# LANGUAGE SUPPORT AND COMPILER OPTIMIZATIONS FOR OBJECT-BASED SOFTWARE TRANSACTIONAL MEMORY


BY

GUY EDDON

B.A., THOMAS EDISON STATE COLLEGE, 2002

Sc.M., BROWN UNIVERSITY, 2004


A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN THE DEPARTMENT OF COMPUTER SCIENCE AT BROWN UNIVERSITY


PROVIDENCE, RHODE ISLAND

MAY 2008

© Copyright 2008 by Guy Eddon

This dissertation by Guy Eddon is accepted in its present form

by the Department of Computer Science as satisfying the

dissertation requirement for the degree of Doctor of Philosophy.

Date ___________________ ________________________________________

Maurice Herlihy, Advisor

Recommended to the Graduate Council

Date ___________________ ________________________________________

Ali-Reza Adl-Tabatabai, Reader

Date ___________________ ________________________________________

John Jannotti, Reader

Approved by the Graduate Council

Date ___________________ ________________________________________

Sheila Bonde, Dean of the Graduate School

# Abstract

Modern multicore processors, such as Intel's Core 2 Duo, bring urgency to parallel programming research, which, despite several decades of work, has not changed much since the advent of mutual exclusion in the 1960s. Software Transactional Memory (STM) is a promising alternative that borrows heavily from the database community to propose a transactional approach to program state. By optimistically assuming that most computations executed in parallel do not conflict and then detecting and terminating those few computations that do, STM provides the basis for new programming language abstractions that permit programmers to think linearly while the code they write is safely executed in parallel.

The difficulty with today's STM libraries, however, is their generally poor performance and convoluted programming model. In order to make STMs more popular, accessible, and functional, this thesis argues that three things must happen: first, a streamlined programming model can hide all complex details of the STM library from the programmer; second, static analysis can optimize the client code's interaction with the underlying STM; third, the STM library itself should be decomposed and optimized for use by an optimizing compiler. Together, we show that these optimizations achieve an order of magnitude performance improvement.

As part of this work, we develop four nonblocking transaction synchronization and validation algorithms designed for use in STM systems, all of which support a nonblocking progress condition called obstruction-freedom, provide always consistent reads, and integrate orthogonal contention management. Finally, we propose the first language extensions and compiler support for transactional boosting, a methodology for transforming highly concurrent linearizable objects into highly concurrent transactional objects. Based on these results, we conclude that appropriate language support and high quality compiler optimizations are necessary for the success of any STM system.

# Acknowledgments

I gratefully acknowledge the support of my advisor, Maurice Herlihy, whose guidance ensured that my last few years at Brown were an extraordinary opportunity for growth and development. I also owe thanks to Ali-Reza Adl-Tabatabai of Intel's Programming Systems Lab; the summer I spent at Intel helped ground this work. Professors Krishnamurthi and Reiss kindly provided helpful comments on several drafts of the thesis. Finally, a few fellow graduate students made the Brown years more enjoyable: Toke Lindegaard Knudsen (with whom I studied ancient models of planetary motion), Nikos Triandopoulos (who helped me survive a semester of Hellenic Greek) and Frank Wood. And with gratitude to Raluca for showing me that this was possible.

# Contents

**Figures** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **viii**

**1. Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

1.1. Background

1.2. Thesis statement

1.3. Outline of the thesis

**2. Background.** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **8**

2.1. Concurrency

2.2. Nonblocking algorithms

2.3. Low-level synchronization primitives

2.4. Hardware transactional memory

2.5. Software transactional memory

2.6. Transactions in programming languages

3. **Approach** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **24**

3.1. Language support

3.2. The transaction model

3.3. The atomic attribute

3.4. Atomic blocks

3.5. Definition of atomic types, methods and blocks

3.6. Limitations and restrictions

3.7. Retry and conditional execution of atomic blocks

3.8. Transactional boosting

**4. Optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47**

4.1. Properties

4.2. Whole object dataflow analysis

4.3. Blocking optimizations

4.4. Obstruction-free optimizations

4.5. Retry semantics

4.6. The CASe primitive

4.7. Array algorithms

**5. Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94**

5.1. Properties

5.2. The VisibleReaders algorithm

5.3. The WarningWord algorithm

5.4. The BloomFilter algorithm

5.5. The WriterBins algorithm

5.6. Comparisons

**6. Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126**

**7. Conclusions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133**

7.1 Future work and open questions

**8. Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136**

**9. Appendix . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149**

# Figures


1.1 Increasing processor transistor counts 2

1.2 Processor clock speeds plateau 3

2.1 CAS and the ABA problem 12

3.1 Throughput for a transactionally-boosted skip list 46

4.1 Sample control flow graph for a simple function 58

4.2 Blocking mode results 81

4.3 Obstruction-free results 84

4.4 STAMP benchmark performance 85

4.5 Performance comparison of atomic array algorithms 93

5.1 Comparison of several TM algorithms 95

5.2 The transactional metadata model 98

5.3 Two transactions concurrently prune the reader list 100

5.4 Nonlinearizable transactions 121

5.5 Comparison of the algorithms 122

5.6 Support for contention management 123

5.7 Performance comparison with low contention 125

5.8 List benchmark at increasing levels of contention 125

# Chapter 1

# Introduction

## 1.1 Background

In 1965, Dr. Gordon Moore famously predicted that the number of transistors placed on an integrated circuit would double approximately every 24 months. Such exponential growth in transistor counts translates into roughly equivalent improvements in clock speed and microprocessor power over the period. Although technically still holding true, as transistor counts have continued to increase at the rate predicted by Moore's Law (see Figure 1.1), processor clock speeds have not kept pace for a number of reasons, including heat dissipation and difficulty further reducing the 65 nanometer technology used in current processor manufacturing. Clock speeds have hovered around

two gigahertz since about 2000, forcing chip manufacturers to look to other avenues for achieving future gains in hardware processing power.

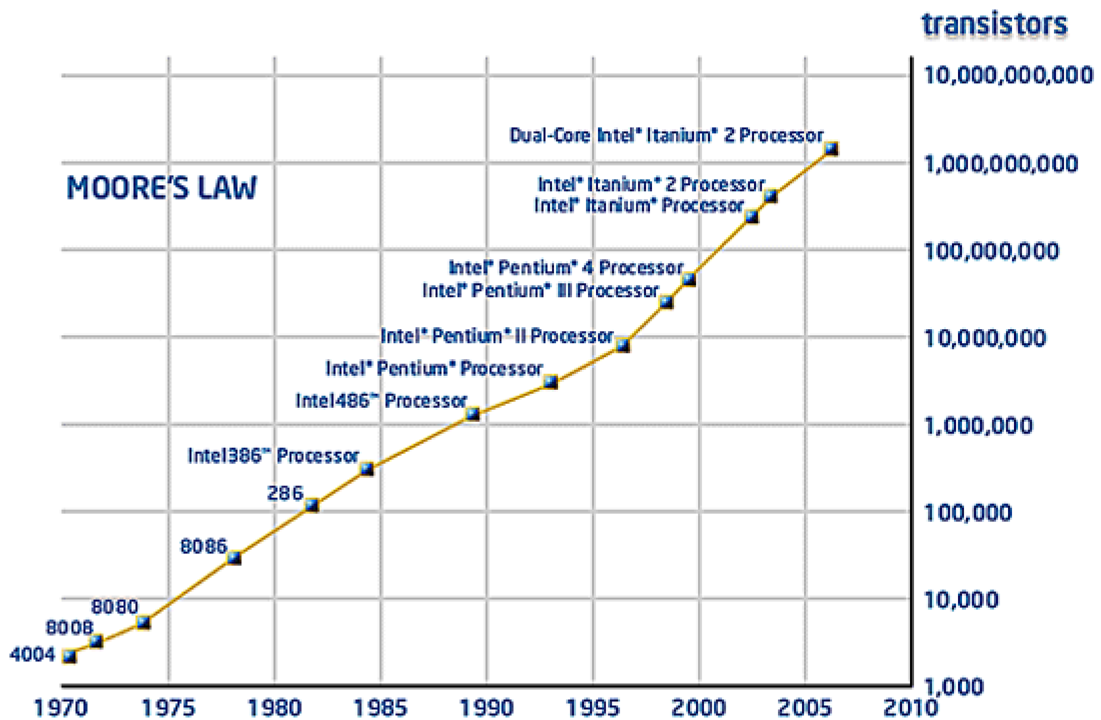



Figure 1.1: Increasing processor transistor counts[1]

Instead of deriving increased processing power from scaling clock speed (see Figure 1.2), which eventually increases power usage and heat emission to unmanageable levels, chip manufacturers have begun to increase overall processing power by adding additional CPUs, or "cores" to the microprocessor package. In order to take advantage of this new type of computer, however, software must be parallelizable. Unfortunately, standard techniques for writing multithreaded code haven't changed very much since the advent of mutual exclusion in the 1960s. Clearly, multicore processors demand language

[1] Source: Intel web site. See http://www.intel.com/technology/mooreslaw.

abstractions and tools designed to help programmers write software that can take advantage of their power safely and efficiently.

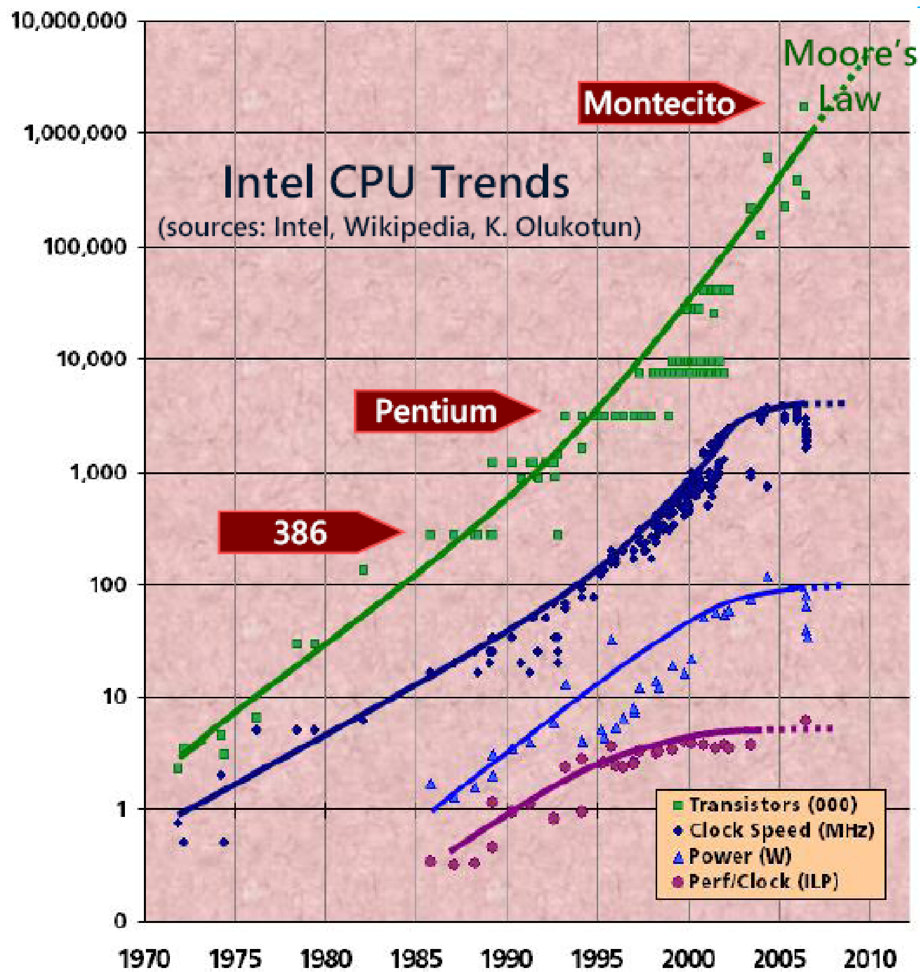



Figure 1.2: Processor clock speeds plateau[2]

Transactional memory is one promising alternative to lock-based multiprogramming. This dissertation proposes language support and compiler optimizations for Software Transactional Memory (STM) systems in order to make STM more accessible and efficient than is possible in the context of a purely library-based approach. Currently, STM libraries typically have cumbersome language interfaces that

[2] Source: Herb Sutter, Software and the Concurrency Revolution.

require programmers to write code in particular ways (e.g., implementing interfaces, defining fields as properties, using delegates to start transactions, coding a retry loop, etc.). Furthermore, in the form of libraries, STM implementations have little information about the client algorithms that use their services. As a result, STM libraries must make very conservative assumptions about client code—an aspect which negatively affects their performance.

An STM compiler named Peet, developed as part of this project, attempts to address both of the shortcomings described above. In the area of language integration, the compiler uses annotations that indicate atomic types, methods, and blocks. Standard keywords, such as *const*, *readonly*, *sealed*, and *static*, provide additional contextual information to the compiler, and the *using* keyword is overloaded to allow for dynamic and conditional transactional execution. In order to provide improved performance, the compiler performs a dataflow analysis to determine whether an atomic object is accessed in “read” or “write” mode and whether a specific atomic field access is guaranteed to occur subsequent to a previous use of that object. On the basis of this information, the compiler can make more efficient use of the STM library, often inlining the transactional machinery directly in the user’s code in the form of several bytecode instructions.

Peet supports a friendly programming interface, and, to date, offers a performance improvement of between four and ten times compared with the same algorithms implemented solely with an STM library. This performance improvement helps close the gap between STM systems and conventionally synchronized concurrent algorithms. In addition, the research on STM compiler optimizations has revealed several new aspects of STM systems, including the fact that subsequent reads and writes of atomic fields can

be made entirely lock-free, even in the context of an STM model that uses short critical sections for initial reads and writes.

Finally, we also propose novel language extensions to support transactional boosting, a powerful new technique for transforming existing linearizable objects into transactional objects, thus permitting highly concurrent objects such as those found in the java.util.concurrent package to participate in STM transactions. When applied in conjunction with compiler support, we show that transactional boosting is both a flexible and natural way to escape the standard transactional model, and thus offers a promising alternative to existing "expert" approaches, such as open nesting and early release. Based on our results, we conclude that appropriate language support and high quality compiler optimizations are necessary parts of any STM system.

## 1.2 Thesis statement

The difficulty with today's STM libraries is their generally poor performance and convoluted programming model. In order to make STMs more popular, accessible, and functional, we argue that three things must happen: first, we streamline the programming model so that all complex details of the STM library are hidden from the programmer; second, we use static analysis to optimize the client code's use of the underlying STM; third, we decompose and optimize the STM library itself for use by an optimizing compiler. Together, we show that these optimizations achieve an order of magnitude performance improvement.

There is currently much debate about whether STMs that support constructive theoretic properties such as lock freedom, always consistent reads, and orthogonal

contention management can be made to perform as well as STMs that jettison these properties in pursuit of lower overheads. Some [2, 23] have argued that STMs should not be wait-free or even obstruction-free because of the performance overhead that early non-blocking STM systems suffered [48]. Other systems [1] permit inconsistent reads in the underlying STM, which violate the isolated property of transactions, and then try to compensate for that problem in the compiler. We show that with the right combination of language design, compiler support, and library optimization, these important properties can be supported efficiently.

STM systems are often compared with the development of garbage collection algorithms and virtual memory systems. In both cases, initially there was a great deal of concern that the overhead of such approaches would render them too unwieldy for practical use, and so much effort was spent on offering ways to escape the constraints of these models where performance was the top priority. In both cases, however, once quality hardware support (for virtual memory) or runtime/compiler support (for garbage collection) was developed, the need for most programmers to escape the constraints of these models dissipated. The problem that this dissertation aims to address, then, is to help provide the missing language, compiler, and library support that we believe will be essential in making STMs as ubiquitous in mainstream programming languages as garbage collection is today.

## 1.3 Outline of the thesis

Chapter Two reviews the development of concurrent programming, including locks, non-blocking algorithms, and modern STM systems. Chapter Three describes the

programming language extensions supported by Peet, and Chapter Four covers the library and compiler optimizations that enable the significant performance gains. Chapter Five introduces four obstruction-free transaction synchronization and validation algorithms designed for object-based STMs. Finally, Chapter Six evaluates the compiler and Chapter Seven summarizes the contributions of this thesis.

# Chapter 2

# Background

## 2.1 Concurrency

Concurrent programming, or programming with multiple threads of execution, has remained largely unchanged since time sharing systems first emerged in the late 1960s. Now, as then, mutual exclusion algorithms (a.k.a. locks) prevent the concurrent use of shared data, and thus remain among the fundamental building blocks of concurrent programs [83].

