any of these operations to *fail*. Failed operations can be safely retried, as all operations are idempotent by definition.

For the moment, consider the following naïve implementation:

- FA-CREATE$(n)$: Return an ordinary imperative array of size $n$.

- FA-UPDATE$(A, i, v)$: Create a new imperative array $A'$ and copy the contents of $A$ to $A'$. Set $A'[i] = v$. Return $A'$.

- FA-READ$(A, i)$: Return $A[i]$.

This implementation has $O(1)$ read and $O(n)$ update, so it matches the performance of imperative arrays only when $n = O(1)$. I will therefore call these *small object functional arrays*. Operations in this implementation never fail. Every operation is non-blocking and no synchronization is necessary, since the imperative arrays are never mutated after they are created. In chapter 5.4 we will review better implementations of functional arrays, and present our own lock-free variant.

## 5.2 A single-object protocol

Given a non-blocking implementation of functional arrays, we can construct a transaction implementation for single objects. In this implementation, fields of at most one object may be referenced during the execution of the transaction.

I will consider the following two operations on objects:

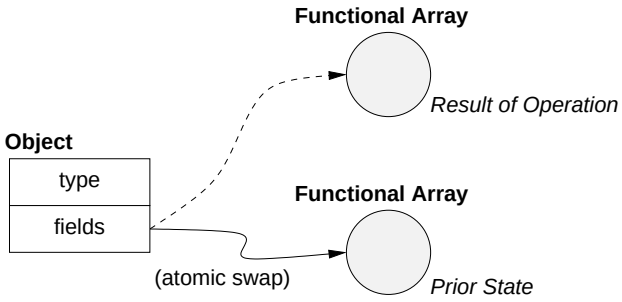



Figure 5.2: Implementing non-blocking single-object concurrent operations with functional arrays.

- READ(o, f): Read field f of o. We will assume that there is a constant mapping function that given a field name returns an integer index. We will write the result of mapping f as f.index. For simplicity, and without loss of generality, we will assume all fields are of equal size.
- WRITE(o, f, v): Write value v to field f of o.

All other operations on Java objects, such as method dispatch and type interrogation, can be performed using the immutable type field in the object. Because the type field is never changed after object creation, non-blocking implementations of operations on the type field are trivial.

As Figure 5.2 shows, our single-object transaction implementation represents objects as a pair, combining type and a reference to a functional array. When not inside a transaction, object reads and writes are implemented using the corresponding functional array operation, with the array reference in the object being updated appropriately:

- READ(o, f): Return FA-READ(o.fields, f.index).
- WRITE(o, f, v): Replace o.fields with the result of FA-UPDATE(o.fields, f.index, v).

The interesting cases are reads and writes inside a transaction. At entry to our transaction that will access (only) object o, we store o.fields in a

local variable $u$. We create another local variable $u'$ which we initialize to $u$. Then our read and write operations are implemented as:

- READT$(o, f)$: Return FA-READ$(u', f.\texttt{index})$.
- WRITET$(o, f, v)$: Update variable $u'$ to the result of FA-UPDATE$(u', f.\texttt{index}, v)$.

At the end of the transaction, we use Compare-And-Swap to atomically set $o.\texttt{fields}$ to $u'$ iff it contained $u$. If the CAS fails, we the transaction is aborted (we simply discard $u'$) and we retry.

With our naïve "small object" functional arrays, this implementation is exactly the "small object protocol" of Herlihy [34]. Herlihy's protocol is rightly criticized for an excessive amount of copying. I will address this with a better implementation of functional arrays in chapter 5.4. However, the restriction that only one object may be referenced within a transaction is overly limiting. I will first fix this problem.

## 5.3 Extension to multiple objects

I extend the implementation to allow the fields of any number of objects to be accessed during the transaction. Figure 5.3 shows our new object representation. Compare this to Figure 3.6; we've successfully recast our earlier transaction system design now in terms of operations on an array datatype. Objects consist of two slots, and the first represents the immutable `type`, as before. The second field, `versions`, points to a linked list of `Version` structures. The `Version` structures contain a pointer `fields` to a functional array, and a pointer `owner` to an *transaction identifier*. The transaction identifier contains a single field, `status`, which can be set to one of three values: *COMMITTED*, *IN-PROGRESS*, or *ABORTED*. When the transaction identifier is created, the status field is initialized to *IN-PROGRESS*, and it will be updated exactly once thereafter, to either *COMMITTED* or

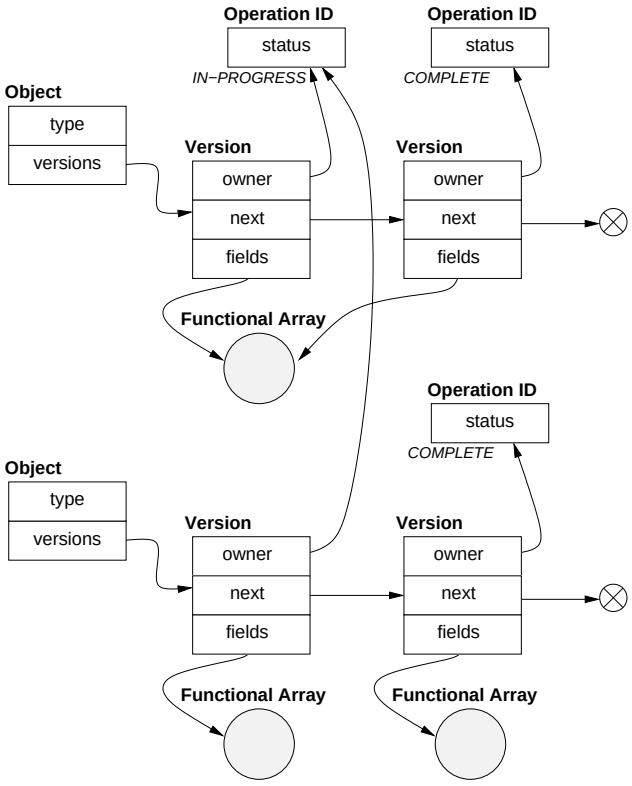



Figure 5.3: Data structures to support non-blocking multi-object concurrent operations. Objects point to a linked list of versions, which reference transaction identifiers. Versions created within the same execution of a transaction share the same transaction identifier. Version structure also contain pointers to functional arrays, which record the values for the fields of the object. If no modifications have been made to the object, multiple versions in the list may share the same functional array. (Compare this model of a transaction system to our concrete design in Figure 3.6.)

```
READ(o, f):
begin
retry:
  u ← o.versions
  u′ ← u.next
  s ← u.owner.status
  if (s = DISCARDED)                          [Delete DISCARDED?]
    CAS(u, u′, &(o.versions))
    goto retry
  else if (s = COMPLETE)
    a ← u.fields                                [u is COMPLETE]
    u.next ← null                               [Trim version list]
  else
    a ← u′.fields                               [u′ is COMPLETE]
  return FA-READ(a, f.index)                    [Do the read]
end

READT(o, f):
begin
  u ← o.versions
  if (oid = u.owner)                          [My OID should be first]
    return FA-READ(u.fields, f.index)           [Do the read]
  else                                          [Make me first!]
    u′ ← u.next
    s ← u.owner.status
    if (s = DISCARDED)                        [Delete DISCARDED?]
      CAS(u, u′, &(o.versions))
    else if (oid.status = DISCARDED)            [Am I alive?]
      fail
    else if (s = IN-PROGRESS)                 [Abort IN-PROGRESS?]
      CAS(s, DISCARDED, &(u.owner.status))
    else                                        [Link new version in:]
      u.next ← null                             [Trim version list]
      u′ ← new Version(oid, u, null)            [Create new version]
      if (CAS(u, u′, &(o.versions)) ≠ FAIL)
        u′.fields ← u.fields                    [Copy old fields]
    goto retry
end
```

Figure 5.4: READ and READT implementations for the multi-object protocol.

```
WRITE(o, f, v):
begin
retry:
  u ← o.versions
  u′ ← u.next
  s ← u.owner.status
  if (s = DISCARDED)                          [Delete DISCARDED?]
    CAS(u, u′, &(o.versions))
  else if (s = IN-PROGRESS)                   [Abort IN-PROGRESS?]
    CAS(s, DISCARDED, &(u.owner.status))
  else                                        [u is COMPLETE]
    u.next ← null                             [Trim version list]
    a ← u.fields
    a′ ← FA-UPDATE(a, f.index, v)
    if (CAS(a, a′, &(u.fields)) ≠ FAIL)       [Do the write]
      return                                  [Success!]
  goto retry
end

WRITET(o, f, v):
begin
  u ← o.versions
  if (oid = u.owner)                          [My OID should be first]
    u.fields ← FA-UPDATE(u.fields, f.index, v)[Do write]
  else                                        [Make me first!]
    u′ ← u.next
    s ← u.owner.status
    if (s = DISCARDED)                        [Delete DISCARDED?]
      CAS(u, u′, &(o.versions))
    else if (oid.status = DISCARDED)          [Am I alive?]
      fail
    else if (s = IN-PROGRESS)                 [Abort IN-PROGRESS?]
      CAS(s, DISCARDED, &(u.owner.status))
    else                                      [Link new version in:]
      u.next ← null                           [Trim version list]
      u′ ← new Version(oid, u, null)          [Create new version]
      if (CAS(u, u′, &(o.versions)) ≠ FAIL)
        u′.fields ← u.fields                  [Copy old fields]
    goto retry
end
```

Figure 5.5: WRITE and WRITET implementations for the multi-object protocol.

*ABORTED*. A *COMMITTED* transaction identifier never later becomes *IN-PROGRESS* or *ABORTED*, and a *ABORTED* transaction identifier never becomes *COMMITTED* or *IN-PROGRESS*.

We create an transaction identifier when we begin or restart a transaction and place it in a local variable *tid*. At the end of the transaction, we use CAS to set *tid*.`status` to *COMMITTED* iff it was *IN-PROGRESS*. If the CAS is successful, the transaction has also executed successfully; otherwise *tid*.`status` = *ABORTED* (which indicates that our transaction has been aborted) and we must back off and retry. All `Version` structures created while in the transaction will reference *tid* in their `owner` field.

Semantically, the current field values for the object will be given by the first version in the versions list whose transaction identifier is *COMMITTED*. This allows us to link *IN-PROGRESS* versions in at the head of multiple objects' versions lists and atomically change the values of all these objects by setting the one common transaction identifier to *COMMITTED*. We only allow one *IN-PROGRESS* version on the versions list, and it must be at the head, so before we can link a new version at the head we must ensure that every other version on the list is *ABORTED* or *COMMITTED*.

Since we will never look past the first *COMMITTED* version in the versions list, we can free all versions past that point. In our presentation of the algorithm, we do this by explicitly setting the `next` field of every *COMMITTED* version we see to **null**; this allows the versions past that point to be garbage collected. An optimization would be to have the garbage collector do the list trimming for us when it does a collection.

We don't want to inadvertently chase the null `next` pointer of a *COMMITTED* version, so we always load the `next` field of a version *before* we load `owner.status`. Since the writes occur in the reverse order (*COMMITTED* to `owner.status`, then **null** to `next`) we have ensured that our `next` pointer is valid whenever the status is not *COMMITTED*.

We begin an atomic method with TransStart and attempt to complete an atomic method with TransEnd. They are defined as follows:

- TransStart: create a new transaction identifier, with its status initialized to *IN-PROGRESS*. Assign it to the thread-local variable *tid*.
- TransEnd: If

  $$\text{CAS}(\textit{IN-PROGRESS}, \textit{COMMITTED}, \&(\textit{tid}.\texttt{status}))$$

  is successful, the transaction as a whole has completed successfully, and can be linearized at the location of the CAS. Otherwise, the transaction has been aborted. Back off and retry from TransStart.

Pseudo-code describing Read, Write, ReadT, and WriteT is presented in Figures 5.4 and 5.5. In the absence of contention, all operations take constant time plus an invocation of FA-Read or FA-Update.

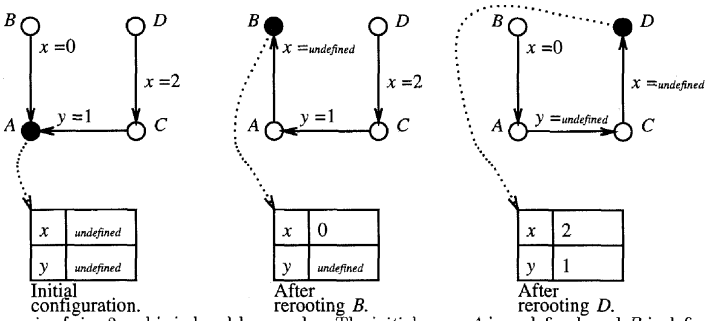
## 5.4 Lock-free functional arrays

In this section I will present a lock-free implementation of functional arrays with $O(1)$ performance in the absence of contention. The crucial operation is a rotation of a *difference node* with the main body of the array. Using this implementation of functional arrays in the multi-object transaction protocol of the previous chapter will complete our re-implementation of non-blocking transactions, solving the large object problem.

Let's begin by reviewing the well-known functional array implementations. As mentioned previously, O'Neill and Burton [56] give an inclusive overview. Functional array implementations fall generally into one of three categories: *tree-based*, *fat-elements*, or *shallow-binding*.

Tree-based implementations typically have a logarithmic term in their complexity. The simplest is the persistent binary tree with $O(\ln n)$ look-up time; Chris Okasaki [55] has implemented a purely-functional random-access list with $O(\ln i)$ expected lookup time, where $i$ is the index of the desired element.

Fat-elements implementations have per-element data structures indexed by a master array. Cohen [17] hangs a list of versions from each element in




NOTE. The array is of size 2 and is indexed by $x$ and $y$. The initial array $A$ is undefined, and $B$ is defined as an update to $A$ at index $x$ by value 0. Similarly for $C$ and $D$. The dark node is the root node which has the cache. White nodes are differential nodes which must first be rerooted before be read. Note that only the root node has the cache.

Figure 5.6: Shallow binding scheme for functional arrays, from [15, Figure 1].

the master array. O'Neill and Burton [56], in a more sophisticated technique, hang a splay tree off each element and achieve $O(1)$ operations for single-threaded use, $O(1)$ amortized cost when accesses to the array are "uniform", and $O(\ln n)$ amortized worst case time.

Shallow binding was introduced by Baker [9] as a method to achieve fast variable lookup in Lisp environments. Baker clarified the relationship to functional arrays in [8]. Shallow binding is also called *version tree arrays*, *trailer arrays*, or *reversible differential lists*. A typical drawback of shallow binding is that reads may take $O(u)$ worst-case time, where $u$ is the number of updates made to the array. Tyng-Ruey Chuang [15] uses randomized cuts to the version tree to limit the cost of a read to $O(n)$ in the worst case. Single-threaded accesses are $O(1)$.

Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang's scheme is attractive because it limits the worst-case cost of an abort, with very little added complexity. In this section I will present a lock-free version of Chuang's randomized algorithm.

In shallow binding, only one version of the functional array (the *root*) keeps its contents in an imperative array (the *cache*). Each of the other versions is represented as a path of *differential nodes*, where each node describes the differences between the current array and the previous array. The difference is represented as a pair $\langle index, value \rangle$, representing the new value to be stored at the specified index. All paths lead to the root. An update to the functional array is simply implemented by adding a differential node pointing to the array it is updating.

The key to constant-time access for single-threaded use is provided by the read operation. A read to the root simply reads the appropriate value from the cache. However, a read to a differential node triggers a series of rotations that swap the direction of differential nodes and result in the current array acquiring the cache and becoming the new root. This sequence of rotations is called *re-rooting*, and is illustrated in Figure 5.6. Each rotation exchanges the root nodes for a differential node pointing to it, after which

the differential node becomes the new root and the root becomes a differential node pointing to the new root. The cost of a read is proportional to its re-rooting length, but after the first read accesses to the same version are $O(1)$ until the array is re-rooted again.

Shallow binding performs badly if read operations ping-pong between two widely separated versions of the array, as we will continually re-root the array from one version to the other. Chuang's contribution is to provide for *cuts* to the chain of differential nodes: once in a while we clone the cache and create a new root instead of performing a rotation. This operation takes $O(n)$ time, so we amortize it over $n$ operations by randomly choosing to perform a cut with probability $1/n$.

Figure 5.7 shows the data structures used for the functional array implementation, and the series of atomic steps used to implement a rotation. The `Array` class represents a functional array; it consists of a `size` for the array and a pointer to a `Node`. There are two types of nodes: a `CacheNode` stores a value for every index in the array, and a `DiffNode` stores a single change to an array. `Array` objects that point to `CacheNodes` are roots.

In step 1 of the figure, we have a root array $A$ and an array $B$ whose differential node $d_B$ points to $A$. The functional arrays $A$ and $B$ differ in one element: element $x$ of $A$ is $z$, while element $x$ of $B$ is $y$. We are about to rotate $B$ to give it the cache, while linking a differential node to $A$.

Step 2 shows our first atomic action. We have created a new `DiffNode` $d_A$ and a new `Array` $C$ and linked them between $A$ and its cache. The `DiffNode` $d_A$ contains the value for element $x$ contained in the cache, $z$, so there is no change in the value of $A$.

We continue swinging pointers until step 5, when can finally set the element $x$ in the cache to $y$. We perform this operation with a DCAS operation that checks that $C$`.node` is still pointing to the cache as we expect. Note that a concurrent rotation would swing $C$`.node` in its step 1. In general, therefore, the location pointing to the cache serves as a reservation on the cache.

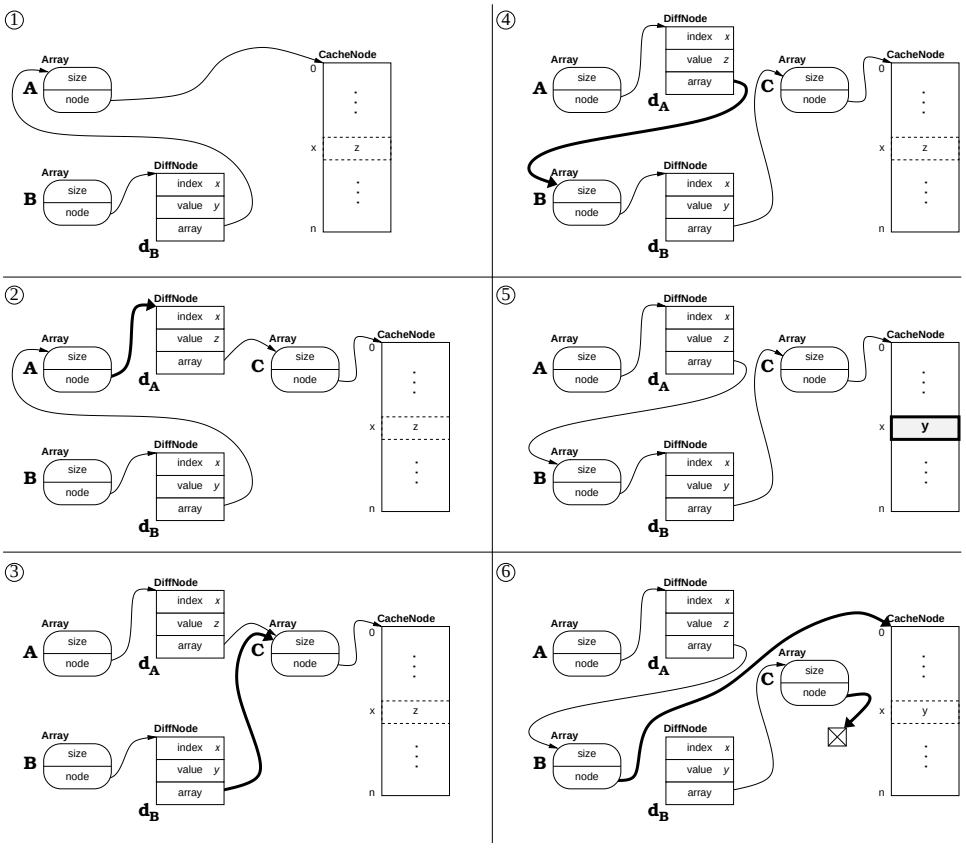



Figure 5.7: Atomic steps in FA-ROTATE(B). Time proceeds top-to-bottom on the left hand side, and then top-to-bottom on the right. Array $A$ is a root node, and FA-READ$(A, x) = z$. Array B has the almost the same contents as $A$, but FA-READ$(B, x) = y$.

```
FA-UPDATE(A, i, ν):
begin
  d ← new DiffNode(i, ν, A)
  A′ ← new Array(A.size, d)
  return A′
end

FA-READ(A, i):
begin
retry:
  d_C ← A.node
  if d_C is a cache, then
    ν ← A.node[i]
    if (A.node ≠ d_C)[consistency check]
      goto retry
    return ν
  else
    FA-ROTATE(A)
    goto retry
end
```

Figure 5.8: Implementation of lock-free functional array using shallow binding and randomized cuts (part 1).

```
FA-ROTATE(B):
begin
retry:
  d_B ← B.node     [step (1): assign names as per Figure 5.7.]
  A ← d_B.array
  x ← d_B.index
  y ← d_B.value
  z ← FA-READ(A, x)                    [rotates A as side effect]

  d_C ← A.node
  if d_C is not a cache, then
    goto retry

  if (0 = (random mod A.size))                              [random cut]
    d'_C ← copy of d_C
    d'_C[x] ← y
    s ← DCAS(d_C, d_C, &(A.node), d_B, d'_C, &(B.node))
    if (s ≠ SUCCESS) goto retry
    else return

  C ← new Array(A.size, d_C)
  d_A ← new DiffNode(x, z, C)

  s ← CAS(d_C, d_A, &(A.node))                              [step (2)]
  if (s ≠ SUCCESS) goto retry

  s ← CAS(A, C, &(d_B.array))                               [step (3)]
  if (s ≠ SUCCESS) goto retry

  s ← CAS(C, B, &(d_A.array))                               [step (4)]
  if (s ≠ SUCCESS) goto retry

  s ← DCAS(z, y, &(d_C[x]), d_C, d_C, &(C.node))            [step (5)]
  if (s ≠ SUCCESS) goto retry

  s ← DCAS(d_B, d_C, &(B.node), d_C, nil, &(C.node))        [step (6)]
  if (s ≠ SUCCESS) goto retry
end
```

Figure 5.9: Implementation of lock-free functional array using shallow binding and randomized cuts (part 2).

Thus in step 6 we need to again use DCAS to simultaneously swing C.node away from the cache as we swing B.node to point to the cache.

Figures 5.8 and 5.9 present pseudocode for FA-ROTATE, FA-READ, and FA-UPDATE. Note that FA-READ also uses the cache pointer as a reservation, double-checking the cache pointer after it finishes its read to ensure that the cache hasn't been stolen from it.

Let us now consider cuts, where FA-READ clones the cache instead of performing a rotation. Cuts also check the cache pointer to protect against concurrent rotations. But what if the cut occurs while a rotation is mutating the cache in step 5? In this case the only array adjacent to the root is B, so the cut must be occurring during an invocation of FA-ROTATE(B). But then the differential node $d_B$ will be applied after the cache is copied, which will safely overwrite the mutation we were concerned about.

Note that with hardware support for small transactions [36] we could cheaply perform the entire rotation atomically, instead of using this six-step approach.

People who are really serious about software should make their own hardware. Remember, it's all software, it just depends on when you crystallize it.

*Alan Kay*

# Chapter 6

# Transactions in hardware: Unbounded Transactional Memory

The previous chapters have detailed the design of an efficient software-only transaction system for object-oriented programs. Given hardware support, we can construct even more efficient transaction systems for certain types of transactions. In this chapter, we will present UTM, an implementation of unbounded transactional memory [5] which fully virtualizes transactions. We will follow this with LTM, a much simpler design which can be pin-compatible with today's processors. LTM supports more limited transactions, but we conclude by showing how LTM can be combined with the efficient software system we have already demonstrated to yield a hybrid system with a great deal of power and flexibility.[1]

[1] Portions of this chapter are adapted from [5], co-written with Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie.

## 6.1 The UTM architecture

We begin by describing UTM, a system that implements unbounded transactional memory in hardware. UTM allows transactions to grow (nearly) as large as virtual memory. It also supports a semantics for nested transactions, where interior transactions are subsumed into the atomic region represented by the outer transaction. Unlike previous schemes that tie a thread's transactional state to a particular processor and/or cache, UTM maintains bookkeeping information for a transaction in a memory-resident data structure, the ***transaction log***. This enables transactions to survive timeslice interrupts and process migration from one processor to another. We first present the software interface to UTM, and then describe the implementation details.

### 6.1.1 New instructions

UTM adds two new instructions to a processor's instruction set architecture:

`XBEGIN pc`**:** Begin a new transaction. The `pc` argument to `XBEGIN` specifies the address of an ***abort handler*** (e.g., using a PC-relative offset). If at any time during the execution of a transaction the hardware determines that the transaction must fail, it immediately rolls back the processor and memory state to what it was when `XBEGIN` was executed, then jumps to `pc` to execute the abort handler.

`XEND`**:** End the current transaction. If `XEND` completes, then the transaction is committed, and all of its operations appear to be atomic with respect to any other transaction.

Semantically, we can think of an `XBEGIN` instruction as a conditional branch to the abort handler. The `XBEGIN` for a transaction that fails has the behavior of a mispredicted branch. Initially, the processor executes the `XBEGIN` as a not-taken branch, falling through into the body of the transaction. Eventually the processor realizes that the transaction cannot

commit, at which point it reverts all processor and memory state back to the point of misprediction and branches to the abort handler.

In the same manner as our software implementation (chapter 4.2.1), UTM supports the nesting of transactions by subsuming the inner transaction. For example, within an outer transaction, a subroutine may be called that contains an inner transaction. UTM simply treats the inner transaction as part of the atomic region defined by the outer one. This strategy is correct, because it maintains the property that the inner transaction executes atomically. Subsumed nested transactions are implemented by using a counter to keep track of nesting depth. If the nesting depth is positive, then `XBEGIN` and `XEND` simply increment and decrement the counter, respectively, and perform no other transactional bookkeeping.

### 6.1.2 Rolling back processor state

The branch mispredict mechanism in conventional superscalar processors can roll back register state only for the small window of recent instructions that have not graduated from the reorder buffer. To circumvent the window-size restriction and allow arbitrary rollback for unbounded transactions, the processor must be modified to retain an additional snapshot of the architectural register state. A UTM processor saves the state of its architectural registers when it graduates an `XBEGIN`. The snapshot is retained either until the transaction aborts, at which point the snapshot is restored into the architectural registers, or until the matching `XEND` graduates indicating that the transaction has committed.

UTM's modifications to the processor core are illustrated in Figure 6.1. We assume a machine with a unified physical register file, and so rather than saving the architectural registers themselves, UTM saves a snapshot of the register-renaming table and ensures the corresponding physical registers are not reused until the transaction commits. The rename stage maintains an additional "saved" bit for each physical register to indicate which registers are part of the working architectural state, and takes a snapshot as

each branch or XBEGIN is decoded and renamed. When an XBEGIN instruction graduates, activating the transaction, the associated "S bit" snapshot will have bits set on exactly those registers holding the graduated architectural state. Physical registers are normally freed on graduation of a later instruction that overwrites the same architectural register. If the S bit on the snapshot for the active transaction is set, the physical register is added to a FIFO called a ***Register Reserved List*** instead of the normal ***Register Free List***. This prevents physical registers containing saved data from being overwritten during a transaction. When the transaction's XEND commits, the active snapshot's S bits are cleared and the Register Reserved List is drained into the regular Register Free List. In the event that the transaction aborts, the saved register-renaming table is restored and the reorder buffer is rolled back, as in an exception. After restoring the architectural register state, the branch is taken to the abort handler. Even though the processor can internally speculatively execute ahead through multiple transactions, transactions only affect the global memory system as instructions graduate, and hence UTM requires only a single snapshot of the architectural register state.

The current transaction abort handler address, nesting depth, and register snapshot are part of the transactional state. They are made visible to the operating system (as additional processor control registers) to allow them to be saved and restored on context switches.

### 6.1.3 Memory state

Previous HTM systems [46, 36] represent a transaction partly in the processor and partly in the cache, taking advantage of the coincidence between the cache-consistency protocol and the underlying consistency requirements of transactional memory. Unlike those systems, UTM transactions are represented by a single ***xstate*** data structure held in the memory of the system. The cache in UTM is used to gain performance, but the correctness of UTM does not depend on having a cache. In the following paragraphs, we first

describe the xstate and how the system uses it assuming there is no caching. Then, we describe how caching accelerates xstate operations.

The xstate is illustrated in Figure 6.2. The xstate contains a transaction log for each active transaction in the system. A transaction log is allocated by the operating system for each thread, and two processor control registers hold the base and bounds of the currently active thread's log. Each log consists of a ***commit record*** and a vector of ***log entries***. The commit record maintains the transaction's status: `PENDING`, `COMMITTED`, or `ABORTED`. Each log entry corresponds to a block of memory that has been read or written by the transaction. The entry provides a pointer to the block and the old (backup) value for the block so that memory can be restored in case the transaction aborts. Each log entry also contains a pointer to the commit record and pointers that form a linked list of all entries in all transaction logs that refer to the same block.

The final part of the xstate consists of a ***log pointer*** and one ***RW bit*** for each block in memory (and on disk, when paging). If the RW bit is `R`, any transactions that have accessed the block did so with a load; otherwise, if it is `W`, the block may have been the target of a transaction's store. When a processor running a transaction reads or writes a block, the block's log pointer is made to point to a transaction log entry for that block. Further, if the access is a write, the RW bit for the block is set to `W`. Whenever another processor references a block that is already part of a pending transaction, the system consults the RW bit and log pointer to determine the correct action, for example, to use the old value, to use the new value, or to abort the transaction.