Synchronization primitives such as critical sections, mutexes, and semaphores share a common characteristic: when a desired lock is unavailable, they block. Blocking, however, is undesirable, because it means that some threads will waste time waiting for

other threads to release their locks. Furthermore, in certain cases, it can also produce undesirable side effects, such as deadlock, livelock, and priority inversion [40]. Equally challenging, lock-based synchronization requires programmers to choose between locking small bits of code (fine-grained locking) or larger sections, such as entire methods (coarse-grained locking). While fine-grained locking can lead to improved parallelism, it is more difficult to reason about and can therefore lead to subtle race conditions; coarse-grained locking, by contrast, is easier to implement and often safer, but typically exhibits poor parallelism.

Section 2.2 describes important properties of nonblocking algorithms, which build on the low-level synchronization primitives outlines in section 2.3. Section 2.4 introduces hardware transactional memory. Section 2.5 describes the subsequent development of software transactional memory, originally designed as a stopgap measure until the arrival of hardware support. Finally, section 2.6 explores transactional programming extensions available in existing programming languages.

## 2.2 Nonblocking algorithms

For the reasons outlined above, in the last decade a great deal of research has focused on a class of concurrent algorithms called nonblocking algorithms [42]. These types of algorithms permit multiple threads to read and write shared data concurrently without corrupting it. Indeed, nonblocking algorithms permit concurrent access to shared data without using any form of locking or mutual exclusion, and are thus considered an essential component for the development of scalable software systems. Such algorithms are generally categorized as either wait-free, guaranteeing that all threads can complete a

given operation in a finite number of steps, or lock-free, meaning that some thread can complete an operation in a finite number of steps. An algorithm can be lock-free without being wait-free.

Algorithms that do not employ locks are not subject to problems such as deadlock or priority inversion, and therefore, in theory, should be easier to employ. Also, since no time is wasted waiting for a lock to become available, non-blocking algorithms can be expected to perform better than their lock-based counterparts. However, the challenge for non-blocking algorithms is to enable multiple threads of execution to concurrently share access to the same data—a problem that, as it turns out, is far more difficult to contend with than deadlock and priority inversion. While concurrent read access to shared memory is relatively easy to achieve, the problem of how to enable concurrent write access to shared memory is much more difficult to solve. Truly concurrent write access to shared memory isn't possible without special hardware support, but its effect can be simulated through the use of low-level synchronization primitives that test the value stored in a memory location before changing it (see § 2.3).

Thus, in brief, while nonblocking algorithms are, in principle, far more appealing than their more conventional blocking counterparts, in practice they have been somewhat disappointing on account of two primary drawbacks. First, their performance often does not compare favorably with highly optimized blocking versions of the same algorithms [40]. Second, they have proven very difficult to write compared with equivalent lock-based algorithms [46]; in part, this is why researchers have thus far generally focused on relatively simple nonblocking algorithm such as lists, queues, stacks, and trees. (There has been some research on automatically converting sequential algorithms into

nonblocking concurrent algorithms, but it has seldom produced algorithms that yield acceptable performance results [40].)

## 2.3 Low-level synchronization primitives

Nonblocking algorithms are difficult to write because they are generally built from very low-level atomic primitives provided by hardware, such as compare-and-swap (CAS) or load linked/store conditional (LL/SC). Both CAS and LL/SC enable a thread to read a value from memory, modify the value, and then later atomically save the new value back to its original location, if and only if no other thread modified the value stored at that location in the interim. A CAS operation thus requires three inputs: a memory address, the value expected to be found at that location, and a new value that should be stored at the same address. If the address specified contains the expected value, the CAS operation atomically swaps the original value with the new value and returns successfully. If not, the value stored remains unchanged and the CAS is considered to have failed. Although CAS operates atomically, under no circumstances does it block. The Java code below shows the basic semantics of a CAS operation:

```
public class register {
    private Object value;     // the actual value

    public synchronized boolean CAS(Object oldV,
                                    Object newV) {
        if(value != oldV)
            return false;
        value = newV;
        return true;
    }
}
```

Thus, the CAS operation may be thought of as a very primitive transaction. It operates atomically, such that all of its changes become visible to other threads at one instant known as the linearization point. Its work is also isolated, in the sense that, should it fail, no intermediate changes will ever have been visible to any other threads.

The CAS operation is, however, subject to the ABA problem, whereby CAS may falsely succeed even if the underlying value has changed (see Figure 2.1). This problem manifests itself when one thread issues two or more CAS operations that change the value stored at a given memory location from A to B, and then eventually back to A again. In this case, a second thread might have read the original value A prior to the first thread's mutations and then later attempted to use CAS in order to change A to another value C. Thus, as a consequence of the ABA problem, data may easily become corrupted, since the guarantee provided by CAS, namely that when CAS succeeds the value hasn't changed since the last read, is broken. (The presence of automatic garbage collection eliminates some instances of the ABA problem, but a number of others remain [48].)

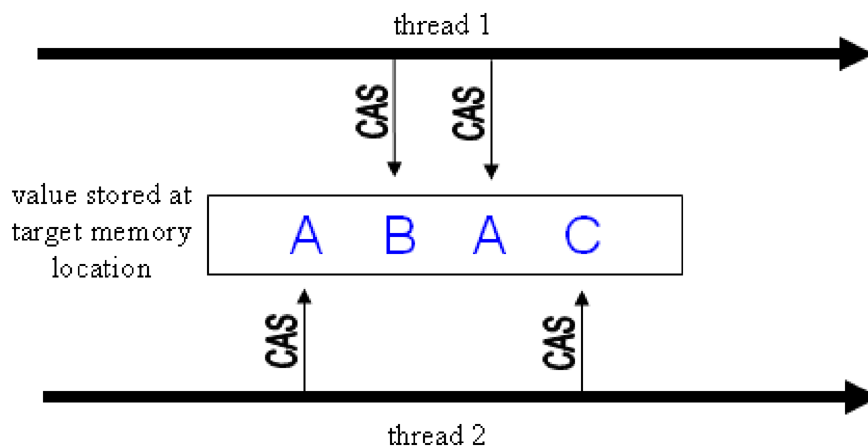



Figure 2.1: CAS and the ABA problem.

In contrast to CAS, LL/SC provides a more straightforward answer to the ABA problem, since the store operation is guaranteed not to succeed if any mutations to the target memory location have occurred since the last load.

In addition to CAS and LL/SC, which have been proven to represent fundamentally equivalent universal synchronization primitives [43], other slightly more powerful nonblocking synchronization primitives include double-compare-and-swap (DCAS[1]), which atomically compares and swaps two words [28], and its variants: double-compare-single-swap (DCSS) [id.], k-compare-single-swap (KCSS) [60], and multi-word compare-and-swap (MCAS) [36]. In sum, however, more powerful versions of the CAS and LL/SC primitives generally do not offer a sufficient answer to the difficulties inherent in nonblocking algorithm design [19].

The enqueue and dequeue algorithms shown below are from the nonblocking concurrent queue algorithm given by Michael and Scott in [65]. They are representative of the complexity and fragility of nonblocking algorithms built using solely CAS.

```
enqueue(Q: pointer to queue t, value: data type)
E1: node = new node()         # Allocate a new node from the free list
E2: node->value = value       # Copy enqueued value into node
E3: node->next.ptr = NULL     # Set next pointer of node to NULL
E4: loop                      # Keep trying until Enqueue is done
E5:    tail = Q->Tail         # Read Tail.ptr and Tail.count together
E6:    next = tail.ptr->next  # Read next ptr and count fields together
E7:    if tail == Q->Tail     # Are tail and next consistent?
E8:       if next.ptr == NULL # Was Tail pointing to the last node?
E9:          # Try to link node at the end of the linked list
E10:         if CAS(&tail.ptr->next, next, <node, next.count+1>)
E11:            break         # Enqueue is done. Exit loop
E12:         endif
E13:      else                # Tail was not pointing to the last node
E14:         # Try to swing Tail to the next node
E15:         CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
E16:      endif
E17:   endif
E18: endloop
```

[1] A hardware implementation of DCAS was provided by the Motorola 68040, although its performance was generally quite poor.

```
E19: # Enqueue is done. Try to swing Tail to the inserted node
E20: CAS(&Q->Tail, tail, <node, tail.count+1>)


dequeue(Q: pointer to queue t, pvalue: pointer to data type): boolean
D1: loop                          # Keep trying until Dequeue is done
D2:    head = Q->Head             # Read Head
D3:    tail = Q->Tail             # Read Tail
D4:    next = head->next          # Read Head.ptr->next
D5:    if head == Q->Head         # Are head, tail, and next consistent?
D6:       # Is queue empty or Tail falling behind?
D7:       if head.ptr == tail.ptr
D8:          if next.ptr == NULL # Is queue empty?
D9:             return FALSE      # Queue is empty, couldn't dequeue
D10:         endif
D11:         # Tail is falling behind. Try to advance it
D12:         CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)
D13:      else                    # No need to deal with Tail
D14:         # Read value before CAS, otherwise another dequeue might
D15:         # free the next node
D16:         *pvalue = next.ptr->value
D17:         # Try to swing Head to the next node
D18:         if CAS(&Q->Head, head, <next.ptr, head.count+1>)
D19:            break             # Dequeue is done. Exit loop
D20:         endif
D21:      endif
D22:   endif
D23: endloop
D24: free(head.ptr)          # It is safe now to free the old dummy node
D25: return TRUE             # Queue was not empty, dequeue succeeded
```

## 2.4 Hardware transactional memory

In light of the fact that none of the synchronization primitives described above adequately address the difficulties of nonblocking algorithm design, the question arises as to whether a higher-level abstraction might be necessary to address their weaknesses and to facilitate concurrent programming. Transactional memory [48] represents a significant step in this direction. It builds on the abstraction of a transaction to provide hardware instructions that atomically modify the value of an arbitrary number of memory locations. In transactional memory systems, if the transaction succeeds, all updated values appear to change at one time; if it fails, no changes are ever recorded.

Most hardware-based transactional memory systems utilize processor caches to store tentative changes, as well as the cache coherence protocol to advertise committed transactions to all other processors [77]. However, such approaches are limited, to the extent that a transaction can only modify as many values as will fit into a processor's cache. (To address this challenge, some researchers have proposed protocols that spill over into virtual memory after a processor's main cache has been filled.) Furthermore, because hardware-based transactional memory requires hardware modifications (new instructions, a second set of processor caches, as well as extra storage for cache lines involved in an active transaction) to implement successfully, it has yet to be supported by any commercially available system.

Within this context, it is instructive to compare transactional memory to Oklahoma update [82], an alternative hardware-based memory update operation that supports the updating of multiple memory locations atomically in order to simplify the development of nonblocking algorithms. Closely modeled on the LL/SC approach, in Oklahoma update a thread reserves access to a memory location and can later update that location if and only if no other thread has modified its value in the interim. Unlike LL/SC, however, which supports the atomic update of only a single memory location, an Oklahoma update can encompass a number of memory locations. The primary difference between transactional memory and Oklahoma update consists in divergent underlying beliefs with respect to the maximum number of independent memory locations that need to be controlled atomically. Oklahoma update conjectures that the maximum should be a number of three or perhaps four independent memory locations, placing it solidly in the camp of nonblocking multiprocessor synchronization primitives that are slightly more

powerful than CAS and LL/SC. Transactional memory, by contrast, proposes to support transactions of between ten to one hundred independent memory locations, a qualitative as well as a quantitative difference.

## 2.5 Software transactional memory

First developed as a stopgap measure until the arrival of hardware support, software transactional memory (STM) is a programming interface that provides a software implementation of the ideas first developed in the context of hardware-based transactional memory. The advantage of this approach is that it allows researchers to experiment with transactional memory protocols without waiting for these ideas to be implemented in hardware. The software approach has proven so compelling, however, that is raises the question of whether complete hardware support is necessary or even desirable. It is likely that some very low-level hardware support will be desirable to aid in the implementation of efficient software transactional memory systems; indeed, we make one such proposal in § 4.6.

The first STM proposal was static [77]: it required the programmer to specify, in advance, what data would be accessed as part of the transaction. Later, however, Dynamic STM (DSTM) [48] generalized these ideas such that the system does not a priori require the user to specify which objects will be accessed from a transaction.

The *insert* method shown below is a fragment from a nonblocking singly-linked integer list implemented using DSTM. There are several aspects worth noting about this code:

1. Classes that will be used in transactions need to support cloning through a custom TMCloneable interface.
2. A target object that will be participating in a transaction must be wrapped by a proxy class called TMObject.
3. The body of the code resides in a while loop, which is required in order to retry the transaction in case of failure.
4. An explicit call is necessary to begin the transaction.
5. Before accessing a transactional object, the programmer must enlist it in the transaction by being opened in read or read/write mode.
6. The transaction must explicitly end with an attempt to commit its work; if commit fails for any reason, such as a concurrency violation, the transaction is aborted and control flow returns to the while loop in step 3 to retry.

The insert method itself is essentially not transactional: while transactions begin and end by invoking methods of the current thread, only the set of objects specifically wrapped by a TMObject proxy, and then opened for reading and/or writing, actively participate in the transaction.

```
public boolean insert(int v) {
    // (1) List must implement TMCloneable
    List newList = new List(v);

    // (2) need to wrap the target object in a TMObject
    TMObject newNode = new TMObject(newList);

    TMThread thread = (TMThread)Thread.currentThread();

    // (3) loop to retry the transaction in case it fails
    while(true) {

        // (4) start the transaction
        thread.beginTransaction();
```

```
        boolean result = true;
        try {

            // (5) open the objects in read/write mode
            List prevList =
                (List)first.open(TMObject.WRITE);
            List currList =
                (List)prevList.next.open(TMObject.WRITE);

            while(currList.value < v) {
                prevList = currList;
                currList =
                    (List)currList.next.open(
                                               TMObject.WRITE);
            }

            if(currList.value == v) {
                result = false;
            }
            else {
                result = true;
                newList.next = prevList.next;
                prevList.next = newNode;
            }
        }
        catch(Denied d) { }

        // (6) attempt to commit the transaction
        if(thread.commitTransaction())
            return result;
    }
    return false;
}
```

The example above suggests that DSTM code is far easier to understand than a similar nonblocking algorithm based on CAS. However, the programming paradigm forced on the programmer by this model is still rather awkward and requires a solid grasp of the underlying concurrency issues. DSTM makes an effort to address this shortcoming by separating concurrency issues from the main programming model through the use of customizable contention managers, which are responsible for setting policy followed

when two or more transactions conflict through contention for the same resources (some contention managers might abort all transactions in their path, potentially leading to live-lock, while others might use a more "polite" scheme such as exponential back-off before aborting another transaction).

In contrast to DSTM, SXM [41] offers a newer STM package designed to support C# and the .NET Framework with a friendlier programming interface. The advantage of this approach is that, by using the .NET reflection package to transparently rewrite atomic classes, SXM is able to hide much of the boilerplate code required to manage transactions in DSTM. Transactions are started through a delegate that begins the transaction and contains a retry loop. The downside of this approach, however, is that all transactional methods must have an identical signature, which means that all parameters much be passed through an array of objects. In addition, in order to successfully intercept the reads and writes on fields of atomic objects, all the fields must be implemented as public virtual properties with appropriate get and set methods. Below is the *Insert* method as implemented with SXM:

```
                                        // array of objects
public override object Insert(params object[] _v) {
    // transaction creation is handled by the XStart
    //    delegate
    int v = (int)_v[0];

    // instantiation not done with new -
    //    special factory creator method
    Node newNode = (Node)factory.Create(v);

    Neighborhood hood = Find(v);
    if(hood.currNode != null)
        return false;

    Node prevNode = hood.prevNode;

    // transparent atomic field access
```

```
    newNode.Next = prevNode.Next;

    prevNode.Next = newNode;
    return true;
}
```

An additional benefit of the SXM library is that, in addition to an obstruction-free non-blocking mode (called OFree), it also offers a model that uses very short critical sections during field reads and writes (called TMem)[2]. The TMem approach is somewhat less elegant than the obstruction-free mode, but it yields superior performance. In this model, rather than cloning objects when they are opened for writing, SXM transparently introduces a shadow field for each field of an atomic object. When an object is opened for writing, a Backup method is called that copies the value of each "real" field to its shadow field. If a transaction is aborted, a corresponding Restore method copies the saved values back from the shadow fields to the real fields. One reason that the TMem model performs better than the OFree approach is that a heap allocation (for the object clone) is not required during the Backup/Restore operations. This means less work for the garbage collector. Also, because shadow fields are part of the original object, and are allocated as part of the same heap allocation that creates that object, it is far more likely that the object's real and shadow fields will be found in the same cache line during a Backup/Restore operation than it is that the object and its clone will be in the same cache line.

[2] Note that both the TMem and OFree factories in SXM are implemented directly in bytecode. A newer version under development, SXM2, allows factories to be defined in source code rather than bytecode.