When a processor makes an update as part of a transaction, the new value is stored in memory and the old value is stored in an entry in the transaction log. In principle, there is one log entry for every load or store performed by the transaction. If the memory allocated to the log is not large enough, the transaction aborts and the operating system allocates a larger transaction log and retries the transaction. When operating on the same

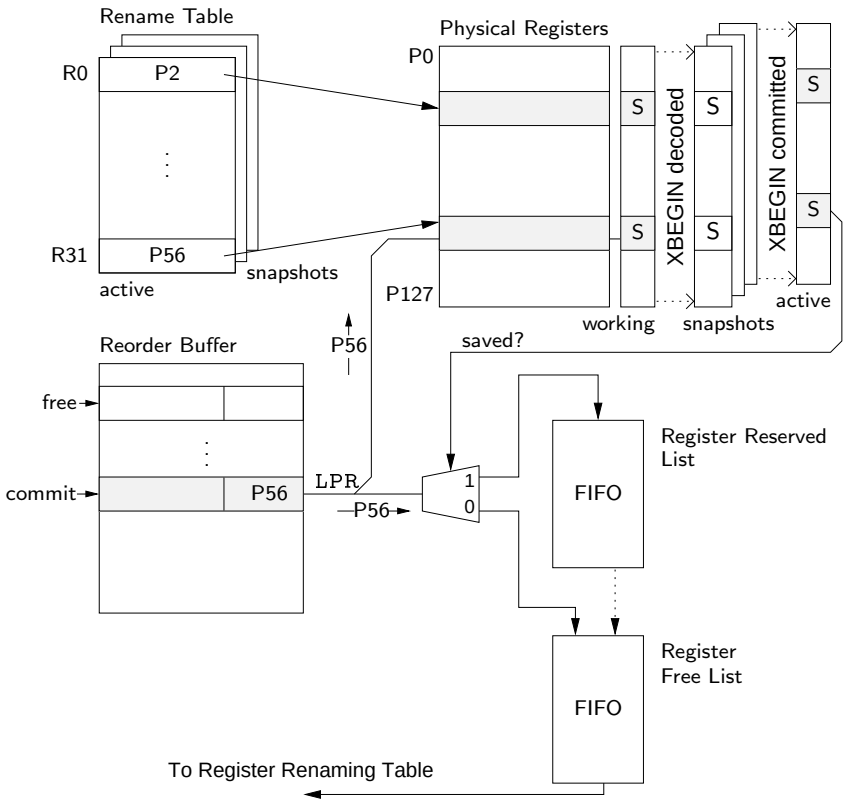



Figure 6.1: UTM processor modifications. The S bit vector tracks the active physical registers. For each rename table snapshot, there is an associated S bit vector snapshot. The Register Reserved List holds the otherwise free physical registers until the transaction commits. The LPR field is the next physical register to free (the last physical register referenced by the destination architectural register).

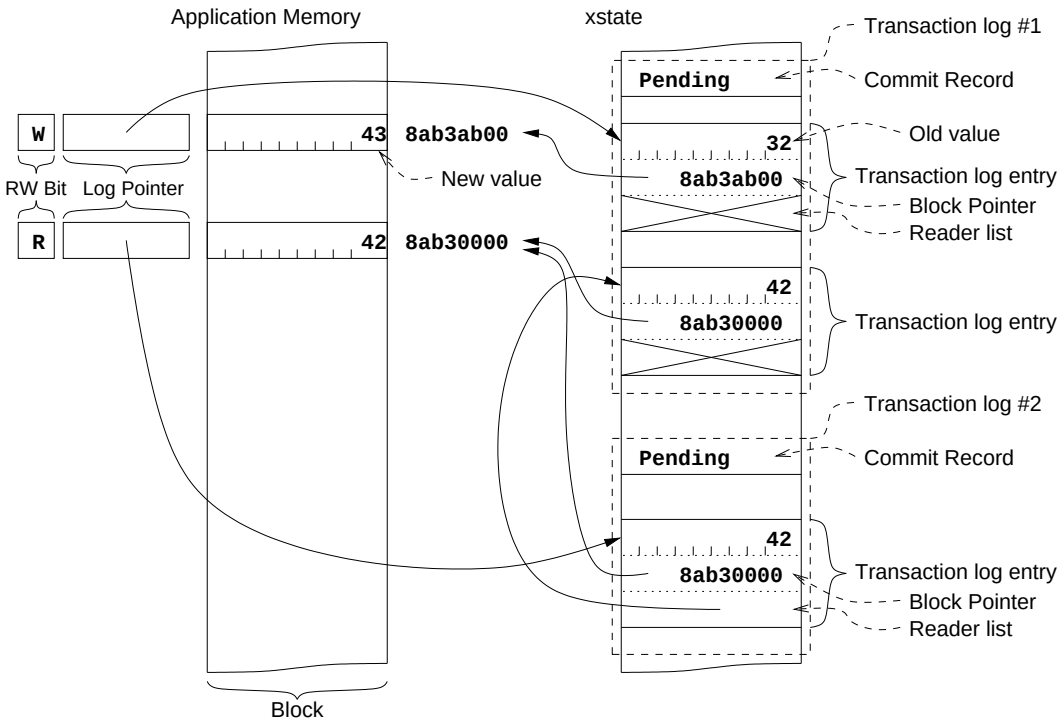



Figure 6.2: The xstate data structure. The transaction log for a transaction contains a commit record and a vector of log entries. The log pointer of a block in memory points to a log entry, which contains the old value of the block and a pointer to the transaction's commit record. Two transaction logs are shown here; generally, the xstate includes the active transaction logs for the entire system.

block more than once in a transaction, the system can avoid writing multiple entries into the transaction log by checking the log pointer to see whether a log entry for the block already exists as part of the running transaction.

By following the log pointer to the log entry, then following the log entry pointer to the commit record, one can determine the transaction status (pending, committed, or aborted) of each block. To commit a transaction, the system simply changes the commit record from `PENDING` to `COMMITTED`. At this point, a reference to the block produces the new value stored in memory, albeit after some delay in chasing pointers to discover that the transaction has been committed. To avoid this delay, as well as to free the transaction log for reuse, the system must clean up after committing. It does so by iterating through the log entries, clearing the log pointer for each block mentioned, thereby finalizing the contents of the block. Future references to that block will continue to produce the new value stored in memory, but without the delay of chasing pointers. To abort a transaction, the system changes the commit record from `PENDING` to `ABORTED`. To clean up, it iterates through the entries, storing the old value back to memory and then clearing the log pointer. We chose to store the old value of a block in the transaction log and the new value in memory, rather than the reverse, to optimize the case when a transaction commits. No data copying is needed to clean up after a commit, only after an abort.

When two or more pending transactions have accessed a block and at least one of the accesses is a store, the transactions conflict. Conflicts are detected during operations on memory. When a transaction performs a load, the system checks that either the log pointer refers to an entry in the current transaction log, or else that the RW bit is `R` (additionally creating an entry in the current log for the block if needed). When a transaction performs a store, the system checks that no other transaction is referenced by the log pointer (i.e., that the log pointer is cleared or that the linked list of log entries corresponding to this block are all contained in the current transaction log). If the conflict check fails, then some of the conflicting

transactions are aborted. To guarantee forward progress, UTM writes a timestamp into the transaction log the first time a transaction is attempted. Then, when choosing which transactions to abort, older transactions take priority. As an alternative, a backoff scheme [53] could also be used.

When a writing transaction wins a conflict, there may be multiple reading transactions that must be aborted. These transactions are found efficiently by following the block's log pointer to an entry and traversing the linked list found there, which enumerates all entries for that block in all transaction logs.

### 6.1.4 Caching

Although UTM can support transactions of unbounded size using the xstate data structure, multiple memory accesses for each operation may be required. Caching is needed to achieve acceptable performance. In the common case of a transaction that fits in cache, UTM, like the earlier proposed HTM systems [46, 36], monitors the cache-coherence traffic for the transaction's cache lines to determine if another processor is performing a conflicting operation. For example, when a transaction writes to a memory location, the cache protocol obtains exclusive ownership on the whole cache block. New values can be stored in cache with old values left in memory. As long as nothing revokes the ownership of any block, the transaction can succeed. Since the contents of the transaction log are undefined after the transaction commits or aborts, in many cases the system does not even need to write back a transaction log. Thus, for a small transaction that commits without intervention from another transaction, no additional interprocessor communication is required beyond the coherence traffic for the nontransactional case. When the transaction is too big to fit in cache or interactions with other transactions are indicated by the cache protocol, the xstate for the transaction overflows into the ordinary memory hierarchy. Thus, the UTM system does not actually need to create a log entry or update the log pointer for a cached block unless it is evicted. After a transaction commits

or aborts, the log entries of unspilled cached blocks can be discarded and the log pointer of each such block can be marked clean to avoid writeback traffic for the log pointer, which is no longer needed. Most of the overhead is borne in the uncommon case, allowing the common case to run fast.

The in-cache representation of transactional state and the xstate data structure stored in memory need not match. The system can optimize the on-processor representation as long as, at the cache interface, the view of the xstate is properly maintained. For convenience, the transaction block size can match the cache line size.

### 6.1.5 System issues

The goal of UTM is to support transactions that can run for an indefinite length of time (surviving time slice interrupts), can migrate from one processor to another along with the rest of a process's state, and can have footprints bigger than the physical memory. Several system issues must be solved for UTM to achieve that goal. The main technique that we propose is to treat the xstate as a system-wide data structure that uses global virtual addresses.

Treating the xstate as data structure directly solves part of the problem. For a transaction to run for an indefinite length of time, it must be able to survive a time-slice interrupt. By adding the log pointer to the processor state and storing everything else in a data structure, it is easy to suspend a transaction and run another thread with its own transaction. Similarly, transactions can be migrated from one processor to another. The log pointer is simply part of the thread or process state provided by the operating system.

UTM can support transactions that are even larger than physical memory. The only limitation is how much virtual memory is available to store both old and new values. To page the xstate out of main memory, the UTM data structures might employ global virtual addresses for their pointers. Global virtual addresses are system-wide unique addresses that remain

valid even if the referenced pages are paged out to disk and reloaded in another location. Typically, systems that provide global virtual addresses provide an additional level of address translation, compared to ordinary virtual memory systems. Hardware first translates a process's virtual address into a global virtual address. The global virtual address is then translated into a physical address. Multics [11] provided user-level global virtual addressing using segment-offset pairs as the addresses. The HP Precision Architecture [49] supports global virtual addresses in a 64-bit RISC processor.

The log pointer and state bits for each user memory block, while typically not visible to a user-level programmer, are themselves stored in addressable physical memory to allow the operating system to page this information to disk. The location of the memory holding the log pointer information for a given user data page is kept in the page table and cached in the TLB.

During execution of a single load or store instruction, the processor can potentially touch a large number of disparate memory locations in the xstate, any of which may be paged out to disk. To ensure forward progress, either the system must allow load or store instructions to be restarted in the middle of the xstate traversal, or, if only precise interrupts are allowed, the operating system must ensure that all pages required by an xstate traversal can be resident simultaneously to allow the load or store to complete without page faults.

UTM assumes that each transaction is a serial instruction stream beginning with an `XBEGIN` instruction, ending with a `XEND` instruction, and containing only register, memory, and branch instructions in between. A fault occurs if an I/O instruction is executed during a transaction.

## 6.2 The LTM architecture

UTM is an idealized design for HTM that requires significant changes to both the processor and the memory subsystem of a current computer architecture. By scaling back on the degree of "unboundedness," however, a com-

promise between programmability and practicality can be achieved. This section presents such an architectural compromise, called LTM, for which we have implemented a detailed cycle-level simulation using UVSIM [71]. The limited transactions supported by LTM are still powerful enough to serve as the basic for an hybrid system, as we will show in chapter 6.4.

LTM's design is easier to implement than UTM, because it does not support transactions of virtual-memory size. Instead, LTM avoids the intricacies of virtual memory by limiting the footprint of a transaction to (nearly) the size of physical memory. In addition, the duration of a transaction must be less than a time slice and transactions cannot migrate between processors. With these restrictions, LTM can be implemented by only modifying the cache and processor core and without making changes to the main memory, the cache-coherence protocols, or even the contents of the cache-coherence messages. Unlike a UTM processor, an LTM processor can be pin-compatible with a conventional processor. The design presented here is based on the SGI Origin 3000 shared-memory multiprocessor, with memory distributed among the processor nodes and cache coherency maintained using a directory-based write-invalidate protocol.

The UTM and LTM schemes share many ideas. Like UTM, LTM maintains data about pending transactions in the cache and detects conflicts using the cache-coherency protocol in much the same way as previous HTM proposals [46, 40]. LTM also employs an architectural state-save mechanism in hardware. Unlike UTM, LTM does not treat the transaction as a data structure. Instead, it binds a transaction to a particular cache. Transactional data overflows from the cache into a hash table in main memory, which allows LTM to handle transactions too big to fit in the cache without the full implementation complexity of the xstate data structure.

LTM has similar semantics to UTM, and the format and behavior of the `XBEGIN` and `XEND` instructions are the same. The information that UTM keeps in the transaction log is kept partly in the processor, partly in the cache, and partly in an area of physical memory allocated by the operating

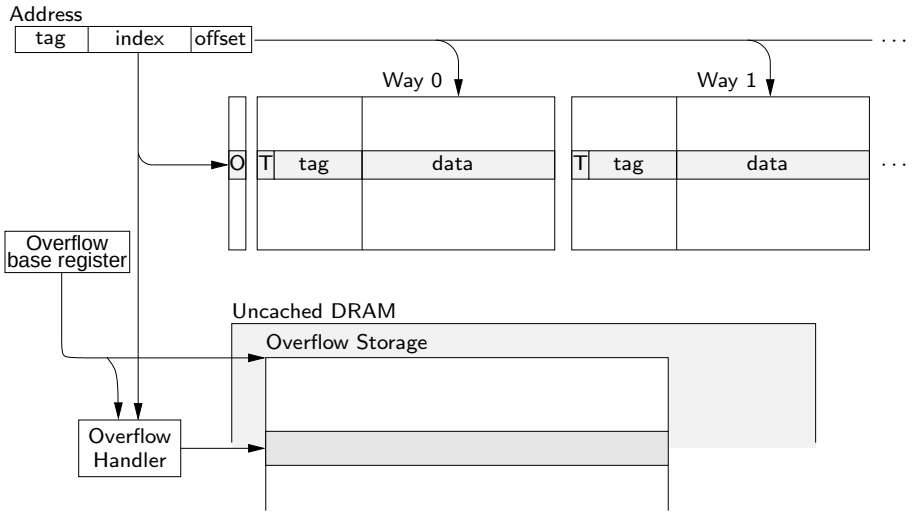



Figure 6.3: LTM cache modifications. The T bit indicates if the line is transactional. The O bit indicates if the set has overflowed. Overflowed data is stored in a data structure in uncached DRAM.

system.

LTM requires only a few small modifications to the cache, as shown in Figure 6.3. For small transactions, the cache is used to store the speculative transactional state. For large transactions, transactional state is spilled into an overflow data structure in main memory. An additional bit (T) is added per cache line to indicate if the data has been accessed as part of a pending transaction. When a transactional-memory request hits a cache line, the T bit is set. An additional bit (O) is added per cache set to indicate if it has overflowed. When a transactional cache line is evicted from the cache for capacity reasons, the O bit is set.

In LTM, the main memory always contains the original state of any data being modified transactionally, and all speculative transactional state is stored in the cache and overflow hash table. A transaction is committed by simply clearing all the T bits in cache and writing all overflowed data back to memory. Conflicts are detected using the cache-coherency protocol. When

an incoming cache intervention hits a transactional cache line, the running transaction is aborted by simply clearing all the T bits and invalidating all modified transactional cache lines.

The overflow hash table in uncached main memory is maintained by hardware, but its location and size are set up by the operating system. If a request from the processor or a cache intervention misses on the resident tags of an overflowed set, the overflow hash table is searched for the requested line. If the requested cache line is found, it is swapped with a line in the cache set and handled like a hit. If the line is not found, it is handled like a miss. While handling overflows, all incoming cache interventions are stalled using a NACK-based network protocol.