## 2.6 Transactions in programming languages

To address the weaknesses of library-based approaches to STM, there have been several attempts to integrate transactions directly into programming languages. An early approach developed at MIT in the 1980s was called Argus [59]. It supported distributed programming through dynamically created guardians and atomic actions, much like transactions. Argus, however, supported distributed transactions with a two-phase commit protocol [57] and used the guardian abstraction to model programs that could even survive hardware failures. These far-reaching goals resulted in a language that, while impressive in its scope, was too inefficient to be applied to systems that did not benefit from its expensive machinery.

More recently, researchers at Cambridge have pursued a lighter-weight approach in the design of an atomic keyword for Java [34]. Their work builds on the idea of conditional critical regions (CCRs), first proposed for concurrency control by Anthony Hoare in 1972 [54]. This atomic keyword supports a boolean guard condition that causes calling threads to block until the condition is satisfied. Instead of implementing this guard through mutexes, the atomic keyword in Java builds the CCR concept on top of a word-sized STM system that supports obstruction freedom. Thus, for example, the get method shown below is from a class that implements a shared buffer using the atomic keyword with a guard condition. Here, the guard condition is used to guarantee that the buffer is not empty when attempting a get. When the guard condition is satisfied, the atomic block ensures that both the *items* count is decremented and the last value is returned atomically:

```
public int get() {
    atomic(items != 0) {
        items--;
        return buffer[items];
    }
}
```

The release of Intel's C++ STM Compiler Prototype[3] is one recent area of language work in the STM space. Intel's STM compiler support is integrated with high-level C++ language extensions. In Intel's proposed language extension for atomic code, atomic blocks are denoted with the __tm_atomic keyword, as shown below:

```
__tm_atomic {
     // code block of statement(s)
}
```

While reads and writes made from an atomic block appear atomic, isolated, and consistent with respect to all other atomic blocks in the program, there is no such guarantee for non-atomic regions. Closed nesting, where the effects of nested transactions are visible only when the outermost transaction commits, is also supported.

Intel's approach is to clone transactional methods marked with the __declspec(tm_callable) annotation. Within the cloned transactional version of the method, every memory read and write is transformed into a transactional read/write with the support of the underlying STM library. Whenever the transactional method is called from within an atomic block (__tm_atomic), the transactional version of the method is invoked; when called from outside a transaction the original version of the method is used. For indirect calls invoked through a function pointer, Intel's compiler adds a runtime check to determine whether the caller is calling from an atomic block and

[3] Intel C++ STM Compiler Prototype Edition 2.0: Language Extensions and Users' Guide, Document #318253-001US, revision 2.0.

dynamic dispatch code to invoke either the instrumented version or the original function, as appropriate.

Where atomic blocks call methods not marked as tm_callable, such legacy code is serialized and is irrevocable (i.e., its execution cannot be undone). Such irrevocable operations are guaranteed to commit. However, in order to switch to irrevocable mode from the regular transactional mode, the current speculative transaction must be aborted and then the entire transaction re-executed in the irrevocable mode.

Methods known to the programmer to be safe for transactional as well as non-transactional use may be marked as __declspec(tm_pure). Transactional function clones are not created for tm_pure methods. However, such methods should not access, even for read, any static or non-transaction local memory; otherwise its behavior is undefined.

One interesting aspect of Intel's language approach is the class-level tm_callable annotation. When applied to a C++ class, this annotation makes all member functions, both virtual and non-virtual, assume the tm_callable attribute. While on the one hand this is simply syntactic sugar that saves the programmer from marking each method individually, it is also a first step towards using abstract data types as the basic atomic building blocks of a program. The class-level tm_callable attribute is even inherited by derived classes in Intel's model.

Intel's language proposal also includes an explicit abort operation via the __tm_abort statement, which causes a control flow change: execution continues with the next lexical statement following the innermost transaction's scope. This is similar to the *Retry* operation proposed by some STM systems, although it does not automatically retry the current transaction.

# Chapter 3

# Approach

## 3.1 Language support

The Peet compiler introduces special support for transactional programming in order to help alleviate the difficulties of programming with an STM library described in Chapter 2. By supporting atomic types directly in the language and transparently rewriting field accesses for those objects, the compiler hides most of the transformations required to convert a sequential algorithm to one that uses STM. This chapter describes the programming language interface supported by Peet and shows how the code generation pass handles the various language elements. Chapter Four discusses the STM-

specific optimizations supported by the compiler, as well as modifications to the SXM library necessary to support Peet's functionality.

Technically, Peet is implemented as a post-compilation pass that rewrites binary code[1], and it therefore does not have the opportunity to process new keywords in the source code. Instead of first-class keywords, Peet uses annotations to describe atomic types and methods and then inserts calls to an optimized version of the SXM library in order to implement the transactional memory semantics. Annotations are an attractive way to prototype language features because they are easily extensible, familiar to programmers, and do not require introducing new syntactic structures. Another advantage of this approach is that it allows Peet to support bytecode generated by Java, C# or Visual Basic (i.e., any front-end compiler that produces standard bytecodes is supported).

Section 3.2 gives the correctness properties of our transaction model. Section 3.3 introduces the atomic attribute, which is used to annotate atomic types and methods; atomic blocks are covered in Section 3.4. A comprehensive definition of atomic types, methods, and blocks is provided by Section 3.5 and Section 3.6 covers the limitations and restrictions on atomic code in C# and Java. Retry semantics and the conditional execution of atomic blocks are described in Section 3.7, and our proposed language extensions for transactional boosting are detailed in Section 3.8.

## 3.2 The transaction model

The concurrency model supported by the compiler and underlying STM is known as linearizability [44], where each committing transaction appears to take effect

[1] Peet is implemented as a binary rewriter executed subsequently to the compiler. The Phoenix compiler framework developed by Microsoft Research is used for rewriting .NET programs; the JikesBT library is used to rewrite Java programs.

atomically at the linearization point. Every transaction is atomic, isolated and consistent [86]. For a formal definition of what a memory transaction should be, see Guerraoui and Kapalka [30] who extend linearizability, as defined by Herlihy and Wing [43, 44], to the transactional memory sphere. Guerraoui and Kapalka define a safety property known as opacity that captures the intuition that (1) all operations performed by all committing transactions appear to take effect at a single, indivisible point (the linearization point); (2) no operation performed by any aborting transaction is ever visible to other transactions (including live ones); (3) the system preserves real-time order; and (4) every transaction always observes a consistent state. Our STM model ensures opacity.

The progress condition of obstruction freedom [44] guarantees that some thread will make progress. However, we make no guarantees about fairness or liveness [58]; these concerns are the responsibility of the contention management policy [76]. Finally, our STM model does not support interactions between transactional and non-transactional threads; such conflicts are detected by all code (including dynamically loaded code) processed by the compiler, and exceptions are thrown. This model is generally referred to in the literature as weak atomicity [6], although our model is safe in the sense that interactions between transactional and non-transactional threads are detected. Finally, nested transactions are not supported (see § 3.8 for a proposed alternative).

## 3.3 The atomic attribute

The basic annotation is the Atomic attribute, which can be used on both types and methods. When applied to types, the Atomic attribute indicates that the fields of that type

should be accessed from within a transaction. The code fragment below shows an atomic Node class from the List benchmark:

```
[Atomic]
public class Node { // List element
    public int value; // key for list element
    public Node next; // reference to next list element
}
```

The fields of atomic classes may be *public*, *internal*, *protected*, *private*, *static*, *const*, or *readonly*. The only restriction is that the field types must either be value types, or, in the case of reference types, the reference types must themselves be marked as atomic. Certain special pre-existing objects that are known to be safe for transactional use but are not marked with the Atomic attribute are permitted when the field reference is tagged with the TxSafe attribute, which informs Peet that the field is in fact safe for transactional use[2]. This is a type of programmer override of the restrictions of transactional programming, but should be used only where the programmer is certain the object is thread-safe.

Warning messages are issued where non-atomic data types are accessed from transactional methods. Error messages are also issued where atomic data types are accessed from non-transactional methods. Unfortunately, however, because Peet is a post-compiler, it may not see dynamically loaded code in time to catch such errors; they are caught at runtime and exceptions are thrown.

The code fragment below shows a portion of the SkipList class. The Random type, defined by the .NET Framework, is not atomic. However, because the programmer

[2] The TxSafe attribute can be applied to classes, methods, or fields that are considered safe for use is transactions. TxSafe methods are permitted to access atomic fields without an active transaction, which may be unsafe.

knows that it is safe for use in a transaction (i.e., it is thread-safe), the TxSafe attribute is used to indicate this fact. The IntSetBenchmark base class is also an atomic class; inheritance of atomic types is covered in § 3.6.3.

```
[Atomic] public class SkipList : IntSetBenchmark {
    private const int MaxLevel = 32; // Maximum level


    // Probability factor
    private const double Probability = 0.5;

    // The skip list header
    private readonly Node header;

    // Random number generator
    [TxSafe] // safe for transactional use
    private Random random = new Random();
```

Methods and properties[3] can also be annotated with the Atomic attribute to indicate that the entire method runs in a transaction. Atomic methods, by default, do not require or create a transaction. If a transaction exists on the current thread when the method is called, the atomic method's work will become part of that transaction. If, on the other hand, no transaction exists, the method will run unsynchronized. This design choice was made to make the behavior of unmarked methods as close to that in standard Java or C#. This approach works well because, in most cases, a class is marked as atomic and then a few entry points are marked as beginning a transaction; all other methods in the type implicitly join the active transaction.

All methods belonging to an atomic type are assumed to support transactions. Atomic methods are not cloned; the instrumented version can run correctly with or

---

[3] In C#, properties are represented by a pair of get and set methods. For example, an integer property named *counter* would be represented internally by the methods `int get_counter()` and `set_counter(int newVal)`. Java does not explicitly support properties.

without a transaction. While this reduces code bloat (cloning methods effectively doubles code size), it also means that even threads that run without a transaction incur some of the overhead of the STM model. Where an atomic method is run without a transaction, concurrent access to atomic data types is illegal. Such access is detected at runtime and exceptions are thrown. There is no built-in ability to recover from this illegal state.

Methods can also be marked with an attribute indicating that they begin a transaction. Such methods define the scope of a dynamic transaction, which does not attempt to commit until the method returns. Closed nesting is not well supported—nested transactions are flattened. To ensure that a method is always run in the context of a transaction, the programmer can provide a parameter to the Atomic attribute indicating the desired semantics. The available options are Uses (this is the default behavior when no parameter is specified), Requires, Starts. The Requires option indicates that the method must always run in a transaction. If no transaction exists on the current thread, the method will automatically begin a new transaction before executing the user's code. However, if a transaction already exists, the "Requires" method will simply use it. In contrast, the Starts option indicates that the method will always begin a new transaction.

The Contains method shown below always runs in a new transaction:

```
[Atomic(XKind.Starts)]
public override bool Contains(int v) {
    Neighborhood hood = Find(v);
    return hood.currNode != null;
}
```

## 3.4 Atomic blocks

In addition to the method-based approach to transaction creation represented by the [Atomic(XKind.Starts)] annotation, Peet also supports the using(new XAction()) { }

block. This approach allows for dynamic control of transaction creation at the call site, so that one method can be called from different places, sometimes with, and other times without, a transaction. Critically, local variables used inside the atomic "using" block are saved and restored on abort, so that the block is semantically idempotent. Without this safeguard, the effect of transaction aborts and automatic restarts might be visible through mutations of local variables that were not members of atomic objects.

```
using(new XAction()) {
    // code here runs in a transaction
}
```

Both Visual Basic and C# support a “using” construct. Unfortunately, J#, .NET’s version of Java, does not. To overcome this limitation, however, Peet recognizes a special try-finally block preceded by an instantiation of the XAction class as being equivalent to a “using” block, as shown in the code fragment below:

```
// the J# version of a using(new XAction()) { } block
new XAction();
try {
    // code here runs in a transaction
}
finally {
    XAction.TxFinally();
}
```

The XAction() constructor used to syntactically mark the start of an atomic block is a no-op. The XAction.TxFinally method, however, is used to clean up the current transaction, as shown below. Where the transaction’s txJoinCounter is non-zero, this indicates that the atomic method or block simply joined an already existing transaction and thus the end of the method or block does not indicate the end of the dynamic transaction.

```
TX-FINALLY()
1   if currentTx = NIL
2       then return
3   if currentTx.txJoinCounter ≠ 0
4       then currentTx.txJoinCounter--
5           return
6   currentTx ← currentTx.parent
```

## 3.5 Definition of atomic types, methods and blocks

This section defines the semantics of the atomic annotation as applied to types, methods and blocks, in terms of their transformation into standard C#.[4]

### 3.5.1 Atomic types

An atomic type is a standard Java or C# type whose data can be accessed from within a transaction. Each method that begins a transaction in an atomic type defines the scope of a dynamic transaction; this includes constructors, static methods, and static initializers. At runtime, the data belonging to an atomic object is accessed through the underlying STM interface described in Chapter 4. The model verifies that code accessing atomic objects is executed in the context of a transaction.[5]

In its blocking mode, each atomic object implicitly implements the IRecoverable interface, defined below, and its two methods: Backup and Restore. These methods copy the value of the "real" fields to the "shadow" fields and back. Backup, which is invoked on the first attempt to write to an atomic object in the current transaction, copies the real fields to the shadow fields. The first attempt to write an atomic object is detected through the object's ownership field. If the writing transaction is not already the owner that indicates this is the first write by the transaction. Restore, which is invoked when the

---

[4] Once represented by a C# transformation, the standard semantics of that language are assumed.
[5] Warnings or error messages (depending on the compiler settings) are issued where atomic types are accessed outside of a transaction, or where non-atomic types are accessed inside a transaction.

previous writer to the object is found to have aborted, copies the last known good values from the shadow fields back to the real fields. The IRecoverable interface is defined as follows:

```
// Objects that can be backed-up and restored
// The main interface for blocking atomic objects
public interface IRecoverable {
    // Create a backup copy
    void Backup();

    // Restore backup copy
    void Restore();
}
```

For each field in an atomic type, our model defines a shadow field (named fieldname$shadow) that is used to store the backup value during the execution of a transaction. In addition, atomic types implicitly implement the Backup and Restore methods of the IRecoverable interface to copy the pre- and post-transactional values from the actual fields to the copies (on transaction start) and back (on transaction abort). *Const*, *readonly*, and TxSafe fields are considered idempotent and therefore do not require shadow fields.

Every atomic object also defines a reference to the appropriate SynchState object, which maintains the information about the transaction(s) currently reading and/or writing the atomic object's data. In the blocking model, this state is maintained by the TMemSynchState class; obstruction-free atomic objects use the OFreeSynchState type. All atomic objects define the appropriate SynchState object, instantiating the SynchState in the atomic object's constructor(s). Note that the SynchState field must be public, because it is accessed by algorithms inlined in the user's code.

The code fragment below shows pseudocode for the Node type described in § 3.3 in terms of its C# transformation:

```
                                    // implements IRecoverable
[Atomic] public class Node : IRecoverable {
    public int value;
    public Node next;

    // SynchState reference added by the compiler
    public TMemSynchState synch;

    // default constructor added, if necessary
    public Node() {
        // code to instantiate the SynchState added
        synch = new TMemSynchState(this);
    }

    // shadow fields
    private int value$shadow;
    private Node next$shadow;

    // copy fields to shadow fields
    public void Backup() {
        value$shadow = value;
        next$shadow = next;
    }


    // copy shadow fields to fields
    public void Restore() {
        value = value$shadow;
        next = next$shadow;
    }
}
```

In the obstruction free model, every atomic object implicitly implements the ICloneable interface (or its analog in Java, the Cloneable interface) and its Clone method:

```
    // The main interface for nonblocking atomic objects
    public interface ICloneable {

        // Create a new copy of the object
        object Clone();
    }
```

The Clone method creates a shallow copy of the atomic object by creating a new instance of the object and then copying the fields of the current object to the new object. For value type fields, a bit-by-bit copy of the field is performed. If a field is a reference type, the reference is copied but the referred object is not; therefore, the original object and its clone refer to the same object. In C#, the object.MemberwiseClone method is a shortcut that automatically performs these tasks, as shown below, whereas in Java the actual code must be generated:

```
                                 // implements ICloneable
[Atomic] public class Node : ICloneable {
    public int value;
    public Node next;

    // SynchState reference added by the compiler
    public OFreeSynchState synch;

    // default constructor added, if necessary
    public Node() {
        // code to instantiate the SynchState added
        //    represents the transaction-state of the obj
        synch = new OFreeSynchState(this);
    }

    // clone the object
    public object Clone() {
        return this.MemberwiseClone();
    }
}
```

### 3.5.2 Atomic methods

For atomic methods that may begin new transactions, the semantics of our model defines the transactional machinery required to start a new transaction, commit or abort it, retry aborted transactions, and trap stray exceptions from user code. Pseudocode for this transformation is shown below:

```
[RetType] retval;
while(true) { // keep trying until the tx commits
    XState me = XAction.TxStart(txKind);
    try {

        /* User's
                  Code
                       Here */

        retval = [stack];
        if(me.Commit())    // try to commit
            goto ReturnLabel; // success - return now
    }
    catch(AbortedExcecption) { // handle abort requests
        XAction.TxHandleAbort(me);
    }
    catch(Excecption e) { // trap user exceptions
        me.Abort();        // abort the transaction
        throw e;           // rethrow the errror
    }
    finally {
        XAction.TxFinally(me);
    }
}
ReturnLabel:
return retval;
```

The pseudocode generated by the compiler and shown above relies on several static helper functions: TxStart, TxHandleAbort, and TxFinally. The TxStart method is defined below:

TX-START(*txKind*)
1  **switch** *txKind*
2      **case** STARTS
3          *currentTx* ← **new** XSTATE(*currentTx*)
4          **return** *currentTx*
5      **case** REQUIRES
6          **if** *currentTx* = NIL    ▷ must be a root
7              **then** *currentTx* ← **new** XSTATE(NIL)
8                  **return** *currentTx*
           ▷ we aren't starting a new tx, so just increment the counter
           ▷ ensures that we don't commit on the way out
9          *currentTx.txJoinCounter*++
10         **return** *currentTx*

The TxHandleAbort static helper method is defined below; see § 3.4 for the TxFinally algorithm.