The LTM overflow data structure uses the low-order bits of the address as the hash index and uses linear probing to resolve conflicts. When the overflow data structure is full, the hardware signals an exception so that the operating system can increase the size of the hash table and retry the transaction.

LTM was designed to be a first step towards a truly unbounded transactional memory system such as UTM. LTM has most of the advantages of UTM while being much easier to implement. As I will show at the end of the chapter, LTM's more practical implementation of quasi-unbounded transactional memory suffices for many real-world concerns, and can be symbiotically paired with our more flexible software transaction system to achieve truely unbounded transactions at minimal hardware cost.

## 6.3 Evaluation

In this section we will evaluate the UTM and LTM designs, demonstrating low overhead and scalability. We will also consider overflow behavior, providing some motivation for the hybrid system proposed in the next section.

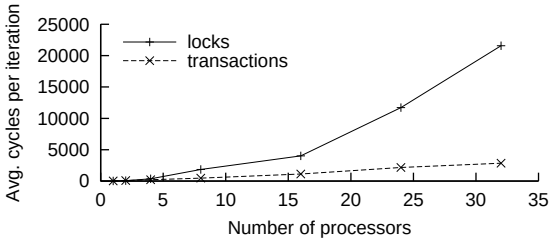



Figure 6.4: `Counter` performance on UVSIM.

### 6.3.1 Scalability

We used a parallel microbenchmark to examine program behavior for small transactions with high contention. Our results show that the extremely low overhead of small hardware transactions enable them to almost always complete even when contention is high.

The `Counter` microbenchmark has one shared variable that each processor atomically increments repeatedly with no backoff policy; the basic idea is identical to the microbenchmark we used in chapter 3.4. Each transaction is only a few instructions long and every processor attempts to write to the same location repeatedly. Both a locking and a transactional version of `Counter` were run on UVSIM with LTM, and the results are shown in Figure 6.4. In the locking version, there is a global spin-lock that each processor obtains using a load-linked/store-conditional (LLSC) sequence from the SGI synchronization libraries.

The locking version scales poorly, because the LLSC 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 is likely to be able to execute a few more instructions before receiving an intervention since the network latency is high. Therefore, small transactions can start and complete (and perhaps even start and complete the next transaction) before the cache line is taken away. Similar behavior is expected from UTM, and other

transactional-memory systems which use the cache and cache-coherency protocol to store transactional state, since small transactions effectively use the cache the same way.

### 6.3.2 Overhead

A main goal of LTM and UTM is to run the common case fast. As shown in chapter 3.1, the common case is when transactions are small and fit in the cache. Therefore, by using the cache and cache coherency mechanism to handle small transactions, LTM is able to execute with almost no overhead over serial code in the common case. In this section, we discuss qualitatively how the LTM implementation is optimized for the common case and how similar techniques are used in UTM. The discussion is broken into the following three cases: starting, running, committing a transaction.

Starting a transaction in LTM requires virtually no overhead in the common case since the hardware only needs to record the abort handler address. No communication with the cache or other external hardware is necessary. There is the added overhead of decoding the `XBEGIN` however that overhead is generally insignificant compared to the cost of the transaction. Further, instruction decode overhead is much lower in LTM than with locks. Even schemes where the lock is not actually held such as SLE [57] have higher decode overhead since they have more instructions. LTM's low transaction startup overhead is a very good indicator of the corresponding overhead in UTM since transaction start up in UTM is virtually the same.

Running a transaction in LTM requires no more overhead than running the corresponding non-synchronized code in the common case. In LTM, the T bit is simply set on each transactional cache access. LTM's low overhead in this case unfortunately does not translate directly to UTM since UTM modifies the transaction record on each memory request. However, in the common case the transaction record entry is also in the cache. Thus all operations are local and no external communication is needed. Also, in some cases, the cache can respond to the memory request once the requested data

is found. However, if the request requires data from the transaction record before it can be serviced, an additional cache lookup is necessary. However, the lookup is local and thus can be done relatively quickly. Therefore, the common case overhead of running a transaction can be minimal even in UTM.

Committing a transaction in LTM has virtually no overhead in the common case since it can be done in one clock cycle. LTM transaction commits only requires a simply flash clear of all the transaction bits in the cache. Similarly, UTM transaction commits only require a single change of the cached transaction record to "committed". UTM transaction commit also writes the updated values from the transaction record back to memory. However, this write back can be done lazily in the background. Therefore, since transaction commit requires only a single change in the cache for both LTM and UTM, the overhead is minimal in both cases.

### 6.3.3 Overflows

Overflows occur only in the uncommon case however our studies show that it is important to have a scalable data structure even though it is used infrequently.

For evaluation, we compiled three versions of the SPECjvm98 benchmark suite to run under UVSIM using FLEX. We compiled a ***Base*** version that uses no synchronization, a ***Locks*** version that uses spin-locks for synchronization, and a ***Trans*** version that uses LTM transactions for synchronization. To measure overheads, we ran these versions of the SPECjvm98 benchmark suite on one processor of UVSIM.

As described in chapter 4.2, our transactional version uses method cloning to flatten transactions; we performed the same cloning on the other compiled versions so that performance improvements due to the specialization would not be improperly attributed to transactionification. The three different benchmark versions were built from a common code-base using method

inlining in gcc[2] to remove or replace all invocations of lock and transaction entry and exit code with appropriate implementations. No garbage collection was performed during these benchmark runs.

Our initial results from chapter 3.1 suggested that overflows ought to be infrequent, thus the efficiency of the overflow data structure would have a negligible effect on overall performance. Therefore, our first LTM implementation used an unsorted array that required a linear search on each miss to an overflowed set. The unsorted array was effective for most of our test cases, as they had less overhead than locks. Using LTM with the unsorted array, however, the transactional version of 213_javac was 14 times slower than the base version. Virtually all of the overhead came from handling overflows, which is not surprising, since the entire application is enclosed in one large transaction. The large transaction touches 13K cache lines with 9K lines overflowed. So, even though only 0.5% of the transactional memory operations miss in the cache, each one incurs a huge search cost. This unexpected slowdown indicated that a naive unsorted array is insufficient as an overflow data structure. Therefore, LTM was redesigned to use a hash table to store overflows.

Since the entire application was enclosed in a transaction, the 213_javac application was clearly not written to be a parallel application. However, it is important that an unbounded transactional memory system be able to support even such applications with reasonable performance. Therefore, we redesigned LTM to use hash table as described in chapter 6.2.

Using LTM with the hash table, the SPECjvm98 application overheads were much more reasonable as shown in Figure 6.5. The hash table data structure decreased the overhead from a 14x slowdown to under 15% in 213_javac. Using the hash table, LTM transactional overhead is less than locking overhead in all cases.

[2]We compiled the files generated by FLEX's "Precise C" backend (chapter 4.1) with -O9 for a -mips4 target using the n64 API to generate fully-static binaries executable by UVSIM.

| Benchmark application | Base time (cycles) | Locks time (% of Base time) | Trans time (% of Base time) | Time in trans (% of Trans time) | Time in overflow (% 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% |
| 222_mpegaudio | 99.0M | 100.3% | 99.6% | 0.8% | 0% |
| 228_jack | 261.4M | 175.3% | 104.3% | 32.1% | 0.0056% |

Figure 6.5: SPECjvm98 performance on a 1-processor UVSIM simulation. The *Time in trans* and *Time in overflow* are the times spent actually running a transaction and handling overflows respectively. The input size is 1. The overflow hash table is 128MB.

## 6.4 A hybrid transaction implementation

We've seen that UTM and LTM can operate with very little overhead, but hardware schemes encounter difficulties when scaling too very large or long-lived transactions. We have overcome some of the difficulties with an overflow cache (LTM), or by virtualizing transactions and dumping their state to a data structure (UTM). However, it is worth considering whether this extra complexity is worthwhile: why not combine the strengths of our object-based software transaction system (explicit transaction state, unlimited transaction size, flexibility) with the fast small transactions at which a hardware system naturally excels?

Figure 6.6 presents the results of such a combination. In the figure, combining the systems is done in the most simple-minded way: all transactions are begun in LTM, and after any abort the transaction is restarted in the object-based software system. The field flag mechanism described in section 3.2.5 ensures that software transactions properly abort conflicting hardware transactions: when the software scribbles `FLAG` over the original field the hardware will detect the conflict. Hardware transactions must per-

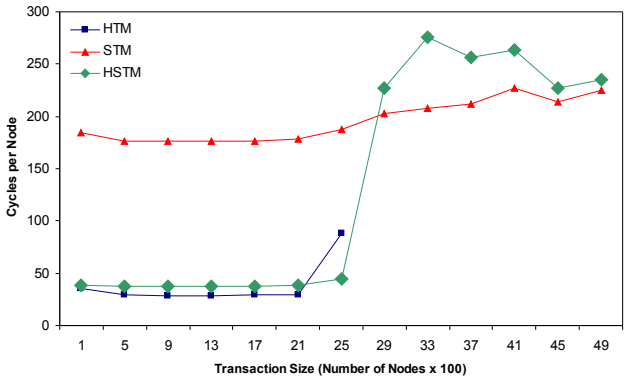



Figure 6.6: Performance (in cycles per node push on a simple queue benchmark) of LTM [5] (HTM), the object-based system presented in this paper (STM) and a hybrid scheme (HSTM).

form the `ReadNT` and `WriteNT` algorithms to ensure they interact properly with concurrent software transactions, although these checks need not be part of the hardware transaction mechanism. In the figure, the checks were done in software, with an implementation similar to that described in chapter 3.4.

The figure shows the performance of a simple queue benchmark as the transaction size increases. The hardware transaction mechanism is fastest, as one would expect, but it performance falters and then fails at a transaction size of around 2500 nodes pushed. At this transaction size the hardware scheme ran out of cache; in a more-realistic system it might also have run out of its timeslice, aborting (LTM) or spilling (UTM) the transaction at the context switch.

Above HTM in the figure is the performance of the software transaction system, which is about 4x slower. This is a pessimistic figure: no special effort was made to tune code or otherwise minimize slowdown, and the processor simulated had limited ability to exploit ILP (2 ALUs and 4-instruction issue width). However, the software scheme is unaffected by

increasing transaction size.

The hybrid scheme successfully combines the best features of both. It is only about 20% slower than the basic hardware scheme, due to the read and write barriers it much implement, but at the point where the hardware stops working well, it smoothly crosses over to track the performance of the software transaction system.

There are many fortuitous synergies in such an approach. Hardware support for small transactions may be used to implement the software transaction implementation's Load Linked/Store Conditional sequences, which may not otherwise be available on a target processor. The small transaction support can also facilitate a functional array solution to the large object problem, as we saw in chapter 5.4. We might further improve performance by adding a bit of hardware support for the `readNT`/`writeNT` barriers [16].

I believe this hybrid approach is the most promising direction for transaction implementations in the near future. It preserves the flexibility to investigate novel solutions to the outstanding challenges of transactional models, which we will review in the next chapter, and solves an important chicken-and-egg problem preventing the development of transactional software and the deployment of transactional hardware. Since the speed of the hardware mechanism is tempered by the cooperation protocol of the software transaction system, high-efficiency software transaction mechanisms, such as the one presented in this thesis, are the key enabler for hybrid systems.

The thing is to remember that the future comes one day at a time.

*Dean Acheson*

# Chapter 7

# Challenges

In this section we will review objections that have been raised to straightforward or naïve transaction implementations. Some of these objections do not apply to our implementation; discussion of these may further illuminate our design choices. Others apply to certain situations, and should be kept in mind when creating applications. Some of the problems raised remain unsolved, and are the subject of future work; for these we will attempt to sketch research directions.

## 7.1 Performance isolation

Zilles and Flint [73] identified ***performance isolation*** as a potential issue for transaction implementations. In a system with performance isolation, the execution of one task (process, thread, transaction) should complete in an amount of time which is independent of the execution of other tasks (processes, threads, transactions) in the system. For a system with N processors, it is ideal if a task is guaranteed to complete at least as quickly as it would running alone on 1 processor.

It is obvious that most common systems do not provide any guarantee of performance isolation: on a typical multi-user system, the execution of a given task can be made arbitrarily slow by the concurrent execution of

competing tasks.[1] However, a nontransactional system can be constructed with a good deal of performance isolation by appropriately restricting the processes that can be run and the resources they consume.

Zilles and Flint object that many transactional systems are constructed such that a single large transaction may monopolize the atomicity resources such that no other transactions may commit. By opening a transaction, touching a large number of memory locations, and then never closing the transaction, a malicious application may deny service to all concurrent applications in a transaction system.

For this reason, it is important that there are no global resources required to complete a transaction. Our UTM hardware implementation achieves this end, but the LTM design uses a per-processor overflow table. If an LTM design is implemented with a snoopy bus for coherence traffic, overflows on one processor can impact the performance of all other processors on the bus. A directory-based coherence protocol (as we have described in this thesis) eliminates this problem. Hybrid schemes based on LTM also eliminate the problem, because an overflowing transaction can be aborted and retried in software, which requires no global resources.

Concerns about performance isolation are not limited to transaction systems. Transaction systems provide a solution not available to systems with lock-based concurrency, however: the offending transaction can be safely aborted at any point to allow the other transactions to progress.

## 7.2 Progress guarantees

Aborting troublesome transactions raises another potential pitfall: how do we guarantee that our system will make forward progress? Zilles and Flint [73] note that transaction systems are subject to an "all-or-nothing" problem: it's fine to abort a troublesome large transaction to allow other work to complete, but then we throw away any progress in that transaction.

[1] Grunwald and Ghiasi [28] call this a "microarchitectural denial of service" attack.

The operating system is forced to either allocate a large transaction all the resources it requires, or to refuse to make any progress on the transaction; there is no middle ground.

This criticism applies to the LTM hardware scheme and other unvirtualized transaction implementation. In an LTM system, it is the programmer's responsibility to structure transactions such that the application is likely to complete. The operating system will deny progress when necessary to prevent priority inversion.

The UTM, hybrid, and software-only implementations do not suffer the same problem. UTM and software-only implementations can virtualize the transaction, as all transaction state is in memory. This allows the resources required to be paged in incrementally as needed. The hybrid scheme, even when built on an unvirtualized mechanism such as LTM, can abort and fail-over to a virtualized software system if sufficient resources are not available.

## 7.3 The semantic gap

In a vein of optimism, transactions are often casually said to be "compatible" with locks: transform your lock acquisition and release statements to begin-transaction and end-transaction, respectively, and your application is transformed. We might even claim that your application will suddenly be faster/more parallel and that some lingering locking bugs and race conditions will be cured by the change as well.

It is the latter part of the claim that draws first scrutiny: if you've "fixed" my race conditions, haven't you altered the semantics of my program? What other behaviors might have changed?

Blundell, Lewis, and Martin [13] describe the "semantic gap" opened between the locking and naïvely-transactified code. They point out that programs with data races – even "benign" ones – may deadlock when converted to use transactions, since the data race will never occur. One may even wrap every access with a location-specific lock to "remove" the race

(for some definitions) without altering the behavior of the locking code or the deadlock for the transactional version.

This concern is valid, and claims of automatic transactification should not be taken too lightly. However, most "best-practices" concurrent code *will* behave as expected, and (unlike timing-dependent code with races) deadlocks make it very obvious where things go wrong. Further, type systems are capable of detecting the deadlocks in transactified code and alerting the programmer of the problem.