TX-HANDLE-ABORT(*me*)
1 **if** $me \neq$ NIL $\wedge$ $me.txJoinCounter \neq 0$
2 **then throw** ABORTED-EXCEPTION

In contrast to the delegate-based approach used by the SXM library (and described in § 2.4), one advantage of transactions started using the atomic attribute as shown above is that the method parameters do not need to be passed as an array of objects; likewise, the return value, if any, can also be typed correctly.

### 3.5.3 Atomic blocks

Atomic blocks by definition require an ambient transaction, meaning that a transaction will be created by the block, if one doesn't already exist. Where a transaction exists when the atomic block starts, the block will join that existing transaction (the XAction.TxStart static helper method defined above shows how this is done). Atomic methods called from within an atomic block will simply join the work of that block's transaction. Non-atomic methods not marked with the TxSafe attribute cannot be called from an atomic block; error messages are issued by the compiler.

Atomic blocks are similar to atomic methods, with two significant differences: first, no new return value is generated when the transaction ends; control flow simply passes to the end of the atomic block; second, and more interestingly, all local variables accessed within the atomic block must be backed up on transaction start and restored, if the transaction aborts. Because local variables are stored on the stack, not in fields of the atomic object, they are not captured by the environment of the atomic object. In order to maintain the correct transactional properties for atomic blocks, our model defines an

additional shadow variable for each local used in an atomic block. Atomic blocks may only access local variables (protected explicitly by the atomic block) or fields of atomic objects (protected by the transactional model); access to fields of non-atomic objects not marked with the TxSafe attribute is not permitted. The code fragment below shows the use of an atomic block and a local variable:

```
public override bool Insert(int key) {
    using(new XAction()) {
        key++;
        NestedInsert(key);
    }
}
```

Atomic blocks are defined in terms of their equivalent C# transformation below:

```
public override bool Insert(int key) {

    // backup all locals used in the atomic block
    int key$shadow = key;

    START_LABEL: // retry loop

    XState me = XAction.TxStart(); // start the tx

    using(new XAction()) {
        // try { - provided by the using block

            /* USER CODE HERE */
            key++;
            NestedInsert(key);

            if(!me.Commit())    // try to commit
                goto START_LABEL;
        // } - end try
        catch(AbortedExcecption) { // abort requests

            // restore locals used in the atomic block
           key = key$shadow;

            XAction.TxHandleAbort(me);
        }
        catch(Excecption e) { // trap user exceptions
```

```
                // restore locals used in the atomic block
                key = key$shadow;

                me.Abort();          // abort the transaction
                throw e;             // rethrow the errror
            }
            // finally { - provided by the using block
                // change Dispose call to TxFinally call
                XAction.TxFinally(me);
            // } - end finally
        }
    }
```

Thus mutations to local variables in an atomic block are captured and restored according to the fate of the ambient transaction. Mutations to the fields of other atomic objects are also safe, as such objects participate in the transaction by definition. Mutations to other, non-atomic and non-local, data are detected and the compiler issues warnings. Similarly, calls to non-atomic methods (e.g., I/O routines) are not considered safe; further warnings are issued.

## 3.6 Limitations and restrictions

This section describes the limitations and restrictions on atomic types, methods, and blocks, with a focus on how they interact with other standard language features, such as static and virtual methods, reflection, unsafe code, inheritance, nested types, and anonymous methods.

### 3.6.1 Static and virtual methods

Static methods and fields are fully supported. Virtual methods are supported in atomic types, although the compiler does not perform interprocedural optimizations for

such methods[6] (see § 4.2 for details on the interprocedural optimizations supported for non-virtual atomic methods). This limitation on the compiler's interprocedural optimizations also applies to calls made through interfaces or delegates, where the ultimate runtime type of the target object cannot be safely determined statically. Dynamically loaded types are supported and treated no differently than statically loaded types.[7] Because our model is static in the sense that the compiler must process atomically-annotated code prior to execution, using reflection to dynamically add new fields or methods to an atomic object is not supported. Finally, nested transactions are not supported (see § 3.8 for a proposed alternative).

### 3.6.2 Unsafe code and pointer aliasing

Atomic methods may not contain *unsafe* code in C#[8]. By consequence, this restriction disallows a programmer from accessing direct pointers to fields of atomic objects. Pointer aliasing would be backdoor through which the transactional model could be bypassed and therefore is not permitted. The atomic attribute of a type or method can be detected using standard reflection techniques; atomic blocks, however, are not visible through reflection as they are not defined via attributes.

### 3.6.3 Inheritance of atomic types

Atomic types may be subclassed; both the base class and the derived type must be marked as atomic (the compiler will issue error messages if this requirement is violated; the only exceptions are implicit derivation from the *object* base type and the immutable

---

[6] Virtual methods are marked explicitly with the *virtual* keyword in C#. In Visual Basic, virtual methods are defined with the *Overridable* keyword. In Java, where all methods are presumed to be virtual, the compiler currently does not support interprocedural optimization.
[7] The programmer is obviously responsible for ensuring that any atomic types loaded dynamically have been processed by the STM compiler.
[8] Atomic methods containing code marked as "unsafe" in C# are detected and the compiler issues an error message.

*string* type). When an inheritance relationship is established among two atomic classes, the compiler automatically chains the backup and restore methods in the derived type to those in the base type so that when the derived type is backed-up or restored, the methods of both classes are called, ensuring that all fields of the aggregate atomic object are protected; this includes private as well as static fields.

### 3.6.4 Nested types and anonymous methods

Atomic types may be nested in atomic or non-atomic outer classes. Practically speaking, however, it will usually make sense for both the inner and outer classes to be marked as atomic, particularly in Java, where fields of the outer class can be accessed directly from the "this" reference in methods of the inner class.[9] Anonymous methods in C# cannot be marked as atomic, although atomic blocks can be used to access atomic objects. In Java, anonymous inner classes are likewise not supported for accessing transactional objects. Finally, atomic types support generics and can be used as generic types, although this generally only makes sense where the instantiated generic type is also marked as atomic.

### 3.6.5 Exceptions

There is some debate in the literature about how to handle exceptions thrown from within transactions. Harris [33] describes some of the possible approaches for determining the correct state after an exception is thrown. While a number of possibilities exist, the most standard technique is to trap the exceptions that escape a transaction, abort the transaction, and then re-throw the error. The problem here is that it is possible that the exception object may contain references to transactional data whose state is rolled back

[9] In Java, the inner class has a hidden field this$0, which holds a reference to the outer object. This hidden reference is used for all outer class field access from the inner type.

by the abort operation. Value (i.e., not reference) data stored in the exception object, on the other hand, will remain unaffected by an abort operation. This means that the state seen by the creator of the object may or may not be the same as that seen by its recipient. One solution favored by transactional purists is to mark the exception types themselves at atomic, ensuring that their state is rolled back when the transaction aborts. But this approach has the disadvantage of making it impossible for a transaction to communicate in any meaningful way with the recipient of an exception. Because there is no broad consensus in this area, our model allows the programmer to decide which approach is preferable. By default, exception objects that are not marked as atomic will be thrown in the usual fashion, with all data from within aborting transaction unmodified. Otherwise the programmer can mark exception objects as atomic – the throw mechanism will still work but any state created within the transaction will be rolled-back by the abort operation.

## 3.7 Retry and conditional execution of atomic blocks

The SXM library supports a retry operation that waits until the current transaction has been aborted before re-attempting execution of the transaction. This is useful when the value read in one transaction depends on another transaction to modify that value for its success. The canonical example is that of a shared buffer, where one transaction is reading a value from the buffer while another is writing a value to it. The code below shows the Put method, which retries whenever the buffer's size reaches its maximum capacity. The retry operation then waits for another transaction to read and remove a value from the buffer (thereby invalidating and aborting this transaction), before retrying.

```
// Put a value into the buffer. Retry if full.
[Atomic(XKind.Starts)]
public void Put(int v) {
    // check for the condition
    if(buffer.size + 1 == buffer.capacity)
        XAction.Retry(); // retry...

    buffer.data[++buffer.size] = v;
}
```

Peet offers an alternate way of expressing this behavior. Instead of coding an explicit check for a condition and then executing the retry function, the condition can be more elegantly expressed as a parameter of the *using(new XAction(booleanCondition)) { }* block. This overloaded version of the XAction constructor causes the compiler to generate code rather similar to that shown above. The programmer, however, can express that behavior as shown here:

```
// Put a value into the buffer when not full
[Atomic(XKind.Uses)]
public void Put(int v) {
                      // condition
    using(new XAction(buffer.size + 1 != buffer.capacity))
        buffer.data[++buffer.size] = v;
}
```

In this case Peet overloads the meaning of the *using* block. Normally, the value passed to the XAction constructor would only be evaluated once. Peet, however, generates code that causes the entire expression to be reevaluated every time the transaction is executed. The body of the using block is executed only when the Boolean condition evaluates to true.

## 3.8 Transactional boosting

Transactional boosting [45] is a technique that transforms highly-concurrent linearizable objects into transactional objects. Each method call is associated with an abstract lock, and locks for non-commuting calls conflict. Each method call must also have an inverse, which is logged and applied to the object when a transaction aborts. As discussed elsewhere, transactional boosting can enhance both the concurrency and performance of shared objects. Finally, transactional boosting is a safe alternative to open nested transactions [69].

Consider, for example, a shared set providing methods to add and remove integer items. Calls with distinct arguments commute (say, add(4) and remove(5)). When add(4) is called, the inverse remove(4) call is logged, in case the add operation must later be undone. Operating at this level of abstraction has advantages over more traditional STM models of word-based or object-based conflict detection and resolution. For example, consider a standard STM implementation of a sorted integer list containing the elements {1, 3, 5}. While there is no reason that operations to add(2) and add(4) cannot execute concurrently, these operations will in fact conflict in standard STM systems, since one transaction will always write to a node read by the other.

By exploiting the programmer's understanding of an object's semantic properties, transactional boosting makes it possible introduce concurrency where traditional STMs do not. The initial proposal for transactional boosting, however, required that the programmer write special wrapper classes for existing linearizable objects in order to build the undo logs. We eliminate the need for explicit wrappers by introducing language and compiler support for specifying method inverses and abstract locks. Rather than

writing code to express the locking and undo logging, our approach uses annotations to specify the behavior of transactional boosting via a contract technique.

We support transactional boosting through three new annotations: *TxBoosting*, *Inverse* and *AbstractLock*. The *TxBoosting* attribute is applied to a class instead of the *Atomic* attribute; it signifies to the compiler that the type is safe for transactional use, but that its data does not need to be protected in the traditional manner because the object is already linearizable. The *Inverse* attribute is applied to a method in order to identify its logical inverse, which must have the same signature as the marked method. The *Inverse* annotation also accepts a parameter specifying under what circumstances the inverse call should be logged for undo purposes. While custom handlers can be specified, generally a boolean return value is sufficient to indicate that the method has successfully altered the abstract state of the object, thus requiring that the inverse operation be logged. In the example below, the *ReturnsTrue* flag indicates the corresponding method inverse, *Remove*, should only be logged when the *Insert* method executes successfully (i.e., if the *Insert* method fails, the *Remove* method is not logged, as there is no change in the object's state that must be undone):

```
[TxBoosting]
public class RBTree {
    [Atomic(XKind.Starts)]
    [Inverse(InverseKind.Method, "Remove",
        InverseCondition.ReturnsTrue)]
    public bool Insert([AbstractLock] int key) {
        // ...
    }

    [Atomic(XKind.Starts)]
    [Inverse(InverseKind.Method, "Insert",
        InverseCondition.ReturnsTrue)]
    public bool Remove([AbstractLock] int key) {
        // ...
    }
```

```
}
```

Exceptions thrown from boosted methods operate in the same way as exceptions thrown from regular atomic methods. The exception is trapped by an outer exception handler inserted by the compiler and the current transaction is aborted. In the case of a boosted method, the abort procedure requires executing the log of inverse operations and the release of any abstract locks. If a boosted method threw an exception prior to completing its action, the inverse method is expected to detect and undo any partial updates.

The optional *AbstractLock* attribute identifies the parameter that defines the abstract lock that must be acquired by this method. Without the *AbstractLock* annotation, transactional boosting automatically acquires an exclusive lock, resulting in the serialization of all *Insert* calls. With the addition of the *AbstractLock* attribute, however, the abstract lock acquired only blocks on the *Insert* operations for this particular value. Note that such key-based locking may block commuting calls (e.g., two calls to Insert(3) where 3 is in the set), but it provides sufficient concurrency for practical purposes. Abstract locks are automatically released when the transaction commits or, where it aborts, after the undo log has been executed. For cases where the *Inverse* and *AbstractLock* attributes are not sufficiently flexible to capture the desired concurrency semantics of a transactionally boosted object (see, e.g., the HeapRW example [45]), the model permits programmers to invoke helper methods that perform the customized inverse operation. This compiler and language support for transactional boosting achieves the same striking performance improvements as the prior hand-crafted support reported by Koskinen and Herlihy in [45]. For example, Figure 3.1 shows the performance

provided by the compiler for transactionally boosted skip lists using standard and key-based two-phase locks:

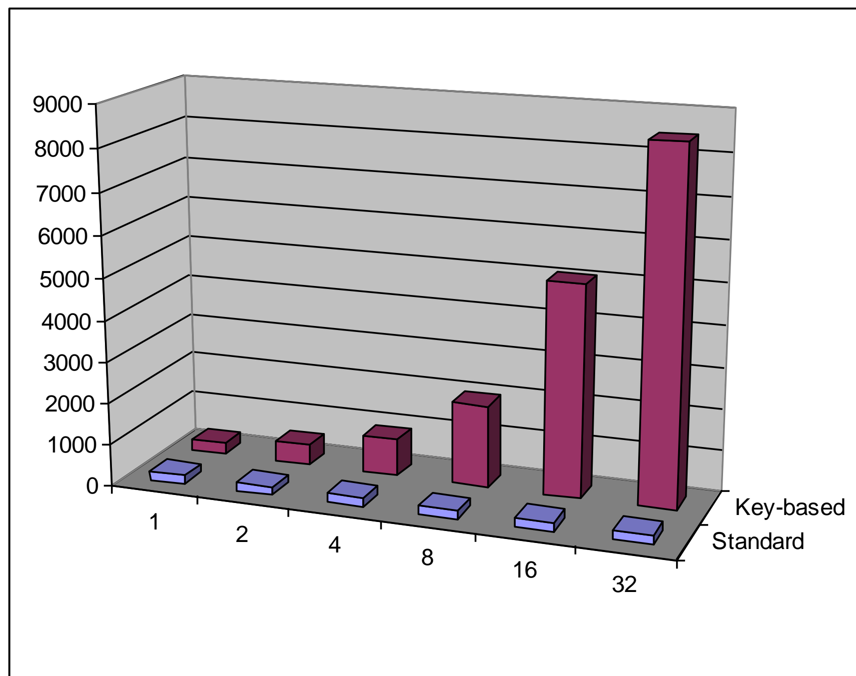



Figure 3.1: Throughput for a transactionally-boosted skip list using standard and key-based two-phase locks

# Chapter 4

# Optimizations

## 4.1 Properties

Software transactional memory systems can be categorized along several axes. First, STMs can be classified as *word-based*[1], synchronizing on uninterpreted regions of memory, or *object-based*, synchronizing on language-level objects. Second, they can either use short locks or support a nonblocking progress condition such as obstruction freedom. Finally, they may or may not support fully consistent reads and orthogonal contention management. Much recent research has focused on word-based systems [1, 15, 16, 17, 25, 74], which are well-suited for "unmanaged" languages such as C or C++.