Blundell, Lewis, and Martin also point out that some transaction implementations ignore "non-transactional" accesses – even if these are to locations which are currently involved in a transaction. This leads to additional alterations in the semantics of the code. In the implementations described in this thesis, we are careful to ensure that "non-transactional" code still executes as if each statement was its own individual transaction, what Blundell, Lewis, and Martin term ***strong atomicity***.

## 7.4 I/O Mechanisms

To be useful, computing systems must be connected to the outside world in some fashion. This creates a discontinuity in the transactional model: real world events can not be rolled back in the same way as can changes to program state.

### 7.4.1 Forbidding I/O

The most straight-forward means to accommodate I/O in the transactional framework is to forbid it: I/O may only happen outside of a transaction. A runtime check or simple type system may be used to enforce this restriction.

A useful programmer technique in this model is to create a separate concurrent thread for I/O. A transaction can interact with a queue to request I/O, and the I/O thread will dequeue requests and enqueue responses. This works best for unidirectional communication. Note that round-trip commu-

nication with the I/O thread can not be accomplished within a single transaction (deadlock will result), so transactions still must be broken between a request and reply: some forms of interaction can not be accomplished atomically.

### 7.4.2 Mutual exclusion

Another alternative is to integrate mutual exclusion into the transaction model: once we start an I/O activity within a transaction, the transaction becomes ***uninterruptible***: it may no longer be aborted, and must be successfully executed through commit. Only a single uninterruptible transaction may execute at a given time (although other interruptible transactions may be concurrent). Effectively there is a single global mutual exclusion lock for I/O activity; transactions attempting I/O while the lock is already held are either delayed or aborted.

This scheme is reasonable as long as I/O is infrequent in transactions, and is convenient for the programmer. A single debugging print-statement, however, is sufficient to serialize a transaction, and efforts to make the single global I/O lock more fine-grained may ultimately give back the gains in simplicity and concurrency afforded by transactions in the first place.

### 7.4.3 Postponing I/O

A hybrid approach attempts to anticipate or postpone I/O operations so that they run only at transaction start or end. Only the I/O operations then need to be run serially; the remainder of the transaction may still execute concurrently. Input must be moved to the start of a transaction, and once the input has been consumed, that transaction must still run uninterruptibly; it may be aborted only if a push-back buffer can be constructed for the input, which is not always reasonable. Output is moved to the end of the transaction, but only the actual I/O must be performed uninterruptibly: the transaction can still be aborted at any time prior to commit.

If output and input need to be interleaved, or the input occurs after an output and thus can not be moved to the start of the transaction, uninterruptible transactions are still required. This approach thus works around the disadvantages of mutual exclusion in some cases, but a single misplaced debugging statement can still force serialization.

It is worth noting that modern interface hardware is often designed such that it works well with this approach. For example, a GPU or network card will take commands from or deliver input to a buffer; a single operation is sufficient to hand over a buffer to the card to commence I/O. This single I/O action may be made atomic with the transaction commit.

### 7.4.4 Integrating do/undo

The most sophisticated integration of I/O with transactions allows the programmer to specify "undo" code for I/O which can not otherwise be rolled back. In the database community, these are referred to as ***compensating transactions***.

Again, I/O would be forbidden within "pure" transactions; however, do/undo blocks may be nested within transactions. A do block executes uninterruptibly. If a transaction aborts before it reaches a do block, rollback occurs conventionally. If it aborts after it has executed a do block, then the undo block is executed uninterruptibly as part of transactional rollback. Note that mutual exclusion must still be used in portions of the transaction processing; ideally the critical regions are short and infrequently invoked.

The do/undo behavior allows sophisticated exception processing: an undo block may emit a backspace to the terminal after do emits a character, or it may send an apology email after an email is sent irrevocably in a do block. The do/undo is invisible to clients outside the transaction. Sophisticated libraries can be built up using this mechanism. For example, disk i/o may be made transactional using file versioning and journalling.

The undo blocks may be difficult to program. The most straightforward implementation would prevent the undo from accessing any transactional

state, and the programmer must take special care if she is to maintain a consistent view of program state. A friendlier implementation would present a view of program state such that it appears that the undo block executes immediately after the do block, in the same lexical environment (regardless of what ultimately-aborted transactional code has executed in the interim). The programmer is then able to naturally write code such as the following:

```
String from = ..., to = ..., subject = ...;
do {
  sendEmail(from, to, subject);
} undo {
  sendApology(from, to, "Re: "+subject);
}
/* code here can modify from, to, subject */
/* before transaction commit */
```

Presenting a time-warped view to the undo block can be difficult or impossible, depending on the history mechanism involved. In particular, if the undo reads a new location, previously untouched by the transaction, the value of this location at the (previous) time immediately following the do might be unavailable. Presenting an inconsistent view of memory may be undesirable.

## 7.5 OS interactions

While transaction-style synchronization has been successfully used to structure interactions within an operating system [52], transactions crossing the boundary between operating system and application present additional challenges. Some operating system requests are either I/O operations or can be handled with the same mechanism used to handle I/O within transactions, as discussed in the previous section. Using memory allocation as an example of an OS request, transactions can be forbidden to allocate memory, required to take a lock, forced to preallocate all required memory,[2] or use a compensation mechanism to deallocate requested memory in case of abort.

[2] A retry mechanism can be used to incrementally increase the preallocation until the transaction can successfully complete.

Since the role of an operating system is to administer shared resources, special care must be taken that transactions involving operating system requests do not "contaminate" other processes. In particular, if data in kernel space is to be included in a transaction:

- If mutual exclusion may used to implement any part of the transaction semantics (as in the various I/O schemes above), then it may be possible to tie up the entire system (including unrelated processes) until a transaction touching kernel structures commits.

- If transaction state if to be tracked in the cache, the kernel address space must be reserved from the application memory map on architectures with virtually-addressed caches.

- It may be desirable to include some loophole mechanism so that kernel data structures may be released from the transaction. Similarly, the operating system may wish to use transactions within itself; it may be desirable for these transactions to be independent from the application's invoking transaction. Motivating examples include fault handlers and the paging mechanism, which ought to be transparent to the application's transaction.

It is possible to handle these challenges simply, for example by forbidding OS calls within a transaction and aborting transactions if necessary on context switches or faults. Such an approach raises hurdles for the application programmer but simplifies the operating system's task considerably. In unvirtualized transaction implementations such as LTM, this approach also limits the maximum duration of a transaction to a single time slice, although the OS may be able to stretch a processes time slice when necessary.

A more sophisticated approach with explicit OS management of transactions may be able to provide better transparency of OS/transaction interactions for the application programmer and improved performance.

## 7.6 Recommendations for future work

Based on the discussion in this section, a virtualizable transaction mechanism is recommended: if LTM is implemented in hardware, a hybrid scheme should support it in order to provide performance isolation and progress guarantees. A do/undo mechanism for transactions allows better management of critical regions where mutual exclusion must be integrated with the transaction mechanism to support I/O and certain OS interactions. All OS interactions can then be performed in do block so that mutual exclusion need not be extended across the OS boundary.

Everything in the universe relates to [transactions], one way or another, given enough ingenuity on the part of the interpreter.

*Principia Discordia* (amended)

# Chapter 8

# Related work

A number of researchers have been investigating transactional memory systems. This thesis is the first to present a hybrid hardware/software model-checked non-blocking object-oriented system which allows co-existence of non-transactional and transactional accesses to a dynamic set of object fields.

## 8.1 Non-blocking synchronization

Lamport presented the first alternative to synchronization via mutual exclusion in [48], for a limited situation involving a single writer and multiple readers. Lamport's technique relies on reading guard elements in an order opposite to that in which they are written, guaranteeing that a consistent data snapshot can be recognized. The writer always completes its part of the algorithm in a constant number of steps; readers are guaranteed to complete only in the absence of concurrent writes.

Herlihy formalized *wait-free* implementations of concurrent data objects in [37]. A wait-free implementation guarantees that any process can complete any operation in a finite number of steps, regardless of the activities of other processes. Lamport's algorithm is not wait-free because readers can be delayed indefinitely.

Massalin and Pu introduced the term *lock-free* to describe algorithms with weaker progress guarantees. A lock-free implementation guarantees only that *some* process will complete in a finite number of steps [52]. Unlike a wait-free implementation, lock-freedom allows starvation. Since other simple techniques can be layered to prevent starvation (for example, exponential backoff), simple lock-free implementations are usually seen as worthwhile practical alternatives to more complex wait-free implementations.

An even weaker criterion, *obstruction-freedom*, was introduced by Herlihy, Luchangco, and Moir in [39]. Obstruction-freedom only guarantees progress for threads executing in isolation; that is, although other threads may have partially completed operations, no other thread may take a step until the isolated thread completes. Obstruction-freedom not only allows starvation of a particular thread, it allows contention among threads to halt all progress in all threads indefinitely. External mechanisms are used to reduce contention (thus, achieve progress) including backoff, queueing, or timestamping.

We will use the term *non-blocking* to describe generally any synchronization mechanism which doesn't rely on mutual exclusion or locking, including wait-free, lock-free, and obstruction-free implementations. We will be concerned mainly with lock-free algorithms.[1]

## 8.2 Efficiency

Herlihy presented the first *universal* method for wait-free concurrent implementation of an arbitrary sequential object [37, 33]. This original method was based on a *fetch-and-cons* primitive, which atomically places an item

[1] Note that some authors use "non-blocking" and "lock-free" as synonyms, usually meaning what we here call *lock-free*. Others exchange our definitions for "lock-free" and "non-blocking", using lock-free as a generic term and non-blocking to describe a specific class of implementations. As there is variation in the field, we choose to use the parallel construction *wait-free*, *lock-free*, and *obstruction-free* for our three specific progress criteria, and the dissimilar *non-blocking* for the general class.

on the head of a list and returns the list of items following it; all concurrent primitives capable of solving the $n$-process consensus problem—*universal* primitives—were shown powerful enough to implement *fetch-and-cons*. In Herlihy's method, every sequential operation is translated into two steps. In the first, *fetch-and-cons* is used to place the name and arguments of the operation to be performed at the head of a list, returning the other operations on the list. Since the state of a deterministic object is completely determined by the history of operations performed on it, applying the operations returned in order from last to first is sufficient to locally reconstruct the object state prior to our operation. We then use the prior state to compute the result of our operation without requiring further synchronization with the other processes.

This first universal method was not very practical, a shortcoming which Herlihy soon addressed [34]. In addition, his revised universal method can be made lock-free, rather than wait-free, resulting in improved performance. In the lock-free version of this method, objects contain a shared variable holding a pointer to their current state. Processes begin by loading the current state pointer and then copying the referenced state to a local copy. The sequential operation is performed on the copy, and then if the object's shared state pointer is unchanged from its initial load it is atomically swung to point at the updated state.

Herlihy called this the "small object protocol" because the object copying overhead is prohibitive unless the object is small enough to be copied efficiently (in, say, $O(1)$ time). He also presented a "large object protocol" which requires the programmer to manually break the object into small blocks, after which the small object protocol can be employed. [This trouble with large objects is common to many non-blocking implementations; our solution is presented in chapter 5.]

Barnes provided the first universal non-blocking implementation method which avoids object copying [10]. He eliminates the need to store "old" object state in case of operation failure by having all threads cooperate to apply

operations. For example, if the first processor begins an operation and then halts, another processor will complete the operation of the first before applying its own. Barnes proposes to accomplish the cooperation by creating a parallel state machine for each operation, so that each thread can independently try to advance the machine from state to state and thus advance incomplete operations.[2] Although this avoids copying state, the lock-step cooperative process is extremely cumbersome and does not appear to have ever been implemented. Furthermore, it does not protect against errors in the implementation of the operations, which could cause *every* thread to fail in turn as one by one they attempt to execute a buggy operation.

Alemany and Felten [2] identified two factors hindering the performance of non-blocking algorithms to date: resources wasted by operations that fail, and the cost of data copying. Unfortunately, they proceeded to "solve" these problems by ignoring short delays and failures and using operating system support to handle delays caused by context switches, page faults, and I/O operations. This works in some situations, but obviously suffers from a bootstrapping problem as the means to implement an operating system.

Although lock-free implementations are usually assumed to be more efficient that wait-free implementations, LaMarca [47] showed experimental evidence that Herlihy's simple wait-free protocol scales very well on parallel machines. When more than about twenty threads are involved, the wait-free protocol becomes faster than Herlihy's lock-free small-object protocol, three OS-aided protocols of LaMarca and Alemany and Felten, and a *test-and-Compare&Swap* spin-lock.

[2] It is interesting to note that Barnes' cooperative method for non-blocking situation plays out in a real-time system very similarly to priority inheritance for locking synchronization.

## 8.3 Transactional Memory systems

Transactions are described in the database context by Gray [25], and [26] contains a thorough treatment of database issues. Hardware Transactional Memory (HTM) was first proposed by Knight [46], and Herlihy and Moss coined the term "transactional memory" and proposed HTM in the context of lock-free data structures [40, 36]. The BBN Pluribus [64, Ch. 23] provided transactions, with an architectural limit on the size of a transaction. Experience with Pluribus showed that the headaches of programming correctly with such limits can be at least as challenging as using locks. The ***Oklahoma Update*** is another variation on transactional operations with an architectural limit on the number of values in a transaction [65].

Transactional memory is sometimes described as an extension of Load-Linked/Store-Conditional [43] and other atomic instruction sequences. In fact, some CISC machines, such as the VAX, had complex atomic instructions such as enqueue and dequeue [19].

Of particular relevance are ***Speculative Lock Elision*** (SLE) [57] and ***Transactional Lock Removal*** (TLR) [58], which speculatively identify locks and use the cache to give the appearance of atomicity. SLE and TLR handle mutual exclusion through a standard programmer interface (locks), dynamically translating locks into transactional regions. My research thrust differs from theirs in that I hope to free programmers from the protocol complexities of locking, not just optimize existing practice. The quantitative results presented in this thesis confirm their finding that transactional hardware can be more efficient than locks.

Martinez and Torrellas proposed ***Speculative Synchronization*** [51], which allows some threads to execute atomic regions of code speculatively, using locks, while guaranteeing forward progress by maintaining a nonspeculative thread. These techniques gain many of the performance advantages of transactional memory, but they still require new code to obey a locking protocol to avoid deadlock.

The recent work on ***Transactional memory Coherence and Consistency*** (TCC) [29] is also relevant to our work. TCC uses a broadcast bus to implement the transaction protocols, performing all the writes of a particular transaction in one atomic bus operation. This strategy limits scalability, whereas both the UTM and LTM proposals in chapter 6 can employ scalable cache-consistency protocols to implement transactions. TCC affirms the conclusion we draw from our own Figure 3.3: most transactions are small, but some are very large. TCC supports large transactions by locking the broadcast bus and stalling all other processors when any processor buffer overflows, whereas UTM and LTM allow overlapped execution of multiple large transactions with local overflow buffers. TCC is similar to LTM in that transactions are bound to processor state and cannot extend across page faults, timer interrupts, or thread migrations.

Several software transaction systems have been proposed. Some constrain the programmer and make transactions difficult to use. All have relatively high overheads, which make transactions unattractive for uniprocessor and small SMP systems. [Once the number of processors is large enough, the increased parallelism which can be provided by optimistic transactions may cancel out the performance penalty of their use.]

The first proposal for software transactional memory was proposed by Shavit and Touitou [63]; their system requires that all input and output locations touched by a transaction be known in advance, which limits its application. It performs at least 10 fetches and 4 stores per location accessed (not counting the loads and stores directly required by the computation). The benchmarks presented were run on between 10 and 64 processors.