[1] In practice, such systems usually use cache lines as their most basic unit of synchronization.

If, however, we turn our attention to “managed” languages such as Java or C#, object-based STM systems have a number of attractive features. Because memory accesses are routed through method calls, it is feasible to provide *strong atomicity* [5, 6, 52, 66, 80], ensuring that transactional and non-transactional code are properly synchronized, thus avoiding a class of difficult-to-detect synchronization errors. Moreover, object-based designs support *transactional boosting* [45], a safe alternative to open nested transactions that can be used to enhance concurrency for “hot-spot” shared objects. Finally, object-based STMs offer better opportunities for optimization because the semantics of object-oriented languages naturally provide the context for each field, allowing for a higher level understanding of memory access patterns.

Peet does not support concurrent access to atomic data structures among transactional and non-transactional threads. While any access to an atomic object from a non-transactional method is disallowed by the compiler, it is possible for dynamically loaded code to avoid these static checks. Because Peet is a post-compiler it cannot guarantee that it will see all dynamically loaded code at compile time. However, were Peet’s STM technology integrated with the main C# compiler, this problem would be obviated.

At runtime, if a (transactional) method running without a transaction concurrently accesses an atomic object being used by one or more transactional threads, such conflict is detected. The STM library checks for actively writing transactions, even where a reader or writer is currently running outside of the transactional context. In such cases, a PanicException is thrown at the non-transactional thread, as shown by the fragment from the atomic read and write algorithms below. While not an ideal solution, this model

leaves open the possibility that the conflicting transactional writer could be aborted, which would allow the non-transactional thread to proceed.

```
// Atomic reads and atomic writes
if(me == null) // not in a transaction, update in place
    if(oldWriter == null)
        return oldLocator.newObject;
    else
        switch((XStates)oldWriter.state) {
            case XStates.COMMITTED:
                return oldLocator.newObject;
            case XStates.ABORTED:
                return oldLocator.oldObject;
            case XStates.ACTIVE:
                throw new PanicException(
                    "Tx/not-tx conflict");
        }
```

The most straightforward way to provide an object-based STM is as a library [41, 47, 48, 62]. However, it has been our experience, and the experience of others [14], that it is difficult to implement object-based STM libraries that are both efficient and provide a simple programming interface, because a simplified interface usually requires a high level of abstraction that does not permit non-local optimizations. As a programming model, transactions have a pervasive influence on both data and control flow. For example, the DSTM library [48], the first library of this kind, required programmers to express data and control structures in an idiosyncratic way, explicitly opening and closing shared atomic objects, and hand-coding retry loops. One important advantage of the DSTM2 [47] and SXM [41] libraries, which build on DSTM, was that they offered a more natural programming interface than their predecessor, automatically generating transactional “wrappers” around fields of shared atomic objects, and automatically retrying failed transactions. Nevertheless, while transactional wrappers led to a simpler, less error-prone programming style, overheads could be high, since libraries did not know

how they were used by clients, and there was little or no opportunity for non-local optimization. Dalessandro et al. report similar dissatisfactions with the RSTM2 [14] library.

It is the premise of this chapter that compiler support is the key to achieving both a simple programming interface and adequate performance in object-based STM systems. Fortunately, we show that relatively straightforward measures can be very effective. Underlying these measures is an STM model that uses annotations to describe atomic types and methods. Annotations are an attractive way to prototype language features because they are easily extensible, familiar to programmers, and do not require introducing new syntactic structures. We describe an STM compiler that performs a dataflow analysis to determine whether an atomic object is accessed in *read* or *write* mode and whether a specific atomic field access is guaranteed to occur subsequent to a previous use of that object. Based on this information, the compiler makes more efficient use of the STM library, inlining much of the transactional machinery directly in the user's code in the form of bytecode instructions. The compiler supports a natural and largely transparent programming interface, and, to date, offers a performance improvement of between four and ten times compared with the same algorithms implemented solely with an STM library. This performance improvement helps close the gap between STM systems and conventionally synchronized concurrent algorithms.

This chapter makes the following contributions:

- This is the first application of whole object dataflow analysis to track the state of open transactional objects.

- Subsequent reads and writes can be made entirely lock-free, even in the context of a blocking STM that relies on short-lived locks for initial reads and writes.
- Based on this application, compiler optimization can yield non-blocking, obstruction-free STMs that perform at least as well as comparable blocking systems.

Section 4.2 describes the static whole object analysis performed by the compiler in order to make the most efficient use of the underlying STM library. Sections 4.3 and 4.4 detail the optimizations unique to the blocking and obstruction-free models, respectively. Library optimizations that support the retry operation and conditional atomic blocks outlined in § 3.7 are the focus of section 4.5. Section 4.6 proposes a novel variant of the Compare-and-Swap hardware primitive specifically designed to support STM libraries. Finally, section 4.7 describes important optimizations in the handling of atomic arrays.

## 4.2 Whole object analysis

Today's STM systems can be broadly divided into two categories depending on the underlying progress condition they support (i.e., a non-blocking progress condition such as obstruction-freedom, or a blocking model that uses short-lived locks). The compiler and the underlying STM library support both approaches. Conventional wisdom holds that although non-blocking algorithms have nicer theoretic properties (e.g., no deadlock or priority inversion), they are nearly always significantly outperformed by functionally equivalent blocking algorithms. We find that with good compiler and library optimizations this no longer holds true—our obstruction-free mode outperforms the blocking mode in nearly all cases.

We have approached optimizations at two levels: traditional flow-based compiler optimization directed towards generating more efficient code and direct optimizations in the underlying STM library. Both have turned out to be extremely valuable in achieving our overall performance results. The compiler supports a whole object view of dataflow analysis that builds on our object-based language interface for STM. In the underlying STM library, we made a number of improvements to the transactional synchronization and validation algorithms that maintain the transactional state of each atomic object.

As noted by Harris et al. in [38], library-based STM systems blindly redirect all heap access through the STM interface, without first checking whether the object is thread-local or has been opened previously. Thus, dataflow analysis can help determine whether an atomic object must be accessed transactionally, or is guaranteed to be thread-local. Second, it can determine when a field is accessed for a first or second (i.e., subsequent) time, guaranteeing that the object is already open for reading or writing. This is possible primarily because the compiler is built on top of an object-based STM, not a word-based one as in [34].

### 4.2.1 Partial redundancy elimination

The compiler can exploit our observation that field reads performed on atomic objects that the transaction has previously read or written can proceed without any locks to significantly reduce the overhead of STM. First the compiler must be able to detect those atomic field accesses guaranteed to occur subsequently to a prior access by the same transaction. To accomplish this, the compiler uses a dataflow analysis known as Partial Redundancy Elimination (PRE) [56, 86], a form of Common Subexpression

Elimination (CSE). We offer no formal proofs of correctness for PRE here, instead deferring to the arguments made in the literature [11, 86].

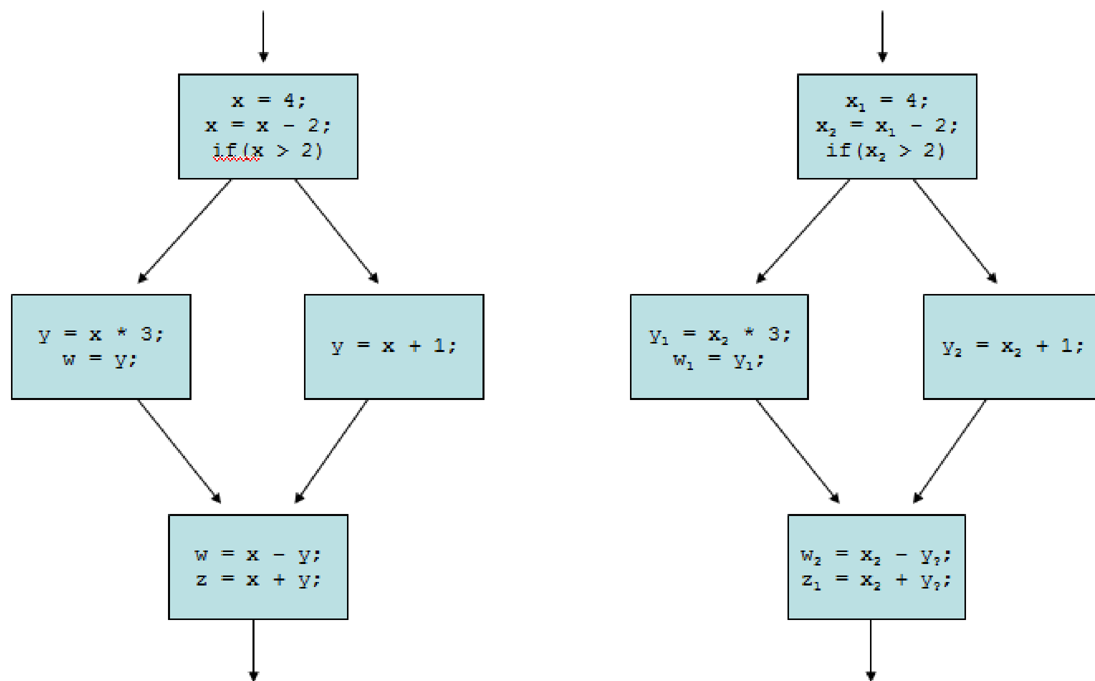
An expression is called *partially redundant* when the value computed by the expression is already available on some but not all paths through the control graph to that expression; an expression is *fully redundant* when the value computed by the expression is available on all paths through the control graph to that expression (this is the traditional CSE analysis). PRE can eliminate partially redundant expressions by inserting a computation for the partially redundant expression on the paths that do not already compute it, thereby making the partially redundant expression fully redundant. In our application of PRE in the context of a compiler for software transactional memory, the "computation" consists of opening an atomic object for read or write access so that it is guaranteed to be open at some future point in the transaction.

We employ a PRE dataflow analysis algorithm based on the Static Single Assignment[2] (SSA) format [11, 86], an intermediate representation in which every variable is assigned exactly once. Existing variables in the original IR are split into *versions*; each version of a variable is indicated by the original name with a subscript, so that every definition gets its own version. In SSA form, use-def chains are explicit and each contains a single element. A function converted into SSA format has the property that every use of a variable is reached by at most one definition of that variable. Moreover, each definition of a variable dominates all uses of that variable. To accomplish

[2] In functional languages, such as Scheme, ML and Haskell, continuation passing style (CPS) is generally used where one might expect to find SSA in a compiler for an imperative programming language such as C. SSA and CPS are formally equivalent, so optimizations and transformations formulated in terms of one can be applied to the other.

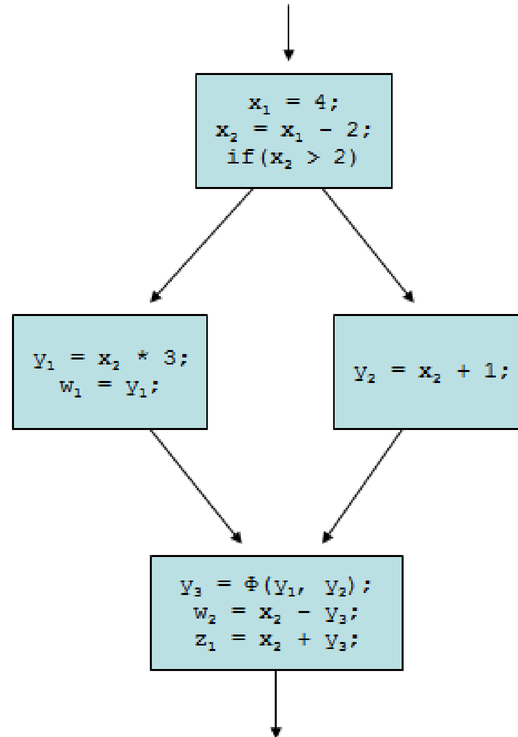
this, the SSA feature adds pseudo operations that merge different definitions into new and unique definitions.

SSA models dependencies as a graph, where they are represented as edges between the intermediate representation's operand objects. Converting ordinary code into SSA form requires replacing the target of each assignment with a new variable, and replacing each use of a variable with the "version" of the variable reaching that point. For example, here is a simple control graph, and that same graph converted to SSA format:




In SSA format, each use of a variable is given a subscript; each possible assignment of a variable is given a new, unique, subscript. Notice, however, that in the last basic block it is not possible to statically determine the correct version of the *y* variable, since each possible path to that block gives a different version of *y*. At such points, where different versions of the same variable meet, a Φ (Phi) function is added to

generate a new version of that variable that joins two or more incoming versions, as shown below:




The correct placement of Φ functions is determined using *dominance frontiers*. A node is said to *strictly dominate* another node in the control flow graph, iff it is impossible to reach the second node without passing through the first. This means that whenever we ever reach the second node, we know that any code in the first has already run. Furthermore, a node is said to be the *dominance frontier* of another node, iff the first node does not strictly dominate the second, but does dominate one of its immediate predecessors. From the perspective of the first node, these are the nodes at which other control paths that don't go through that node make their earliest appearance.

Thus, dominance frontiers capture the precise places at which we need Φ functions: if a node defines a certain variable, then that definition and that definition

alone (or redefinitions) will reach all other nodes dominated by that node. Only when we leave these nodes and enter a new dominance frontier do we need to account for other execution paths that may supply different definitions of the same variable. For proofs that the algorithm identifying dominance frontiers is correct, we defer to Chow et al. [11].

In the sample control graph show above, a traditional CSE optimizer would not be able to make any assumptions about $y_3$ that weren't also true of $y_1$ and $y_2$. PRE optimizers, however, can insert computations along any paths where they are missing. In the context of STM, this requires adding an appropriate atomic open operation along the missing paths so that the compiler can be sure that the result of the Φ function is also "open." This approach enables the compiler to determine those reads and writes that are guaranteed to be executed subsequent to a full open. In such cases, the subsequent reads and writes can be of the fast-path style and are fully inlined in the user's code. This improves performance significantly, as only a lightweight "open" is performed and no method invocation is required.

Note that this is quite different from the peephole optimizers described by Harris et al. in [38] and Adl-Tabatabai et al. [1], which seem to rely primarily on the compiler's existing CSE optimizer to reduce the overhead of STM. The STM described by Saha et al. in [74] is cache line-based rather than object-based, which means it can't assume that all fields of an object are accessible after one field of that object has been accessed. The log-based system of Harris et al. in [38] does not appear to support a non-blocking implementation.

Furthermore, our dataflow graph also incorporates references to atomic object stored in fields of other atomic objects. In other words, where a reference to an open

atomic object is stored in a field of another atomic object, the dataflow graph tracks its open state. This is possible because atomic objects are fully isolated (our model supports always consistent reads), and therefore fields of atomic objects written by one transaction will never be visible to another concurrently executing transaction, even after the first transaction commits. It is thus safe for the dataflow analysis to build a flow graph this includes such references. Note that although this information could be used for the interprocedural optimizations, currently only the “open” status of the actual method arguments is considered.

### 4.2.2 Reaching definitions analysis

Every basic block in the control flow graph consists of a set of instructions executed sequentially. Each instruction dominates all instructions that follow it in the basic block and post-dominates all instructions that precede it in the basic block. In short, a basic block represents a single-entry, single-exit region of the control flow graph. The control flow graph must terminate a basic block wherever there is the potential for an instruction to thrown an exception (sometimes referred to in the literature as a Potential Exception-throwing Instruction, or PEI [9]), which would prevent any instructions that might come after it from post-dominating that instruction. Thus, each basic block begins with either a label instruction or the successor of an instruction that might cause an exception, and ends with either a branch (including a call) instruction or an instruction that can cause an exception. Figure 4.1 shows the control flow graph for a simple method; the red dashed edges leading to basic block #6 show exceptional control flow.