Rudys and Wallach [60] proposed a copying-based transaction system to allow rollback of hostile codelets. They show an order of magnitude slowdown for field and array accesses, and 6x to 23x slowdown on their benchmarks.

Herlihy, Luchango, Moss, and Scherer's scheme [35] allows transactions to touch a dynamic set of memory locations; however the user still has to

explicitly *open* every object touched before it can be used in a transaction. This implementation is based on object copying, and so has poor performance for large objects and arrays. Not including work necessary to copy objects involved in writes, they require $O(R(R+W))$ work to open R objects for reading and $W$ objects for writing, which may be quadratic in the number of objects involved in the transaction. A list insertion benchmark which they present shows 9x slowdown over a locking scheme, although they beat the locking implementation when more than 5-10 processors are active. They present benchmark data with up to 576 threads on 72 processors.

Harris and Fraser built a software transaction system on a flat word-oriented transactional memory abstraction [30], roughly similar to simulating Herlihy's original hardware transactional memory proposal in software. This avoids problems with large objects. Performing $m$ memory operations touching $l$ distinct locations costs at least $m+l$ extra reads and $l+1$ CAS operations, in addition to the reads and writes required by the computation. They appear to execute about twice as slowly as a locking implementation on some microbenchmarks. They benchmark on a 4-processor as well as a 106-processor machine; their crossover point (at which the blocking overhead of locks matches the software transaction overhead) is around 4 processors. Note that Harris and Fraser do not address the problem of concurrent non-transactional operations on locations involved in a transaction. Java synchronization allows such concurrent operations, with semantics given by the Java memory model [50]. We support these operations safely using the mechanisms presented in chapter 3.

Programmers will be reluctant to use transactions to synchronize their code when it results in their code running more slowly on the uniprocessor and small-SMP systems which are most common today.

Herlihy and Moss [36] suggested that small transactions might be handled in cache with overflows handled by software. These software overflows must interact with the transactional hardware in the same way that the hardware interacts with itself, however. In chapter 6.4 we present just such

a system.

## 8.4 Language-level approaches to synchronization

Our work on integrating transactions into the Java programming language is related to prior work on integrating synchronization mechanisms for multiprogramming, and in particular, to prior work on synchronization in an object-oriented framework.

The Emerald system [12, 45] introduced *monitored objects* for synchronization. Emerald code to implement a simple directory object is shown in Figure 8.1. Each object is associated with Hoare-style monitor, which provides mutual exclusion and process signaling. Each Emerald object is divided into a monitored part and a non-monitored part. Variables declared in the monitored part are shared, and access to them from methods in the non-monitored part is prohibited—although non-monitored methods may call monitored methods to effect the access. Methods in the monitored part acquire the monitor lock associated with the receiver object before entry and release it on exit, providing for mutual exclusion and safe update of the shared variables. Monitored objects naturally integrate synchronization into the object model.

Unlike Emerald monitored objects, where methods can only acquire the monitor of their receiver and where restricted access to shared variables is enforced by the compiler, Java implements a loose variant where any monitor may be explicitly acquired and no shared variable protection exists. As a default, however, Java methods declared with the `synchronized` keyword behave like Emerald monitored methods, ensuring that the monitor lock of their receiver is held during execution.

Java's synchronization primitives arguably allow for more efficient concurrent code than Emerald's—for example, Java objects can use multiple locks to protect disjoint sets of fields, and coarse-grain locks can be used which protect multiple objects—but Java is also more prone to programmer

```
const myDirectory == object oneEntryDirectory
  export Store, Lookup
  monitor
    var name : String
    var AnObject : Any

    operation Store [ n : String, o : Any ]
      name ← n
      AnObject ← o
    end Store
    function Lookup [ n : String ] → [ o : Any ]
      if n = name
        then o ← AnObject
        else o ← nil
      end if
    end Lookup

    initially
      name ← nil
      AnObject ← nil
    end initially

  end monitor
end oneEntryDirectory
```

Figure 8.1: A directory object in Emerald, from [12], illustrating the use of monitor synchronization.

```
class Account {

  int balance = 0;

  atomic int deposit(int amt) {
    int t = this.balance;
    t = t + amt;
    this.balance = t;
    return t;
  }

  atomic int readBalance() {
    return this.balance;
  }

  atomic int withdraw(int amt) {
    int t = this.balance;
    t = t - amt;
    this.balance = t;
    return t;
  }

}
```

Figure 8.2: A simple bank account object, adapted from [20], illustrating the use of the `atomic` modifier.

error. However, even Emerald's restrictive monitored objects are not sufficient to prevent data races. As a simple example, imagine that an object provided two monitored methods `read` and `write` which accessed a shared variable. Non-monitored code can call `read`, increment the value returned, and then call `write`, creating a classic race condition scenario. The atomicity of the parts is not sufficient to guarantee atomicity of the whole [20].

This suggests that a better model for synchronization in object-oriented systems is *atomicity*. Figure 8.2 shows Java extended with an `atomic` keyword to implement an object representing a bank account. Rather than explicitly synchronizing on locks, I simply require that the methods marked `atomic` execute atomically with respect to other threads in the system; that is, that every execution of the program computes the same result as some execution where all atomic methods were run *in isolation* at a certain point in time, called the ***linearization point***, between their invocation and return. Note that atomic methods invoked directly or indirectly from an atomic method are subsumed by it: if the outermost method appears atomic, then by definition all inner method invocations will also appear atomic. Flanagan and Qadeer provide a more formal semantics in [20]. Atomic methods can be analyzed using sequential reasoning techniques, which significantly simplifies reasoning about program correctness.

Atomic methods can be implemented using locks. A simple if inefficient implementation would simply acquire a single global lock during the execution of every atomic method. Flanagan and Qadeer [20] present a more sophisticated technique which proves that a given implementation using standard Java monitors correctly guarantees method atomicity.

The transaction implementations presented in this thesis will use nonblocking synchronization to implement atomic methods.

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

Lewis Carroll, *Alice's Adventures in Wonderland*

# Chapter 9

# Conclusion

In this thesis I have shown that it is possible to implement an efficient strongly-atomic software transaction system, and that non-blocking transactions can be used in applications beyond synchronization, including fault tolerance and backtracking search. I have presented implementation details to address the practical problems of building such a system. I have argued the transactions should not be subject to limits on size or duration, and presented both software and hardware implementations free of such restrictions. Finally, the low overhead of my software system allows it to be profitably combined with a hardware transaction system; I have shown how this hybrid yields fast execution of short and small transactions while allowing fallback to software for large or complicated transactions.

There is no escape: parallel systems are in our future. However, programming them does not have to be as fraught as it is presently. I believe that transactions provide a programmer-friendly model of concurrency which eliminates many potential pitfalls of our current locking-based methodologies.

In this thesis I have presented several designs for efficient transaction systems that will enable us to take advantage of the transactional programming model. The software-only system runs on current hardware, and LTM and UTM indicate possible directions for future hardware. However, there

are challenges and design decisions remaining: how should I/O be handled? What are the proper semantics for nested transactions? What loophole mechanisms are necessary to allow information about a transaction's progress to escape?

I believe hybrid systems are the best answer to these challenges: they combine the inherent speed of hardware systems with the flexibility of software, allowing novel solutions to be attempted without requiring that design decisions be cast in silicon. The flag-based software transaction system described in this thesis imposes very low overhead, allowing transactional programming to get off the ground without hardware support in the near term, while later supporting the development of new transactional models and methodologies as part of a hybrid system.

Designing correct transaction systems is not easy, however. In the appendix you will find a Promela model of my software transaction system. Automated verification was essential when designing and debugging the system, uncovering via exhaustive enumeration race conditions much too subtle for me to discover by other means. I believe any credible transaction system must be buttressed by formal verification.

Essentially, all models are
wrong, but some are useful.

George E. P. Box, *Empirical Model-Building and Response Surfaces*

# Appendix A

# Model-checking the software implementation

My work on both software and hardware transaction systems has reiterated the difficulty of creating correct implementations of concurrent and fault-tolerant constructs. Automatic model checking is a prerequisite to achieving confidence in the design and implementation. Versions of the software transaction system have been modeled in Promela using SPIN 4.1.0 and verified on an SGI 64-processor MIPS machine with 16G of main memory.

Sequences of transactional and non-transactional load and store operations were checked using two concurrent processes, and all possible interleavings were found to produce results consistent with the semantic atomicity of the transactions. Several test scripts were run against the model using separate processors of the verification machine (SPIN cannot otherwise exploit SMP). Some representative costs include:

- testing two concurrent `writeT` operations (including "false flag" conditions) against a single object required $3.8 \times 10^6$ states and 170M memory;

- testing sequences of transactional and non-transactional reads and writes against two objects (checking that all views of the two objects

were consistent) required $4.6 \times 10^6$ states and 207M memory; and

- testing a pair of concurrent increments at values bracketing the FLAG value to 99.8% coverage of the state space required $7.6 \times 10^7$ states and 4.3G memory. Simultaneously model-checking a range of values caused the state space explosion in this case.

SPIN's unreachable code reporting was used to ensure that our test cases exercised every code path, although this doesn't guarantee that every interesting interaction is checked.

In the process one bug in SPIN was discovered[1] and a number of subtle race conditions in the model were discovered and corrected. These included a number of modeling artifacts: in particular, we were extremely aggressive about reference-counting and deallocating objects in order to control the size of the state space, and this proved difficult to do correctly. We also discovered some subtle-but-legitimate race conditions in the transactions algorithm. For example:

- A race allowed conflicting readers to be created while a writer was inside `ensureWriter` creating a new version object.
- Allowing already-committed version objects to be mutated when `writeT` or `writeNT` was asked to store a "false flag" produced races between `ensureWriter` and `copyBackField`. The code that was expected to manage these races had unexpected corner cases.
- Using a bitmask to provide per-field granularity to the list of readers proved unmanageable as there were three-way races between the bitmask, the readers list, and the version tree.

In addition, the model-in-progress proved a valuable design tool, as portions of the algorithm could readily be mechanically checked to validate (or dis-

[1] Breadth-first search of atomic regions was performed incorrectly in SPIN 4.0.7; this was fixed in SPIN 4.1.0.

credit) the designer's reasoning about the concurrent system. Humans do not excel at exhaustive state space exploration.

SPIN is not particularly suited to checking models with dynamic allocation and deallocation. In particular, it considers the location of objects part of the state space, and allocating object A before object B produces a different state than if object B were allocated first. This artificially enlarges the state space. A great deal of effort was expended tweaking the model to approach a canonical allocation ordering. A better solution to this problem would allow larger model instances to be checked.

## A.1 Promela primer

A concise Promela reference is available at http://spinroot.com/spin/Man/Quick.html; we will here attempt to summarize just enough of the language to allow the model we've presented in this thesis to be understood.

Promela syntax is C-like, with the same lexical and commenting conventions. Statements are separated by either a semi-colon, or, equivalently, an arrow. The arrow is typically used to separate a guard expression from the statements it is guarding.

The program counter moves past a statement only if the statement is *enabled*. Most statements, including assignments, are always enabled. A statement consisting only of an expression is enabled iff the expression is true (non-zero). Our model uses three basic Promela statements: selection, repetition, and `atomic`.

The selection statement,

```
if
:: guard -> statements
...
:: guard -> statements
fi
```

selects one among its options and executes it. An option can be selected iff its first statement (the guard) is enabled. The special guard `else` is enabled

iff all other guards are not.

The repetition statement,

```
do
:: statements
...
:: statements
fi
```

is similar: one among its enabled statements is selected and executed, and then the process is repeated (with a different statement possibly being selected each time) until control is explicitly transferred out of the loop with a break or goto.

Finally,

```
atomic { statements }
```

executes the given statements in one indivisible step. For the purposes of this model, a d_step block is functionally identical. Outside atomic or d_step blocks, Promela allows interleaving before and after every statement, but statements are indivisible.

Functions as specified in this model are similar to C macros: every parameter is potentially both an input *and* an output. Calls to functions with names starting with move are simple assignments; they've been turned into macros so that reference counting can be performed.