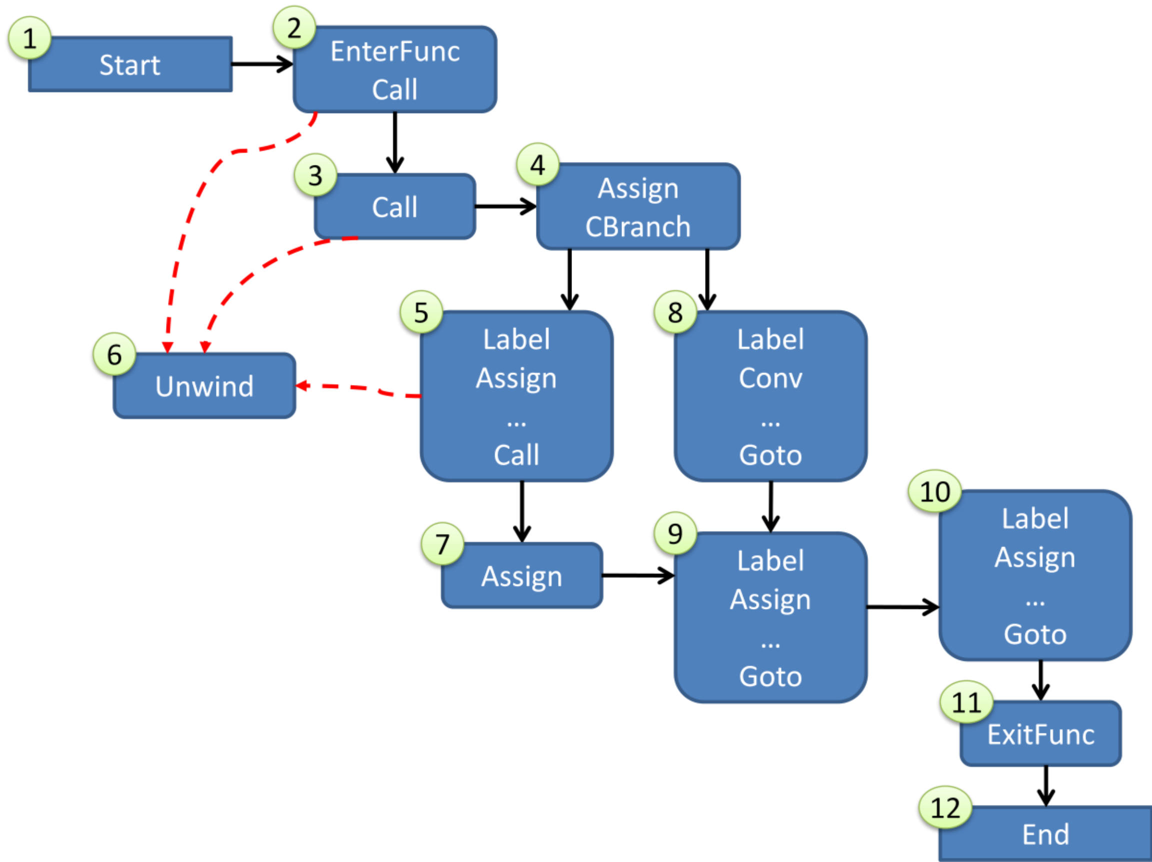



Figure 4.1: Sample control flow graph for a simple function

As shown in Figure 4.1, as well as the sample function below, every instruction that might raise an exception (or otherwise transfer control in an unusual way) indicates this potential with a source label operand that targets the exception handling structure that will ultimately receive control. So, for instance, the call to method Foo() will have an exception handler edge that targets the catch filter associated with the *[...]* construct. Likewise, the call to Bar() will have an exception handler edge that targets the special label *Unwind*, which indicates that control leaves this function in an unusual fashion (see Figure 4.1 above for the Unwind block).

```
int example() {
    int x = 3;

    try {
       x = Foo();
    }
    catch([...]) {
       Bar(x);
    }

    return x;
}
```

The flow graph builder treats these edges just like any other. So both Foo() and Bar() end basic blocks. If the exception-raising instruction has definitions, then the compiler accounts for the fact that the definitions may not happen until it is determined that control will continue normally. So, for instance, in the example above, the definition x = 3 is live at the return point since the assignment to x in the *try* may or may not happen, depending on whether Foo() throws an exception. Such definitions are known as dangling definitions and the SSA package is aware of them. So the use of x in Bar(x) is known to be 3. If it can be proven or asserted that a call does not cause unusual control flow, the label can be removed and the call will no longer end a basic block.

Our forward intraprocedural dataflow analysis is known as a reaching definitions analysis, which computes a set of definitions (assignments) of variables that may reach an instruction. This information is used to construct use-def chains. The algorithm consists of three phases: (1) Summarization, which summarizes the Gen and Kill sets (for reaching definitions) for each basic block; (2) Fixed-point, which uses these summaries to compute a fixed-point solution for each basic block entry; and (3) Local propagation, which propagates the fixed-point solution to instructions within a basic block. Reaching definition information holding at the exit of a basic block is propagated from the basic

block to all its successors. The definitions below show these three standard phases for a reaching definitions analysis algorithm.

```
PERSISTENT-VARIABLES
1   for each Block b
2       In_B: set of definitions that reach the entry of B
3       Out_B: set of definitions that reach the exit of B
4       Kill_B: set of definitions that are killed in B
5       Gen_B: set of definitions in B that are not later killed in B
```

```
SUMMARIZATION(Block b)
1   for each Instruction i in b
2       if i contains a definition d
3           then Kill_B ← Kill_B ∪ ALL-DEFS(d)
4                Gen_B ← Gen_B – OTHER-DEFS(d) ∪ {d}
```

```
FIXED-POINT(Block b)
1   In_B ← ⋃_{p∈predeseccors(b)} Out_p
2   Out_B ← In_B - Kill_B ∪ Gen_B
```

```
LOCAL-PROPAGATION(Block b)
1   Current ← In_B
2   for each Instruction i in b
3       Current_i ← Current
4       if i contains a definition d
5           then Current ← Current - OTHER-DEFS(d) ∪ {d}
```

ALL-DEFS($d$) contains all definitions of the variable defined a $d$
OTHER-DEFS($d$) ← ALL-DEFS($d$) - $\{d\}$

Dataflow analysis is able to detect many of the instances where an object is first read and later written in the same transaction. In such cases, it is more efficient to open the object in write mode initially, and therefore OpenRead calls are automatically promoted to OpenWrites. Where the dataflow analysis detects that an atomic object is used across basic blocks, the Φ node is inserted at the point where two or more object

references merge. If an atomic object will be accessed two or more times subsequent to a Φ node, the compiler inserts a preOpen call at the Φ node. Where the object will only be read, the OpenRead call is inserted; otherwise the compiler inserts a call to the OpenWrite method. This optimization makes it possible for all actual field accesses to be inlined as subsequent accesses with the fast-path code.

For example, the code fragment below from the RBTree benchmark shows a three-pronged *if* statement. Each possible branch reads the atomic node object; two branches (the *else-if* and the final *else*) also modify the node reference. The compiler inserts a Φ function to join the three possible node versions after the *if* block. In this example, nothing can be said about the status of $node_4$ returned by the Φ function. $Node_1$ is certain to have been opened for reading, but $node_2$ and $node_3$ are not yet opened.

```
RBNode node = root;          // node1

if(key == node.value)        // node1(R)
    break;
else if(key < node.value)    // node1
    node = node.left;        // node2 = node1
else
    node = node.right;       // node3 = node1

// node4 = Φ(node1(R), node2, node3)
```

If $node_4$ is used subsequently read, the compiler will insert a preOpen operation on both the *else-if* ($node_2$) and *else* ($node_3$) branches, resulting in a better result at the Φ function. Here the PRE analysis makes sure that all branches leading to $node_4$ are already open for reading, thus the compiler can statically determine that $node_4$ must also be open for read access:

```
RBNode node = root;          // node1

if(key == node.value)        // node1^R
    break;
else if(key < node.value) {  // node1
    node = node.left;        // node2 = node1
    // preOpenR(node2)
}
else {
    node = node.right;       // node3 = node1
    // preOpenR(node3)
}

// node4^R = Φ(node1^R, node2^R, node3^R)
```

Since preOpens are inserted at points where no object access existed in the user's code, the compiler must be careful not to introduce any new exception paths at the preOpen locations (see, e.g., the two preOpen operations added in the code fragment above). This is important because, if an atomic object reference is null when the preOpen call is made, it would be incorrect to have a null reference exception thrown from the spot, since the programmer would have no reason to believe that such an exception might occur there. As a result, if a target object reference is null at the time the preOpen call is made, it will fail silently, as shown below. The exception avoided at a pre-open will be trapped on the subsequent read or write.

```
if(atomicObj != null)
    preOpenR/W(atomicObj);
```

As an example of the type of performance optimization made possible by our PRE-based dataflow analysis, we turn to a fragment of the List benchmark's Insert method, first introduced in § 2.5. In the seemingly straightforward code shown below, we identify the possible optimizations that might be performed by a particularly insightful programmer:

```
[1]     Node newNode = new Node(v);
[2]     Node prevNode = hood.prevNode;
[3]     newNode.next = prevNode.next;
[4]     prevNode.next = newNode;
```

First, the new Node object instantiated in the first line is an atomic object, but never escapes this method. The constructor for the atomic Node object invoked in step 1 also joins the active transaction, so that the constructor does not represent a possible avenue for the new node to escape the scope of the transaction. Even when it is stored in the prevNode.next field in line 4, prevNode is also an atomic object, and so the update will not be visible to any other transaction until this transaction commits. Therefore, any access to the newNode object (e.g., line 3) does not need to use any transactional protection. Next, notice that prevNode object is used twice in the algorithm; the first time for a field read in line 3 and again for a field write in line 4. Rather than opening the object twice, first for read access and immediately after that for write access, it will be more efficient to open the object in "write" mode at line 3, and then to perform a lightweight write at line 4 as the prevNode object will already record the current transaction as a writer. In the obstruction-free model, subsequent writes can simply cache the object returned by the prior open (see § 4.4); in the STM model that uses short locks, subsequent writes need to be protected from concurrent aborts (see § 4.3).

What this mental exercise shows is that even straightforward code sequences often present excellent optimization opportunities to an insightful programmer (or a well designed compiler). The problem is that even clever programmers will find it challenging to identify all such possible optimizations in larger code sequences (not to mention the optimizations that may be possible in the interprocedural context). If they do, the resulting code will be extremely fragile and difficult to maintain, as it will be unclear why

references are being held for later use. Imagine that the code sequence above is later changed so that no write is performed at line 4—will the programmer also remember to change line 3 to open the object in read mode?

One way to evaluate the compiler optimizations are to compare their performance to hand tuned code. As discussed above, finding all possible optimizations is actually much more difficult than it may seem initially and the resulting code is usually both fragile and difficult to understand. For this evaluation, however, we hand-optimized the list benchmark in order to compare it with the compiler optimized version. The results for this relatively simple benchmark showed that the compiler optimized version of the List benchmark performed at 98% of the hand optimized version. As expected, the code of the hand optimized version was very difficult to read.

Finally, at the top every atomic method, the compiler inserts a startup instruction that stores the current transaction in a local variable, as shown below. This simple, but significant, optimization avoids later field accesses from needing to retrieve the current transaction, which is stored in a static thread-local field. Reading thread-local values is known to be expensive, and this optimization reduces such access to once per method execution.

```
[Atomic(XKind.Uses)]
Neighborhood Find(int v) {
    // inserted by the Peet compiler
    XState me = XAction.Current;
    ...
```

#### 4.2.3 Interprocedural optimizations

Where the methods of an atomic object are non-virtual, our object-based model performs a limited interprocedural optimization pass only among the methods of that

atomic object. Thus, our interprocedural optimization might be more accurately called an intra-type, or whole object, optimization. At method boundaries, the arguments passed to and received by a method are evaluated for their “open” status (return values are not tracked). In the current implementation, only where *all* the callers of an atomic method pass, as a parameter, an atomic object reference opened for reading and/or writing, that status is the starting point for the intraprocedural dataflow analysis in the target method of the same class; a more flexible approach might permit various non-overlapping mixes of open parameters by generating method proxies that open the remaining parameters and pass the result onwards to the target method.

Because atomic classes may be subclassed, derived classes must be able to determine the assumptions made regarding the open status of the parameters of the subclass’ methods. The compiler addresses this problem by creating annotations for each method parameter it presumes to be open in read and/or write mode. This metadata is checked by the compiler when a superclass calls a method in a subclass; if the open status of the parameters in the superclass method don’t match those expected by the target subclass method, the compiler inserts the necessary open operations immediately prior to the call.

Our object-based model also makes it possible for the compiler to pass the cached transaction reference (i.e., “me”) among methods of the atomic object. Here, overloaded methods are created that accept an extra argument, the current transaction reference, and all call sites are updated to pass the extra parameter to the target method. This requires overloading the method and changing its signature in order to avoid breaking any client

code that was either not compiled by Peet or that uses dynamic methods such as reflection to call it. Finally, all call sites are updated to load and pass the extra parameter.

```
// overloaded version generated by the compiler
// for use by the interprocedural optimizer
[Atomic(XKind.Uses)]
Neighborhood Find(int v, XState me) {
    ...
```

Then the original unadulterated method, which is now only called by foreign code, simply retrieves the current transaction reference from the thread-local value and calls the new overridden version of the method that requires the extra parameter. Thus, either the original or the overloaded method may be called safely; passing the extra parameter to the overloaded method is simply an optimization.

```
[Atomic(XKind.Uses)]
Neighborhood Find(int v) {
    XState me = XAction.Current;
    return Find(v, me); // call the overridden version
}
```

## 4.3 Blocking optimizations

In the STM model that uses short locks, the compiler generates a shadow field for each field of an atomic type to store the backup value during the transaction's execution. In addition, the compiler automatically generates code implementing the two methods of the IRecoverable interface, *Backup* and *Restore*, which copies the value of the "real" fields to the "shadow" fields and vice versa. Backup, which is invoked on the first attempt to write to an atomic object in the current transaction, copies the real fields to the shadow fields. Restore, which is invoked when the previous writer to the object is found to have aborted, copies the last known good values from the shadow fields back to the

real fields. *Const*, *final* (or *readonly)*, and *TxSafe*-annotated fields are considered idempotent, so no shadow fields need to be generated for them.

In this model, we optimize the OpenRead and OpenWrite methods to acquire a lock only when absolutely necessary. Before acquiring the lock, then, all possible conditions that can be handled without a lock are dealt with first. For example, in the OpenRead algorithm, if the reading transaction also turns out to be the "owner" (the current writer) of the object, the read can proceed in a completely non-blocking fashion. After the desired field is read, the algorithm simply checks to see whether the current transaction has been aborted. This is required because it is possible that the current transaction was aborted prior to the field read, in which case another transaction may already have modified the field's value, and returning the value written by another "in progress" transaction would violate the isolation property. While some STMs, such as the McRT system [74], permit so-called dirty reads in an effort to lower performance overheads, our model does not. The principle that all transactions, even zombies, see a consistent state is important for the overall safety of STM systems. Without such a guarantee, there is no provably safe model for ensuring that transactions do not enter an infinite loop or otherwise perform an incorrect operation as a result of seeing an intermediate value. The pseudocode below shows the basic operation of the OpenRead algorithm. For the complete source code in C#, see the Appendix.

```
OPEN-READ(me, field, txObject)
1   if me = txObject.writer          ▷ am I already the writer?
2       then value ← field           ▷ do the read
3           CHECK-FOR-CONSISTENCY(me)
4           return value
    ▷ post-state: I'm not the writer
5   RetryLabel:
6   CHECK-FOR-CONSISTENCY(me)
```

```
7   lock(txObject)
8   switch txObject.writer.state
9       case ACTIVE
10          unlock(txObject)
11          RESOLVE-CONFLICT(me, txObject.writer)
12          goto RetryLabel
13      case ABORTED
14          RESTORE(txObject)
        ▷ don't need to check for COMMITTED, because we're only reading
15  value ← field          ▷ do the read
16  REPLACE-INACTIVE-READERS(me, txObject)
17  MAINTAIN-READERS-ARRAY(me, txObject)
18  unlock(txObject)
19  return value
```

```
CHECK-FOR-CONSISTENCY(me)
1   if me.state = ABORTED
2       then throw ABORTED-EXCEPTION
```

```
REPLACE-INACTIVE-READERS(me, txObject)
    ▷ replaces first non-active reader with the current reader
1   for i ← 0 to txObject.numReaders
2       currReader ← txObject.readers[i]
3       if currReader.state ≠ ACTIVE or me = currReader
4           then txObject.readers[i] ← me
```

```
MAINTAIN-READERS-ARRAY(me, txObject)
    ▷ if necessary, resize the readers buffer
1   if txObject.numReaders = txObject.readers.Length
2       then RESIZE-ARRAY(ref txObject.readers, txObject.readers.Length * 2)
3   txObject.readers[txObject.numReaders] ← me
```

Assuming that the active transaction was not aborted, we can be certain that the value read was correct and can be returned to the user's code. This is correct even where, by the time the value is returned to the user's code, the transaction has been aborted and the field overwritten with a new value. Consistency does not require that we report the current value of the field. In fact, quite the opposite: transactions require that the client

see only those values that are consistent with some linearizable execution of the active transactions (refer to steps 2 and 3 of the OpenRead algorithm shown above). These insights into the blocking model makes it possible for the compiler to inline fast-path versions of the read and write algorithms that do not require locks, where it is certain that the object has already been opened in read or write mode.