## A.2 Spin model for software transaction system

The complete SPIN 4.1.0 model for the FLEX software transaction system is presented here. It may also be downloaded from http://flex-master.csail.mit.edu/Harpoon/swx.pml.

```
/*************************************************************************
 * Very detailed model of software transaction code.
 * Checking for safety and correctness properties.  Not too worried about
 * liveness at the moment.
```

```
 *
 * (C) 2006 C. Scott Ananian <cananian@alumni.princeton.edu>
 ************************************************************************/

/* CURRENT ISSUES:
 *  none known.
 */
/* MINOR ISSUES:
 * 1) use smaller values for FLAG and NIL to save state space?
 */

/* Should use Spin 4.1.0, for correct nested-atomic behavior. */

#define REFCOUNT

#define NUM_OBJS 2
#define NUM_VERSIONS 6 /* each obj: committed and waiting version, plus nonce
                        * plus addition nonce for NT copyback in test3 */
#define NUM_READERS 4 /* both 'read' trans reading both objs */
#define NUM_TRANS 5 /* two 'real' TIDs, plus 2 outstanding TIDs for
     * writeNT(FLAG) [test3], plus perma-aborted TID. */
#define NUM_FIELDS 2

#define NIL 255 /* special value to represent 'alloc impossible', etc. */
#define FLAG 202 /* special value to represent 'not here' */

typedef Object {
  byte version;
  byte readerList; /* we do LL and CAS operations on this field */
  pid fieldLock[NUM_FIELDS]; /* we do LL operations on fields */
  byte field[NUM_FIELDS];
};
typedef VersiOn { /* 'Version' misspelled because spin #define's it. */
  byte owner;
  byte next;
  byte field[NUM_FIELDS];
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
```

```
};
typedef ReaderList {
  byte transid;
  byte next;
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
};
mtype = { waiting, committed, aborted };
typedef TransID {
  mtype status;
#ifdef REFCOUNT
  byte ref; /* reference count */
#endif /* REFCOUNT */
};
Object object[NUM_OBJS];
VersiOn version[NUM_VERSIONS];
ReaderList readerlist[NUM_READERS];
TransID transid[NUM_TRANS];
byte aborted_tid; /* global variable; 'perma-aborted' */

/* --------------------------- alloc.pml ----------------------- */
mtype = { request, return };

inline manager(NUM_ITEMS, allocchan) {
  chan pool = [NUM_ITEMS] of { byte };
  chan client;
  byte nodenum;
  /* fill up the pool with node identifiers */
  d_step {
    i=0;
    do
    :: i<NUM_ITEMS -> pool!!i; i++
    :: else -> break
    od;
  }
end:
  do
  :: allocchan?request(client,_) ->
```

```
     if
     :: empty(pool) -> assert(0); client!NIL /* deny */
     :: nempty(pool) ->
pool?nodenum;
client!nodenum;
nodenum=0
     fi
  :: allocchan?return(client,nodenum) ->
     pool!!nodenum; /* sorted, to reduce state space */
     nodenum=0
  od
}

chan allocObjectChan = [0] of { mtype, chan, byte };
active proctype ObjectManager() {
  atomic {
    byte i;
    manager(NUM_OBJS, allocObjectChan)
  }
}
chan allocVersionChan = [0] of { mtype, chan, byte };
active proctype VersionManager() {
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_VERSIONS ->
 version[i].owner=NIL; version[i].next=NIL;
 version[i].field[0]=FLAG; version[i].field[1]=FLAG;
 assert(NUM_FIELDS==2);
 i++
      :: else -> break
      od;
    }
    manager(NUM_VERSIONS, allocVersionChan)
  }
}
chan allocReaderListChan = [0] of { mtype, chan, byte };
active proctype ReaderListManager() {
```

```
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_READERS ->
 readerlist[i].transid=NIL; readerlist[i].next=NIL;
 i++
      :: else -> break
      od;
    }
    manager(NUM_READERS, allocReaderListChan)
  }
}
chan allocTransIDChan = [0] of { mtype, chan, byte };
active proctype TransIDManager() {
  atomic {
    byte i=0;
    d_step {
      do
      :: i<NUM_TRANS -> transid[i].status=waiting; i++
      :: else -> break
      od;
    }
    manager(NUM_TRANS, allocTransIDChan)
  }
}

inline alloc(allocchan, retval, result) {
  result = NIL;
  do
  :: result != NIL -> break
  :: else -> allocchan!request(retval,0) ; retval ? result
  od;
  skip /* target of break. */
}
inline free(allocchan, retval, result) {
  allocchan!return(retval,result)
}
inline allocObject(retval, result) {
```

```
  atomic {
    alloc(allocObjectChan, retval, result);
    d_step {
      object[result].version = NIL;
      object[result].readerList = NIL;
      object[result].field[0] = 0;
      object[result].field[1] = 0;
      object[result].fieldLock[0] = _thread_id;
      object[result].fieldLock[1] = _thread_id;
      assert(NUM_FIELDS==2); /* else ought to initialize more fields */
    }
  }
}
inline allocTransID(retval, result) {
  atomic {
    alloc(allocTransIDChan, retval, result);
    d_step {
      transid[result].status = waiting;
#ifdef REFCOUNT
      transid[result].ref = 1;
#endif /* REFCOUNT */
    }
  }
}
inline moveTransID(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
    :: (src!=NIL) ->
       transid[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       transid[dst].ref--;
       if
       :: (transid[dst].ref==0) -> _free=dst
       :: else
```

```
       fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeTransID(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
proctype freeTransID(byte result; chan retval) {
  chan _retval = [0] of { byte };
  atomic {
#ifdef REFCOUNT
    assert(transid[result].ref==0);
#endif /* REFCOUNT */
    transid[result].status = waiting;
    free(allocTransIDChan, _retval, result)
    retval!0; /* done */
  }
}
inline allocVersion(retval, result, a_transid, tail) {
  atomic {
    alloc(allocVersionChan, retval, result);
    d_step {
#ifdef REFCOUNT
      if
      :: (a_transid!=NIL) -> transid[a_transid].ref++;
      :: else
      fi;
      if
      :: (tail!=NIL) -> version[tail].ref++;
      :: else
      fi;
      version[result].ref = 1;
```

```
#endif /* REFCOUNT */
      version[result].owner = a_transid;
      version[result].next = tail;
      version[result].field[0] = FLAG;
      version[result].field[1] = FLAG;
      assert(NUM_FIELDS==2); /* else ought to initialize more fields */
    }
  }
}
inline moveVersion(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
    :: (src!=NIL) ->
       version[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       version[dst].ref--;
       if
       :: (version[dst].ref==0) -> _free=dst
       :: else
       fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeVersion(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
proctype freeVersion(byte result; chan retval) {
```

```
  chan _retval = [0] of { byte };
  byte _free;
  atomic { /* zero out version structure */
#ifdef REFCOUNT
    assert(version[result].ref==0);
#endif /* REFCOUNT */
    moveTransID(version[result].owner, NIL);
    moveVersion(version[result].next, NIL);
    version[result].field[0] = FLAG;
    version[result].field[1] = FLAG;
    assert(NUM_FIELDS==2);
    free(allocVersionChan, _retval, result)
    retval!0; /* done */
  }
}

inline allocReaderList(retval, result, head, tail) {
  atomic {
    assert(head!=NIL);
    alloc(allocReaderListChan, retval, result);
    d_step {
#ifdef REFCOUNT
      readerlist[result].ref = 1;
      transid[head].ref++;
      if
      :: (tail!=NIL) -> readerlist[tail].ref++
      :: else
      fi;
#endif /* REFCOUNT */
      readerlist[result].transid = head;
      readerlist[result].next = tail;
    }
  }
}
inline moveReaderList(dst, src) {
  atomic {
#ifdef REFCOUNT
    _free = NIL;
    if
```

```
    :: (src!=NIL) ->
       readerlist[src].ref++
    :: else
    fi;
    if
    :: (dst!=NIL) ->
       readerlist[dst].ref--;
       if
       :: (readerlist[dst].ref==0) -> _free=dst
       :: else
       fi
    :: else
    fi;
#endif /* REFCOUNT */
    dst = src;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeReaderList(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif
  }
}
proctype freeReaderList(byte result; chan retval) {
  chan _retval = [0] of { byte };
  byte _free;
  atomic {
#ifdef REFCOUNT
    assert(readerlist[result].ref==0);
#endif /* REFCOUNT */
    moveTransID(readerlist[result].transid, NIL);
    moveReaderList(readerlist[result].next, NIL);
    free(allocReaderListChan, _retval, result)
    retval!0; /* done */
  }
}

/* -------------------------- atomic.pml ---------------------- */
```

```
inline DCAS(loc1, oval1, nval1, loc2, oval2, nval2, st) {
  d_step {
    if
    :: (loc1==oval1) && (loc2==oval2) ->
       loc1=nval1;
       loc2=nval2;
       st=true
    :: else ->
       st=false
    fi
  }
}
inline CAS(loc1, oval1, nval1, st) {
  d_step {
    if
    :: (loc1==oval1) ->
       loc1=nval1;
       st=true
    :: else ->
       st=false
    fi
  }
}
inline CAS_Version(loc1, oval1, nval1, st) {
  atomic {
    _free = NIL;
    if
    :: (loc1==oval1) ->
#ifdef REFCOUNT
       if
       :: (nval1!=NIL) -> version[nval1].ref++;
       :: else
       fi;
       if
       :: (oval1!=NIL) -> version[oval1].ref--;
  if
  :: (version[oval1].ref==0) -> _free = oval1
  :: else
  fi
```

```
       :: else
       fi;
#endif /* REFCOUNT */
       loc1=nval1;
       st=true
    :: else ->
       st=false
    fi;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeVersion(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}
inline CAS_Reader(loc1, oval1, nval1, st) {
  atomic {
    /* save oval1, as it could change as soon as we leave the d_step */
    _free = NIL;
    if
    :: (loc1==oval1) ->
#ifdef REFCOUNT
       if
       :: (nval1!=NIL) -> readerlist[nval1].ref++;
       :: else
       fi;
       if
       :: (oval1!=NIL) -> readerlist[oval1].ref--;
  if
  :: (readerlist[oval1].ref==0) -> _free = oval1
  :: else
  fi
       :: else
       fi;
#endif /* REFCOUNT */
       loc1=nval1;
       st=true
```

```
    :: else ->
       st=false
    fi;
#ifdef REFCOUNT
    /* receive must be last, as it breaks atomicity. */
    if
    :: (_free!=NIL) -> run freeReaderList(_free, _retval); _free=NIL; _retval?_
    :: else
    fi
#endif /* REFCOUNT */
  }
}

/* ---------------- end atomic.pml ------------------ */

mtype = { kill_writers, kill_all };
mtype = { success, saw_race, saw_race_cleanup, false_flag };

inline tryToAbort(t) {
  assert(t!=NIL);
  CAS(transid[t].status, waiting, aborted, _);
  assert(transid[t].status==aborted || transid[t].status==committed)
}
inline tryToCommit(t) {
  assert(t!=NIL);
  CAS(transid[t].status, waiting, committed, _);
  assert(transid[t].status==aborted || transid[t].status==committed)
}
inline copyBackField(o, f, mode, st) {
  _nonceV=NIL; _ver = NIL; _r = NIL; st = success;
  /* try to abort each version.  when abort fails, we've got a
   * committed version. */
  do
  :: moveVersion(_ver, object[o].version);
     if
     :: (_ver==NIL) ->
st = saw_race; break /* someone's done the copyback for us */
     :: else
     fi;
```

```
      /* move owner to local var to avoid races (owner set to NIL behind
       * our back) */
     _tmp_tid=NIL;
     moveTransID(_tmp_tid, version[_ver].owner);
     if
     :: (_tmp_tid==NIL) ->
break; /* found a committed version */
     :: else
     fi;
     tryToAbort(_tmp_tid);
     if
     :: (transid[_tmp_tid].status==committed) ->
moveTransID(_tmp_tid, NIL);
moveTransID(version[_ver].owner, NIL); /* opportunistic free */
moveVersion(version[_ver].next, NIL); /* opportunistic free */
break /* found a committed version */
     :: else
     fi;
     /* link out an aborted version */
     assert(transid[_tmp_tid].status==aborted);
     CAS_Version(object[o].version, _ver, version[_ver].next, _);
     moveTransID(_tmp_tid, NIL);
  od;
  /* okay, link in our nonce.  this will prevent others from doing the
   * copyback. */
  if
  :: (st==success) ->
     assert (_ver!=NIL);
     allocVersion(_retval, _nonceV, aborted_tid, _ver);
     CAS_Version(object[o].version, _ver, _nonceV, _cas_stat);
     if
     :: (!_cas_stat) ->
st = saw_race_cleanup
     :: else
     fi
  :: else
  fi;
  /* check that no one's beaten us to the copy back */
  if
```

```
  :: (st==success) ->
     if
     :: (object[o].field[f]==FLAG) ->
_val = version[_ver].field[f];
if
:: (_val==FLAG) -> /* false flag... */
   st = false_flag /* ...no copy back needed */
:: else -> /* not a false flag */
   d_step { /* could be DCAS */
     if
     :: (object[o].version == _nonceV) ->
object[o].fieldLock[f] = _thread_id;
object[o].field[f] = _val;
     :: else /* hmm, fail.  Must retry. */
st = saw_race_cleanup /* need to clean up nonce */
     fi
   }
fi
     :: else /* may arrive here because of readT, which doesn't set _val=FLAG*/
st = saw_race_cleanup /* need to clean up nonce */
     fi
  :: else /* !success */
  fi;

  /* always kill readers, whether successful or not.  This ensures that we
   * make progress if called from writeNT after a readNT sets readerList
   * non-null without changing FLAG to _val (see immediately above; st will
   * equal saw_race_cleanup in this scenario). */
  if
  :: (mode == kill_all) ->
     do /* kill all readers */
     :: moveReaderList(_r, object[o].readerList);
if
:: (_r==NIL) -> break
:: else
fi;
tryToAbort(readerlist[_r].transid);
/* link out this reader */
CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _);
```

```
     od;
  :: else /* no more killing needed. */
  fi;

  /* finally, clean up our mess. */
  moveVersion(_ver, NIL);
  if
  :: (st == saw_race_cleanup || st == success || st == false_flag) ->
     CAS_Version(object[o].version, _nonceV, version[_nonceV].next, _);
     moveVersion(_nonceV, NIL);
     if
     :: (st==saw_race_cleanup) -> st=saw_race
     :: else
     fi
  :: else
  fi;
  /* done */
  assert(_nonceV==NIL);
}

inline readNT(o, f, v) {
  do
  :: v = object[o].field[f];
     if
     :: (v!=FLAG) -> break /* done! */
     :: else
     fi;
     copyBackField(o, f, kill_writers, _st);
     if
     :: (_st==false_flag) ->
v = FLAG;
break
     :: else
     fi
  od
}
inline writeNT(o, f, nval) {
  if
  :: (nval != FLAG) ->
```

```
     do
     ::
atomic {
  if /* this is a LL(readerList)/SC(field) */
  :: (object[o].readerList == NIL) ->
     object[o].fieldLock[f] = _thread_id;
     object[o].field[f] = nval;
     break /* success! */
  :: else
  fi
}
/* unsuccessful SC */
copyBackField(o, f, kill_all, _st)
/* ignore return status */
     od
  :: else -> /* create false flag */
     /* implement this as a short *transactional* write.  this may be slow,
      * but it greatly reduces the race conditions we have to think about. */
     do
     :: allocTransID(_retval, _writeTID);
ensureWriter(_writeTID, o, _tmp_ver);
checkWriteField(o, f);
writeT(_tmp_ver, f, nval);
tryToCommit(_writeTID);
moveVersion(_tmp_ver, NIL);
if
:: (transid[_writeTID].status==committed) ->
   moveTransID(_writeTID, NIL);
   break /* success! */
:: else ->/* try again */
   moveTransID(_writeTID, NIL)
fi
     od
  fi;
}
inline readT(tid, o, f, ver, result) {
  do
  ::
     /* we should always either be on the readerlist or aborted here */
```

```
     atomic { /* complicated assertion; evaluate atomically */
       if
       :: (transid[tid].status == aborted) -> skip /* okay then */
       :: else ->
  assert (transid[tid].status == waiting);
  _r = object[o].readerList;
  do
  :: (_r==NIL || readerlist[_r].transid==tid) -> break
  :: else -> _r = readerlist[_r].next
  od;
  assert (_r!=NIL); /* we're on the list */
  _r = NIL /* back to normal */
       fi
     }
     /* okay, sanity checking done -- now let's get to work! */
     result = object[o].field[f];
     if
     :: (result==FLAG) ->
if
:: (ver!=NIL) ->
   result = version[ver].field[f];
   break /* done! */
:: else ->
   findVersion(tid, o, ver);
   if
   :: (ver==NIL) -> /* use value from committed version */
      assert (_r!=NIL);
      result = version[_r].field[f]; /* false flag? */
      moveVersion(_r, NIL);
      break /* done */
   :: else /* try, try, again */
   fi
fi
     :: else -> break /* done! */
     fi
  od
}
inline writeT(ver, f, nval) {
  /* easy enough: */
```

```
  version[ver].field[f] = nval;
}

/* make sure 'tid' is on reader list. */
inline ensureReaderList(tid, o) {
  /* add yourself to readers list. */
  _rr = NIL; _r = NIL;
  do
  :: moveReaderList(_rr, object[o].readerList); /* first_reader */
     moveReaderList(_r, _rr);
     do
     :: (_r==NIL) ->
break /* not on the list */
     :: (_r!=NIL && readerlist[_r].transid==tid) ->
break /* on the list */
     :: else ->
/* opportunistic free? */
if
:: (_r==_rr && transid[readerlist[_r].transid].status != waiting) ->
   CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _)
   if
   :: (_cas_stat) -> moveReaderList(_rr, readerlist[_r].next)
   :: else
   fi
:: else
fi;
/* keep looking */
moveReaderList(_r, readerlist[_r].next)
     od;
     if
     :: (_r!=NIL) ->
break /* on the list; we're done! */
     :: else ->
/* try to put ourselves on the list. */
assert(tid!=NIL && _r==NIL);
allocReaderList(_retval, _r, tid, _rr);
CAS_Reader(object[o].readerList, _rr, _r, _cas_stat);
if
:: (_cas_stat) ->
```

```
   break /* we're on the list */
:: else
fi
/* failed to put ourselves on the list, retry. */
     fi
  od;
  moveReaderList(_rr, NIL);
  moveReaderList(_r, NIL);
  /* done. */
}

/* look for a version read/writable by 'tid'.  Returns:
 *   ver!=NIL -- ver is the 'waiting' version for 'tid'. _r == NIL.
 *   ver==NIL, _r != NIL -- _r is the first committed version in the chain.
 *   ver==NIL, _r == NIL -- there are no commited versions for this object
 *                          (i.e. object[o].version==NIL)
 */
inline findVersion(tid, o, ver) {
  assert(tid!=NIL);
  _r = NIL; ver = NIL; _tmp_tid=NIL;
  do
  :: moveVersion(_r, object[o].version);
     if
     :: (_r==NIL) -> break /* no versions. */
     :: else
     fi;
     moveTransID(_tmp_tid, version[_r].owner);/*use local copy to avoid races*/
     if
     :: (_tmp_tid==tid) ->
ver = _r; /* found a version: ourself! */
_r = NIL; /* transfer owner of the reference to ver, without ++/-- */
break
     :: (_tmp_tid==NIL) ->
/* perma-committed version.  Return in _r. */
moveVersion(version[_r].next, NIL); /* opportunistic free */
break
     :: else -> /* strange version.  try to kill it. */
/* ! could double-check that our own transid is not aborted here. */
tryToAbort(_tmp_tid);
```

```
if
:: (transid[_tmp_tid].status==committed) ->
   /* committed version.  Return this in _r. */
   moveTransID(version[_r].owner, NIL); /* opportunistic free */
   moveVersion(version[_r].next, NIL); /* opportunistic free */
   break /* no need to look further. */
:: else ->
   assert (transid[_tmp_tid].status==aborted);
   /* unlink this useless version */
   CAS_Version(object[o].version, _r, version[_r].next, _)
   /* repeat */
fi
     fi
  od;
  moveTransID(_tmp_tid, NIL); /* free tmp transid copy */
  assert (ver!=NIL -> _r == NIL : 1)
}

inline ensureReader(tid, o, ver) {
  assert(tid!=NIL);
  /* make sure we're on the readerlist */
  ensureReaderList(tid, o)
  /* now kill any transactions associated with uncommitted versions, unless
   * the transaction is ourself! */
  findVersion(tid, o, ver);
  /* don't care about which committed version to use, at the moment. */
  moveVersion(_r, NIL);
}

/* per-object, before write. */
/* returns NIL in ver to indicate suicide. */
inline ensureWriter(tid, o, ver) {
  assert(tid!=NIL);
  /* Same beginning as ensureReader */
  ver = NIL; _r = NIL; _rr = NIL;
  do
  :: assert (ver==NIL);
     findVersion(tid, o, ver);
     if
```

```
     :: (ver!=NIL) -> break /* found a writable version for us */
     :: (ver==NIL && _r==NIL) ->
/* create and link a fully-committed root version, then
 * use this as our base. */
allocVersion(_retval, _r, NIL, NIL);
CAS_Version(object[o].version, NIL, _r, _cas_stat)
     :: else ->
_cas_stat = true
     fi;
     if
     :: (_cas_stat) ->
/* so far, so good. */
assert (_r!=NIL);
assert (version[_r].owner==NIL ||
transid[version[_r].owner].status==committed);
/* okay, make new version for this transaction. */
assert (ver==NIL);
allocVersion(_retval, ver, tid, _r);
/* want copy of committed version _r.  Race here because _r can be
 * written to under peculiar circumstances, namely: _r has
 * non-flag value, non-flag value is copied back to parent,
 * flag_value is written to parent -- this forces flag_value to
 * be written to committed version. */
/* IF WRITES ARE ALLOWED TO COMMITTED VERSIONS, THERE IS A RACE HERE.
 * But our implementation of false_flag writes at the moment does
 * not permit *any* writes to committed versions. */
version[ver].field[0] = version[_r].field[0];
version[ver].field[1] = version[_r].field[1];
assert(NUM_FIELDS==2); /* else ought to initialize more fields */
CAS_Version(object[o].version, _r, ver, _cas_stat);
moveVersion(_r, NIL); /* free _r */
if
:: (_cas_stat) ->
   /* kill all readers (except ourself) */
   /* note that all changes have to be made from the front of the
    * list, so we unlink ourself and then re-add us. */
   do
   :: moveReaderList(_r, object[o].readerList);
      if
```

```
      :: (_r==NIL) -> break
      :: (_r!=NIL && readerlist[_r].transid!=tid)->
 tryToAbort(readerlist[_r].transid)
      :: else
      fi;
      /* link out this reader */
      CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _)
   od;
   /* okay, all pre-existing readers dead & gone. */
   assert(_r==NIL);
   /* link us back in. */
   ensureReaderList(tid, o);
   break
:: else
fi;
/* try again */
     :: else
     fi;
     /* try again from the top */
     moveVersion(ver, NIL)
  od;
  /* done! */
  assert (_r==NIL);
}
/* per-field, before read. */
inline checkReadField(o, f) {
  /* do nothing: no per-field read stats are kept. */
  skip
}
/* per-field, before write. */
inline checkWriteField(o, f) {
  _r = NIL; _rr = NIL;
  do
  ::
     /* set write flag, if not already set */
     _val = object[o].field[f];
     if
     :: (_val==FLAG) ->
break; /* done! */
```

```
     :: else
     fi;
     /* okay, need to set write flag. */
     moveVersion(_rr, object[o].version);
     moveVersion(_r, _rr);
     assert (_r!=NIL);
     do
     :: (_r==NIL) -> break /* done */
     :: else ->
object[o].fieldLock[f] = _thread_id;
if
/* this next check ensures that concurrent copythroughs don't stomp
 * on each other's versions, because the field will become FLAG
 * before any other version will be written. */
:: (object[o].field[f]==_val) ->
   if
   :: (object[o].version==_rr) ->
      atomic {
if
:: (object[o].fieldLock[f]==_thread_id) ->
   version[_r].field[f] = _val;
:: else -> break /* abort */
fi
      }
   :: else -> break /* abort */
   fi
:: else -> break /* abort */
fi;
moveVersion(_r, version[_r].next) /* on to next */
     od;
     if
     :: (_r==NIL) ->
/* field has been successfully copied to all versions */
atomic {
  if
  :: (object[o].version==_rr) ->
     assert(object[o].field[f]==_val ||
    /* we can race with another copythrough and that's okay;
     * the locking strategy above ensures that we're all
```

```
      * writing the same values to all the versions and not
      * overwriting anything. */
     object[o].field[f]==FLAG);
      object[o].fieldLock[f]=_thread_id;
      object[o].field[f] = FLAG;
      break; /* success!  done! */
   :: else
   fi
}
      :: else
      fi
      /* retry */
   od;
   /* clean up */
   moveVersion(_r, NIL);
   moveVersion(_rr, NIL);
}
```