The only possible exception that might be thrown here is a null reference exception, if the atomic object reference itself is null. This behavior is identical to that seen without an STM, and is therefore considered correct. Note that there is no danger of a lock being held indefinitely, since no lock is acquired by the subsequent read algorithm. In the case of subsequent reads, executed after an earlier read or write, the inlined code need only determine whether the value read is valid. If the transaction wasn't aborted after the time the value was read, then we can safely return the value to the user's code (note that no CAS operation is required):

INLINED-SUBSEQUENT-READ(*me*, *target*)
1 *stack* ← *target.field*
2 CHECK-FOR-CONSISTENCY(*me*)

The .NET memory model does not support sequential consistency, generally defined as requiring that "the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."[3] In fact, the official ECMA-335 specification[4] for .NET guarantees only the following basic semantics:

---

[3] Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Trans. Comput. C-28,9 (Sept. 1979), 690-691.
[4] http://www.ecma-international.org/publications/standards/Ecma-335.htm

1. Read or writes for a given location cannot pass a write for the same location.
2. Reads cannot move before acquisition of a lock.
3. Writes cannot move after release of a lock.
4. Other reads/writes may be reordered, except before or after a volatile read/write.

However, all commercial implementations of .NET provide further guarantees, although still not sequential consistency. In addition to the ECMA-335 rules described above, .NET 2.0 and above provide that:

5. Reads and writes cannot be introduced.
6. Reads can only be removed where they are adjacent to another read for the same location from the same thread; writes can only be removed where they are adjacent to another write for the same location from the same thread. Rule 8 can be used to make reads and writes adjacent before applying this rule.
7. Writes cannot move past other writes from the same thread.
8. Reads can move earlier in time, but never past a write to same location from the same thread.

This is still weaker than traditional x86 behavior (with the possible exception of IA-64), which also provides that:

9. Writes can only move later.
10. Writes cannot move past a read for the same location from the same thread.
11. Reads can only move later to stay after a write, to keep from breaking rule 10, as that write moves later.

There are three levels at which such optimizations can occur: the compiler, the execution environment, and the processor. Because Peet is a backend compiler, it doesn’t have to be concerned about the C# compiler affecting its optimizations. Furthermore,

were STM support integrated into the execution environment, we also would not need to worry about the runtime.

Turning now to the STM optimizations of the Peet compiler, only one optimization suffers from the possibility of a data race, where one thread reads a memory location while another thread may be modifying it: the inlined subsequent read algorithm in the blocking STM model. The lock-free subsequent read algorithm requires that the two read operations for disjoint memory locations (shown below in steps 1 and 2) be executed in the order in which they are encountered by the thread and that the transaction's *state* field is current (rather than a cached value). Reversing the order of the two steps (perhaps by caching the value of the transaction's state) would break the algorithm. Since Peet is a backend compiler executed after the C# compiler, we need not worry about interference from compiler optimizations. However, though not currently implemented in the commercial .NET runtime, the ECMA specification leaves open the possibility that the execution environment itself might perform such optimizations. Thus the runtime could theoretically cache the value of the transaction's *state* field.

1 *stack* ← *target.field*
2 **if** *me.state* = ABORTED
3 **then throw** ABORTED-EXCEPTION

Note that we don't have to worry about a processor cache storing old copies of the transaction's state, since it is always updated with a CAS operation, which provides strong ordering guarantees. In particular, no reads or writes can move in either direction past a CAS operation. This, in part, is what makes CAS operations so expensive: not only does the executing processor need to ensure that no other processor is trying to execute a CAS on the same location at the same time, but, in order to provide the ordering

guarantees, the processor needs to ensure that caches are synchronized so that reads and writes don't seem to move past the CAS operation.

Given the semantics of transactional memory, another thread that wants to write to an atomic object being read or written by another transaction will first need to execute a CAS operation to abort the first transaction, as shown below. If successful, the CAS instruction operates as a cross-thread memory barrier, effectively synchronizing the view of all threads for this location.

CAS(*conflictingTx.state,* ABORTED*,* ACTIVE)

One solution to the problem of read operation reordering is to introduce a memory barrier (Thread.MemoryBarrier() in .NET) after the first read in step 1, as shown below. The memory barrier will synchronize memory such that the processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the barrier execute after accesses following the barrier.

```
1   stack ← target.field
2   MEMORYBARRIER
3   if me.state = ABORTED
4       then throw ABORTED-EXCEPTION
```

While the memory barrier ensures that the order of the two read instructions is not swapped, it still doesn't solve the problem of the runtime potentially caching an old copy of the transaction's state. The only way to definitively solve the problem is to introduce a volatile read for the transaction's state, as shown below. Volatile reads have "acquire semantics" meaning that the read is guaranteed to occur prior to any references to

memory that occur after the read instruction in the bytecode sequence. However, that doesn't obviate the need for a memory barrier, since a load with acquire semantics only creates a downward fence, so although normal loads and stores cannot be moved above the volatile read, they can still be moved below it.

```
1   stack ← target.field
2   MEMORYBARRIER
3   if VOLATILEREAD(me.state) = ABORTED
4       then throw ABORTED-EXCEPTION
```

Through the use of the memory barrier and the volatile read operation, Peet need make no assumptions about the memory model beyond ECMA-compliance. For performance reasons and because Peet is a post-compiler that doesn't need to worry about interference from C# optimizations, on the x86 architecture these barriers can be omitted through the use of a compiler flag. All the other optimizations of the Peet compiler are data race-free: in the blocking STM model subsequent writes acquire counting "locks" with the CAS operation; in the nonblocking model, all readers share access to an object, but writers get their own private copy.

Formally, we use *rely-guarantee* reasoning of the form proposed by Herlihy and Hoare in [84, 85] to prove that our optimizations are linearizable, safe, and that they correctly implement the high-level abstraction. In *rely-guarantee* (R-G) reasoning, each thread is assigned a *rely* condition that characterizes the interference that thread can tolerate from other threads. In return, the thread is assigned a *guarantee* condition that characterizes how that thread can interfere with the others. Proving the safety of the program requires proving that (1) if each thread's rely condition is satisfied, then the

thread satisfies its guarantee condition, and (2) each thread's guarantee condition implies the others' rely conditions.

Classically, a single atomic action is specified by a pair of predicates ($p$, $q$), where $p$ is the pre-condition assumed to hold when the action starts, and $q$ is the post-condition established if and when the action terminates. We write $C \models (p, R, G, q)$ for the judgment saying that the program $C$ meets the R-G specification ($p$, $R$, $G$, $q$). Atomic actions are denoted by enclosing a program in diamond brackets $\langle C \rangle$. The corresponding proof rule is:

$$\frac{\{p\}\ \mathrm{C}\ \{q\}}{\mathrm{C} \models (p, \mathrm{Preserve}(p), q \vee \mathrm{ID}, q)}$$

This rule says that the pre-condition and the post-condition remain sequential; a thread guarantees that it either does $q$ or nothing at all, and all other threads must preserve the precondition. The implementation of atomicity ensures that such interference cannot take place within the diamond brackets, so the proof of correctness of the atomic region is unaffected by interference.

*Definitions*: We now give the definitions of the subsequent read invariant and the associated rely and guarantee conditions; then we will prove the algorithm obeys their specifications. The subsequent read invariant is defined as follows:

$$\mathit{SubReadInv} \equiv \forall_{\mathrm{Transaction}}\ \mathrm{tx}.\ \mathit{tx.state} \neq \mathrm{ACTIVE} \Rightarrow \mathrm{ABORT}(\mathit{tx})$$

The pre-condition is: $p \equiv \mathit{SubReadInv}$

$$\wedge\ (\mathit{me} \in \mathit{target.Locator.readers} \vee \mathit{me} = \mathit{target.Locator.writer})$$

Note that the pre-condition does not require that $\mathit{target} \neq \mathrm{NIL}$; this is only a requirement of the post-condition. Obviously, if target = NIL, a null reference exception will be thrown in statement 1 of the the inlined subsequent read algorithm. Although somewhat counterintuitive, the target object may be null on a subsequent read because preOpen operations silently ignore null atomic object references; see § 4.2.2 for details.

The rely condition is: $R \equiv \text{Preserve}(SubReadInv)$

The guarantee condition is: $G \equiv \text{Preserve}(SubReadInv)$

The post-condition is: $q \equiv SubReadInv \wedge target \neq \text{NIL}$
$\wedge\ (me \in target.Locator.readers \vee me = target.Locator.writer) \wedge stack \leftarrow target.field$

*Proof*: We assume that the atomic object has already been read and/or written previously (at the initial read, a lock is required to update the transactional object's state atomically—see the pseudocode for the OpenRead algorithm above and the Appendix for the complete C# definition). The *rely* and *guarantee* properties are enforced by an explicit consistency check that preserves the *SubReadInv*. Once the consistency check succeeds, the algorithm has executed safely with the *rely* condition maintained throughout. This, in turn, satisfies the post-condition.

$\{\ me \in target.Locator.readers \vee me = target.Locator.writer\ \}$
INLINED-SUBSEQUENT-READ(*me*, *target*)
1 $stack \leftarrow target.field$
$\{\ target \neq \text{NIL} \wedge (me \in target.Locator.readers \vee me = target.Locator.writer) \wedge stack \leftarrow target.field\ \}$
2 CHECK-FOR-CONSISTENCY(*me*)
$\{\ SubReadInv \wedge target \neq \text{NIL} \wedge (me \in target.Locator.readers \vee me = target.Locator.writer)$
$\wedge\ stack \leftarrow target.field\ \}$

Another way of reasoning about this proof is to assume that the atomic object has already been read by this transaction at time $t_n$ and that therefore the SynchState data of the object reflects that prior read. The inlined read happens at time $t_{n+1}$ (step 1 above), but the value read, which is not guaranteed to be safe, is only stored on the stack and not returned to the client algorithm until the transaction is checked for consistency at time $t_{n+2}$ (step 2 above). If the transaction is still consistent (i.e., active) at time $t_{n+2}$, then it follows that the transaction was also active at the earlier time $t_{n+1}$ when the atomic field was read and its value stored on the stack (transactions can only move from the active to

the committed or aborted states, never back). Thus the value on the stack is linearizable with respect to the running transaction and can be returned to the client algorithm. If, on the other hand, the transaction is no longer active at time $t_{n+2}$, then the value read at time $t_{n+1}$ may or may not be correct, depending on whether the transaction in fact aborted before or after the read. Since the correctness of the value is in doubt, the read value is popped from the stack, and an AbortedException is thrown.

Unlike subsequent reads, the algorithm for subsequently writing to a field of an atomic object cannot proceed entirely without a CAS operation. In the case where the atomic object is already owned by the writing transaction (i.e., a subsequent write), we must ensure that the transaction is not aborted prior to the completion of the actual write operation. Unfortunately, checking whether the transaction has been aborted before the write is not effective, since an abort might occur asynchronously immediately after the check but before the write; checking the transaction's status after the write is too late—another transaction's value might have already been overwritten. The OpenWrite algorithm thus must always acquire exclusive access before writing, as shown in the pseudocode below (the complete algorithm in C# is available in the Appendix).

```
OPEN-WRITE(me, field, newValue, txObject)
1   if me = txObject.writer          ▷ am I already the writer?
2       then lock(txObject)
3           field ← newValue         ▷ do the write
4           unlock(txObject)
5           return
    ▷ post-state: I'm not the writer
6   RetryLabel:
7   CHECK-FOR-CONSISTENCY(me)
8   lock(txObject)
9   if CHECK-FOR-READ-CONFLICTS(me, txObject) = TRUE
10      then goto RetryLabel
11  txObject.numReaders ← 0          ▷ reset number of readers
12  switch txObject.writer.state
```

```
13      case ACTIVE
14          unlock(txObject)
15          RESOLVE-CONFLICT(me, txObject.writer)
16          goto RetryLabel
17      case COMMITTED
18          BACKUP(txObject)
19      case ABORTED
20          RESTORE(txObject)
21  txObject.writer ← me            ▷ install myself as the writer
22  field ← newValue                ▷ do the write
23  unlock(txObject)
```

```
CHECK-FOR-READ-CONFLICTS(me, txObject)
    ▷ am I in conflict with any other readers?
1   for i ← 0 to txObject.numReaders
2       currReader ← txObject.readers[i]
3       if currReader.state = ACTIVE and me ≠ currReader
4           then unlock(txObject)
5               RESOLVE-CONFLICT(me, currReader)
6               return TRUE
7   return FALSE
```

However, actually acquiring a lock on the object before subsequent writes is not required, since the danger here is that the transaction might be aborted before or during the write operation (after is fine). To address this problem, inlined subsequent write operations must ensure that the transaction cannot be aborted during the update. By incrementing a special transaction-wide abortCounter, a fast-path write prevents the transaction from being aborted during this critical period. The algorithm is wait-free, but two CAS operations are required per write:

```
INLINED-SUBSEQUENT-WRITE(me, target, value)
    ▷ make sure target is valid
1   if target = NIL
2       then throw NULLREFERENCEEXCEPTION
    ▷ make sure we can't be aborted now
3   if ATOMIC-INCREMENT(me.abortCounter) < 1
4       then throw ABORTED-EXCEPTION        ▷ we've already been aborted!
5   target.field ← value                    ▷ do the update
6   ATOMIC-DECREMENT(me.abortCounter)       ▷ release the lock
```

*Definitions*: We now give the definitions of the subsequent write invariant and the common rely and guarantee conditions; then we will prove the algorithm obeys their specifications. The subsequent write invariant is defined as follows:

$$SubWriteInv \equiv \forall_{\text{Transaction}}\ \text{tx}.\ tx.state \neq \text{ACTIVE} \Rightarrow \text{ABORT}(tx)$$

The rely condition is: $R \equiv \text{Preserve}(SubWriteInv)$
$\wedge \forall_{\text{Transaction}}\ \text{tx}.\ tx.abortCounter < 0 \Rightarrow \text{ABORT}(tx)$
$\wedge\ tx.abortCounter \geq 0 \Rightarrow \text{ID}(tx.abortCounter)$

This rule says that we rely on any transaction whose *abortCounter* value is less than zero to abort, and that while a transaction's *abortCounter* is greater than or equal to zero the before and after values of *abortCounter* will be the same.

The guarantee condition is: $G \equiv \text{Preserve}(SubWriteInv)$
$\wedge \forall_{\text{Transaction}}\ \text{tx}.\ tx.abortCounter < 0 \Rightarrow \text{ABORT}(tx)$
$\wedge\ tx.abortCounter \geq 0 \Rightarrow \text{ID}(tx.abortCounter)$
$\wedge\ tx.abortCounter > 0 \Rightarrow \neg\text{ABORT}(tx)$

This rule says that we guarantee that if our *abortCounter* value is less than zero we will abort; that while our *abortCounter* is greater than or equal to zero the before and after values of *abortCounter* will be the same; and that while our *abortCounter* is greater than zero, the transaction will not abort.