# Bibliography

[1] San Jose, California, Oct. 5–9 2002. ACM Press. [2 citations on pages 188 and 189.]

[2] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *PODC '92*, pages 125–134, Vancouver, British Columbia, Canada, Aug. 1992. [1 citation on page 144.]

[3] C. S. Ananian. The FLEX compiler project. http://flex-compiler.csail.mit.edu/. [2 citations on pages 33 and 63.]

[4] C. S. Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, Sept. 1999. [1 citation on page 69.]

[5] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA-11*, pages 316–327, San Francisco, California, Feb. 2005. [6 citations on pages 22, 28, 38, 109, and 128.]

[6] A. W. Appel. Simple generational garbage collection and fast allocation. *Software–Practice and Experience*, 19(2):171–183, Feb. 1989. [1 citation on page 89.]

[7] A. W. Appel and G. J. Jacobson. The world's fastest Scrabble program.

*Communications of the ACM*, 31(5):572–578, May 1988. [1 citation on page 26.]

[8] H. G. Baker. Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices*, 26(8):145–147, Aug. 1991. [1 citation on page 102.]

[9] H. G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978. [1 citation on page 102.]

[10] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 261–270. ACM Press, June 1993. [1 citation on page 143.]

[11] A. Bensoussan, C. Clingen, and R. Daley. The Multics virtual memory: Concepts and design. *CACM*, 15(5):308–318, May 1972. [1 citation on page 119.]

[12] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–86. ACM Press, Sept. 1986. [3 citations on pages 29, 148, and 149.]

[13] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, Wisconsin, June 2005. [2 citations on pages 17 and 133.]

[14] H. Boehm, A. Demers, and M. Weiser. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/, 1991. [1 citation on page 70.]

[15] T.-R. Chuang. A randomized implementation of multiple functional arrays. In *LFP*, pages 173–184, June 1994. [3 citations on pages 91, 101, and 102.]

[16] C. Click, G. Tene, and M. Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual Execution Environments*, pages 46–56, Chicago, Illinois, June 2005. [1 citation on page 129.]

[17] S. Cohen. Multi-version structures in Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 265–274, Nov. 1984. [1 citation on page 100.]

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd edition, 2001. [1 citation on page 27.]

[19] Digital Equipment Corporation. *VAX MACRO and Instruction Set Reference Manual*, Nov. 1996. [1 citation on page 145.]

[20] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '98*, pages 338–349, Montreal, Quebec, Canada, June 1998. [5 citations on pages 29, 150, and 151.]

[21] Freescale Semiconductor, Chandler, Arizona. *MPC7450 RISC Microprocessor Family Reference Manual, Rev. 5*, Jan. 2005. MPC7450UM. [1 citation on page 42.]

[22] J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, 2nd edition, July 2002. [1 citation on page 26.]

[23] The GNU Classpath project. http://classpath.org/. [1 citation on page 63.]

[24] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Prentice Hall PTR, 3rd edition, June 2005. [1 citation on page 78.]

[25] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154, Sept. 1981. [1 citation on page 145.]

[26] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. [1 citation on page 145.]

[27] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *USENIX OSDI '96*, pages 123–136, Oct. 1996. [2 citations on pages 15 and 28.]

[28] D. Grunwald and S. Ghiasi. Microarchitectural denial of service: insuring [sic] microarchitectural fairness. In *MICRO-35*, pages 409–418, Istanbul, Turkey, Nov. 2002. [1 citation on page 132.]

[29] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA 31*, pages 102–113, München, Germany, June 2004. [1 citation on page 146.]

[30] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*, pages 388–402, Anaheim, California, Oct. 2003. [5 citations on pages 17, 28, 39, 82, and 147.]

[31] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press. [1 citation on page 23.]

[32] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996. [4 citations on pages 41, 57, 60, and 61.]

[33] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, Jan. 1991. [1 citation on page 142.]

[34] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, Nov. 1993. [2 citations on pages 95 and 143.]

[35] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03*, pages 92–101, Boston, Massachusetts, July 2003. [4 citations on pages 15, 16, and 146.]

[36] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA 20*, pages 289–300, San Diego, California, May 1993. [11 citations on pages 15, 16, 28, 36, 39, 107, 112, 117, 145, and 147.]

[37] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88*, pages 276–290, Toronto, Ontario, Canada, Aug. 1988. [3 citations on pages 15, 141, and 142.]

[38] M. P. Herlihy. The transactional manifesto: Software engineering and non-blocking synchronization. Invited talk at PLDI, June 2005. http://research.ihost.com/pldi2005/manifesto.pldi.ppt. [2 citations on pages 21 and 23.]

[39] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, Providence, Rhode Island, May 2003. [2 citations on pages 15 and 142.]

[40] M. P. Herlihy and J. E. B. Moss. Transactional support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, Dec. 1992. [2 citations on pages 120 and 145.]

[41] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003. [1 citation on page 43.]

[42] IBM, Austin, Texas. *PowerPC Virtual Environment Architecture, Book II, Version 2.02*, Jan. 2005. [1 citation on page 42.]

[43] 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, LLNL, Livermore, California, Nov. 1987. [1 citation on page 145.]

[44] M. Jones. What really happened on Mars? http://research.microsoft.com/~mbj/Mars_Pathfinder/, Dec. 1997. [1 citation on page 15.]

[45] E. Jul and B. Steensgaard. Implementation of distributed objects in Emerald. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 130–132, Palo Alto, CA, Oct. 1991. IEEE. [2 citations on pages 29 and 148.]

[46] T. Knight. An architecture for mostly functional languages. In *LFP*, pages 105–112. ACM Press, 1986. [5 citations on pages 15, 112, 117, 120, and 145.]

[47] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *PODC '94*, pages 130–140, Los Angeles, CA, Aug. 1994. [1 citation on page 144.]

[48] L. Lamport. Concurrent reading and writing. *CACM*, 20(11):806–811, Nov. 1977. [2 citations on pages 15 and 141.]

[49] R. B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, Jan. 1989. [1 citation on page 119.]

[50] J. Manson and W. Pugh. Semantics of multithreaded Java. Technical Report UCMP-CS-4215, Department of Computer Science, University of Maryland, College Park, Jan. 2002. [2 citations on pages 34 and 147.]

[51] J. F. Martinez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X* [1], pages 18–29. [1 citation on page 145.]

[52] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, New York, NY 10027, June 1991. [4 citations on pages 15, 28, 137, and 142.]

[53] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *CACM*, 19(7):395–404, July 1976. [1 citation on page 117.]

[54] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, pages 267–276, Philadelphia, PA, May 1996. [1 citation on page 23.]

[55] C. Okasaki. Purely functional random-access lists. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 86–95. ACM Press, June 1995. [1 citation on page 100.]

[56] M. E. O'Neill and F. W. Burton. A new method for functional arrays. *J. Func. Prog.*, 7(5):487–514, Sept. 1997. [3 citations on pages 91, 100, and 102.]

[57] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34*, pages 294–305, Austin, Texas, Dec. 2001. [2 citations on pages 124 and 145.]

[58] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In Rajwar [1], pages 5–17. [2 citations on pages 15 and 145.]

[59] M. C. Rinard. Implicitly synchronized abstract data types: Data structures for modular parallel programming. *Journal of Programming Languages*, 6:1–35, 1998. [1 citation on page 22.]

[60] A. Rudys and D. 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. [1 citation on page 146.]

[61] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *SOSP '97*, pages 157–169, Oct. 1997. [1 citation on page 42.]

[62] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *CACM*, 10(8):501–506, Aug. 1967. [1 citation on page 24.]

[63] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*, pages 204–213, Ottawa, Ontario, Canada, Aug. 1995. [3 citations on pages 15, 28, and 146.]

[64] D. Siewiorek, G. Bell, and A. Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, 1982. [1 citation on page 145.]

[65] J. M. Stone, H. S. Stone, P. Heidelberg, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993. [2 citations on pages 15 and 145.]

[66] I. Sun Microsystems. Java native interface. http://java.sun.com/j2se/1.5.0/docs/guide/jni/, 2004. [3 citations on pages 69, 71, and 72.]

[67] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992. [1 citation on page 14.]

[68] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA '99*, pages 187–206, Denver, Nov. 1999. [1 citation on page 89.]

[69] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *MICRO-34*, Austin, Texas, Dec. 2001. [2 citations on pages 43 and 77.]

[70] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber. *The Java Tutorial: A Short Course on the Basics*. Prentice Hall PTR, 4th edition, Sept. 2006. [1 citation on page 81.]

[71] L. Zhang. UVSIM reference manual. Technical Report UUCS-03-011, University of Utah, Mar. 2003. [1 citation on page 120.]

[72] Y. Zibin, A. Potanin, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. Technical Report MIT-CSAIL-TR-2007-018, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, March 16, 2007. [1 citation on page 89.]

[73] C. Zilles and D. H. Flint. Challenges to providing performance isolation in transactional memories. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Madison, Wisconsin, June 2005. [2 citations on pages 131 and 132.]

# Index