The pre-condition is: $p \equiv SubWriteInv \wedge target \neq \text{NIL} \Rightarrow me = target.Locator.writer$

The post-condition is: $q \equiv SubWriteInv \wedge target \neq \text{NIL} \wedge me = target.Locator.writer$
$\wedge\ target.field \leftarrow newValue$

*Proof*: The final post-condition is the post-condition defined for the algorithm as a whole, and except where the consistency check in step 1 fails, the algorithm satisfies the rely condition defined above. Below is the inlined subsequent write algorithm annotated with atomicity assumptions:

INLINED-SUBSEQUENT-WRITE(*me*, *target*, *newValue*)
$\{\ target \neq \text{NIL} \Rightarrow me = target.Locator.writer\ \}$
1 **if** *target* = NIL
2 **then throw** NULLREFERENCEEXCEPTION
$\{\ target \neq \text{NIL} \wedge me = target.Locator.writer\ \}$
3 **if** ATOMIC-INCREMENT(*me.abortCounter*) < 1

{ $target \neq$ NIL $\wedge$ $me.abortCounter$ = INTMINVAL + 1 $\Rightarrow$ violation of $SubWriteInv$ }
4 **then throw** ABORTED-EXCEPTION
{ $SubWriteInv \wedge target \neq$ NIL $\wedge$ $me = target.Locator.writer \wedge me.abortCounter > 0$ }
5 $target.field \leftarrow newValue$
{ $SubWriteInv \wedge target \neq$ NIL $\wedge$ $me = target.Locator.writer \wedge me.abortCounter > 0$
$\wedge\ target.field \leftarrow newValue$ }
6 ATOMIC-DECREMENT($me.abortCounter$)
{ $SubWriteInv \wedge target \neq$ NIL $\wedge$ $me = target.Locator.writer \wedge target.field \leftarrow newValue$
$\wedge$ ID($me.abortCounter$) }

The algorithm for subsequent writes specifically checks for a null reference so that this exception cannot be thrown after the abortCounter is incremented (an exception thrown at this point would cause the corresponding decrement to be skipped, leading to deadlock). Since subsequent writes occur after an initial write, one might assume that a null reference exception would be thrown at that earlier point. However, because pre-open operations (described in § 4.2) silently ignore null references, it is possible that a null reference will be deferred until a subsequent read or write operation. Note that the algorithm for inlined subsequent writes does not violate, but also does not explicitly enforce, the *guarantee* condition requiring that while this transaction's *abortCounter* is greater than zero, the transaction will not abort. This part of the *guarantee* condition is satisfied by the Abort method, which is where the actual waiting, if any, occurs. Abort must check whether a transaction's abortCounter is zero before aborting it. While it is not the Abort method blocks in a while loop, satisfying the *guarantee* condition, as shown here:

ABORT($me$)
1 **while** CAS($me.abortLock$, INTMINVAL, 0) > 0
▷ loop until lock is released (i.e., while $me.abortCounter > 0$)
2 **end-while**
{ $me.abortCounter < 0 \Rightarrow$ ABORT($me$) }

Because an abort "lock" (in the form of a CAS'ed integer counter) is acquired by the subsequent write algorithm, the abort algorithm is plainly atomic with respect to abort requests (i.e., they can happen before or after, just not during). Therefore, the argument for its correctness will focus on the safety of this algorithm with respect to that lock. Since only three instructions, a local load, an object reference load, and a field store, are executed while the lock is held, these instructions are scrutinized for possible exceptions that might be thrown, thereby preventing the release of the lock. First, we consider the possibility of an overflow exception at the local load instruction; however, this is not a concern because such an exception would be generated earlier, when the value is first stored in the temporary local. Second, we consider the possibility of a null reference exception at the field store instruction; by guaranteeing that subsequent writes occur after a full write operation, the compiler ensures that the atomic object itself is valid, and therefore that no exception can be thrown from within the locked region. This concludes our arguments for the correctness of the inlined subsequent read and subsequent write code sequences in the blocking transactional memory model.

Figure 4.2 compares the performance of several standard benchmark algorithms as implemented with a library-based STM and as optimized by the compiler[5]. Note in particular that the HashTable benchmark uses large arrays and thus benefits greatly from the hybrid array implementation described in § 4.7. The STM-only results give the performance for the un-optimized SXM library [41]; Library Opts shows the improvements of our optimized transactional synchronization algorithms; the Compiler bar shows the results of inlining and automating the generation of STM code; Const +

---

[5] These results were obtained from 30 second runs using 1 thread, 30% modifiers, and the aggressive contention manager on a 4-way machine.

Local is the result of the dataflow analysis that eliminates STM accesses for those objects that are provably local; RWPromo includes the optimization that promotes read accesses to write mode open operations for atomic objects that are both read and written; the Subsequent optimization exploits our understanding of the difference between the first and subsequent accesses of an atomic object (see § 4.2); finally, the PRE optimization shows the results of our full interprocedural PRE analysis. Note that the bars are cumulative (i.e., each more advanced optimization includes all the prior ones).

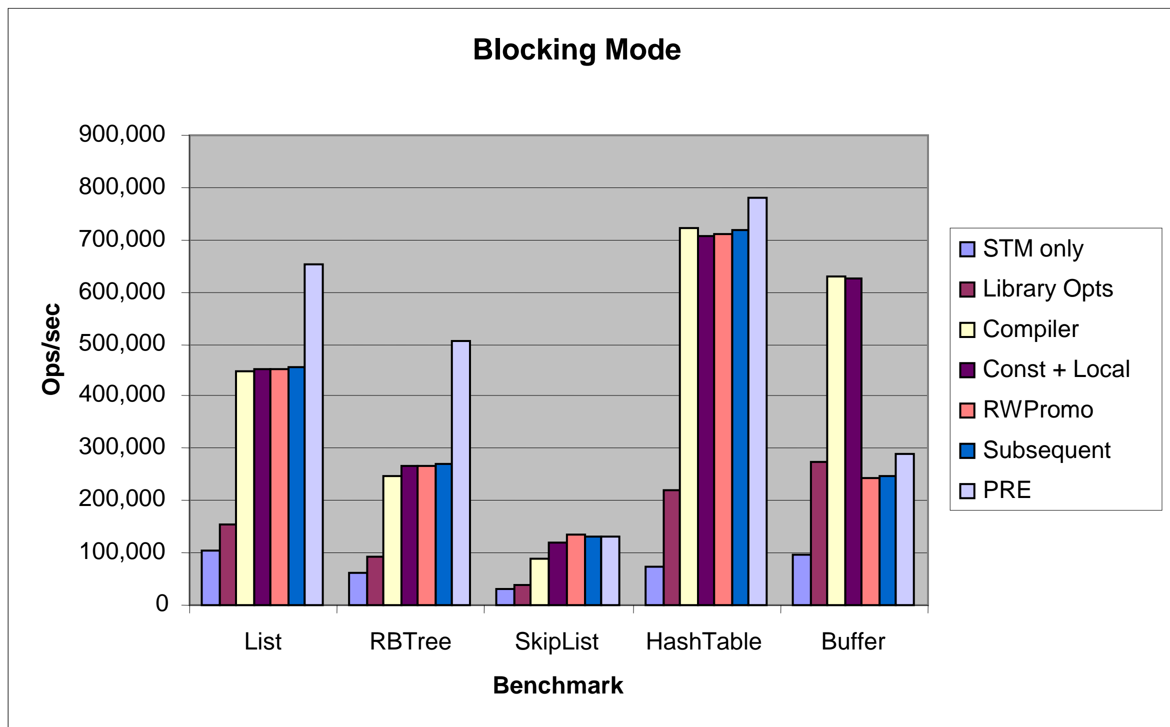



Figure 4.2: Blocking mode results

## 4.4 Obstruction-free optimizations

The compiler optimizations implemented for the obstruction-free mode are slightly different from those that apply to the blocking STM. In the obstruction-free model, in place of shadow fields and the Backup/Restore methods of the IRecoverable

interface, the compiler implements the ICloneable interface's Clone method to create a new, speculative, copy of the object. Interestingly, the obstruction-free STM cannot support static fields, since they cannot be cloned[6] in an object-based STM. Instead, the compiler manages static fields of an atomic object as in the blocking mode, where their values are copied to static shadow fields via the backup/restore methods of the IRecoverable interface.

In its obstruction-free mode, the compiler generates local variables to cache the "open" version of atomic objects. In contrast to the blocking mode, subsequent reads do not need to check whether the transaction has been aborted, and subsequent writes need not temporarily delay aborts during the write. This means that subsequent reads and writes are completely inlined and have no transactional overhead whatsoever. These optimizations are possible because the obstruction-free STM creates clones of atomic objects, such that a subsequent read or write may see or update an inactive copy of the object, but linearizability is not compromised.

Technically, each obstruction free atomic object contains a reference to an OFreeLocator record, shown below. If there is no writing transaction (identified by OFreeLocator.writer), the OFreeLocator.newObject field always points to the current version of the atomic object for read sharing. Thus, so long as there is no writing transaction, readers share access to the current copy of the atomic object. If a transaction wants to open the object for writing, it must first create a new OFreeLocator object, abort any readers and/or writers in the old locator, clone the atomic object, store the cloned object reference in OFreeLocator.newObject, and save the previous version of the object in OFreeLocator.oldObject. Finally the old OFreeLocator reference itself is replaced by

[6] The whole point of static fields is that only one copy exists per class. Thus cloning an object has no effect.

the new locator in a single, atomic CAS operation. If the CAS operation is successful, the new version of the atomic object becomes the speculative copy used by the current transaction until it commits or aborts.

```
class OFreeLocator {
    XState writer = null;
    XState reader = null;
    OFreeLocator next = null; // linked list of readers
    ICloneable oldObject = null;
    ICloneable newObject = theAtomicObject;
}
```

The argument for the safety and linearizability of this approach is as follows. Each obstruction-free atomic object is immutable. Every new transaction that intends to write to an atomic object first creates a private copy solely for its use. Only if the transaction successfully commits, does that copy become visible to subsequently started transactions. Critically, any client code that may hold a cached reference to an earlier version of the atomic object may safely continue to use that version of the object. Once another writing transaction takes ownership of the atomic object via the OFreeLocator data, any still active transactions that previously opened the object are doomed to abort. Nevertheless, because they hold valid references to an earlier version of the atomic object (which have now been disconnected from the OFreeLocator), they may safely (linearizably) continue reading or writing that version of the object until they abort. We rely on the garbage collector to destroy doomed copies of atomic objects once the transaction aborts. First (i.e., not subsequent) accesses include a check to verify that the transaction is still healthy, aborting if it is not. This concludes the argument for the correctness of our approach to subsequent reads and subsequent writes in the obstruction-free transactional memory model.

Figure 4.3 shows the performance results of the compiler using the obstruction-free STM mode. Overall, the graphs show that the obstruction-free STM performs slightly better than the blocking mode using short-lived locks.

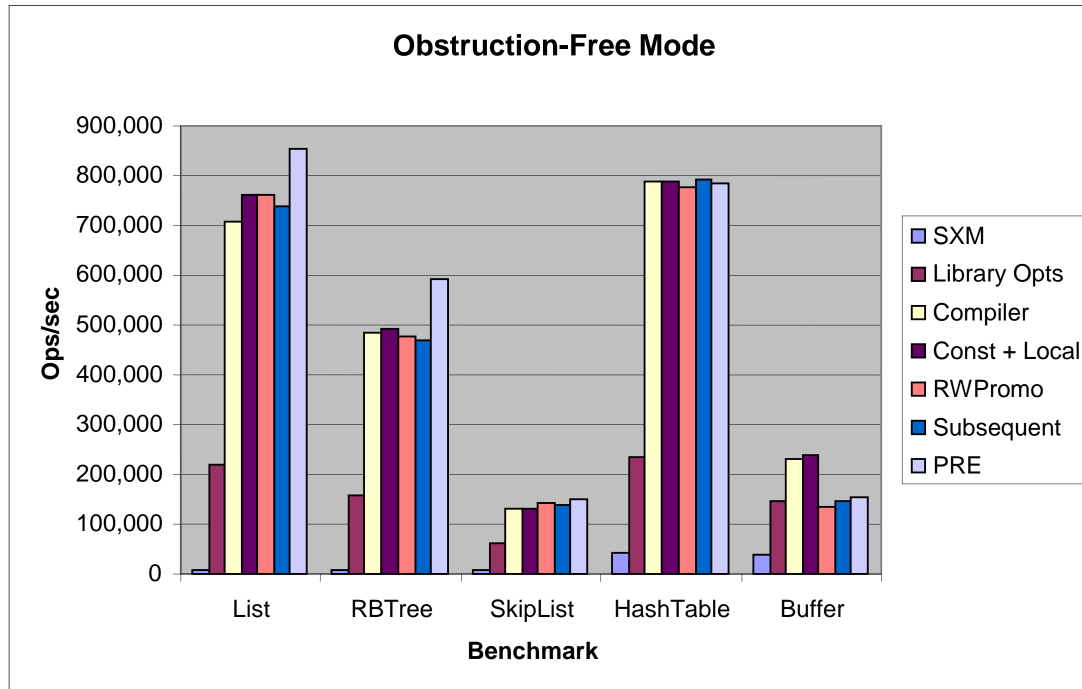



Figure 4.3: Obstruction-free results

For comparison with longer running transactions, we also tested these optimizations with the three STAMP benchmarks [66], genome, kmeans, and vacation. Figure 4.4 shows the performance results for each benchmark at increasing levels of optimization. All the timing results were normalized to 1.0 for the optimized SXM library.

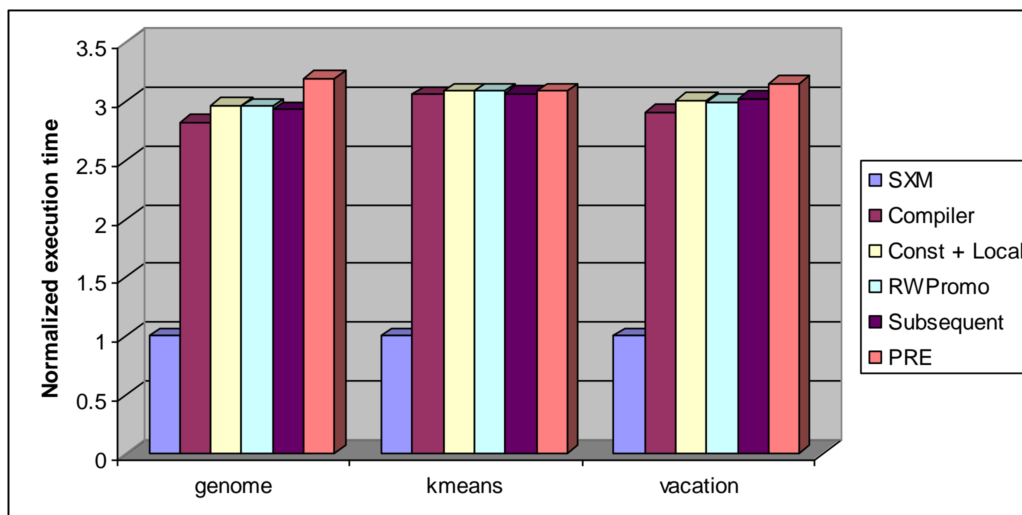



Figure 4.4: STAMP benchmark performance

The performance results reported in the thesis (see, e.g., Figures 4.2 through 4.4) were obtained from averaged thirty second runs using one thread, 30% write operations, and the aggressive contention manager on a four-way hyper-threaded machine. Because most of the overhead in transactional memory consists of base line overhead imposed whether the client is running one thread or many threads, we are primarily interested in showing the lowered overhead of a single transaction.

Tests of one to eight threads executing concurrently (see, e.g., Figure 5.8) are not impressive for several reasons: first, the simple data structure benchmarks offer little opportunity for write concurrency (some, like the integer list, offer none whatsoever); second, our primary test machine is a four-way SMP machine, which, while ideal for stress testing concurrent code, is not a multicore processor, and therefore exhibits relatively high overheads due in part to cache synchronization.

Far more promising results are obtained on an Intel Core 2 Duo processor (see table below). With no writes (and therefore no contention), two threads showed roughly

1.6 times the throughput. As would be expected, the same benchmarks run using course-grained locking performed slightly worse with two threads than with one. Assuming the same rate of improvement will be found on four-core and eight-core processors, the crossover point at which STM will perform about as fast as course-grained mutual exclusion will be a four-core processor. An even more intriguing result, however, is the fact that the STM mode that uses short locks does not show even close to the performance improvement as that shown by the nonblocking STM mode from one to two threads; a further argument in favor of nonblocking models.

| | 1 thread | 2 threads | 4 threads (predicted) | 8 threads (predicted) |
|---|---|---|---|---|
| ListSTM (nonblocking) | 1,276,000 | 2,028,000 | 3,223,000 | 5,123,000 |
| ListMutex | 3,755,000 | 3,752,000 | 3,749,000 | 3,746,000 |
| ListSTM (blocking) | 948,000 | 956,000 | 964,000 | 972,000 |
| RBTreeSTM (nonblocking) | 1,079,000 | 1,778,000 | 2,930,000 | 4,828,000 |
| RBTreeMutex | 3,036,000 | 3,011,000 | 2,986,000 | 2,961,000 |
| RBTreeSTM (blocking) | 771,000 | 787,000 | 803,000 | 820,000 |

## 4.5 Retry semantics

The retry semantics offered by the SXM library have also been optimized for use by the Peet compiler. The BlockWhileActive method, shown in pseudocode below, implements the Retry functionality. Basically, the BlockWhileActive method sleeps until the current transaction is aborted, notifying it that another transaction has modified something in its read/write set. Importantly, the BlockWhileActive algorithm sets a waiting flag that notifies contention managers that this transaction is waiting to be aborted. This prevents polite contention managers from needlessly waiting for this transaction to make progress.

```
BLOCK-WHILE-ACTIVE(me)
1  ▷ suspend calleer until aborted by another transaction—implements "retry"
2  me.waiting ← TRUE          ▷ notify contention manager that we are waiting
3  THREAD-SLEEP(0)            ▷ yield
4  if me.state ≠ ABORTED
5      then me.signal ← new EVENT()
6           while me.state ≠ ABORTED
7               WAIT-SIGNAL(me.signal)
8  me.waiting ← FALSE
```

The optimized version of BlockWhileActive first tries to yield its quantum by calling Thread.Sleep(0). In the case of short transactions, this is sufficient to allow another transaction to abort this transaction and make progress. However, in the case where one time slice is not sufficient, a signal is created. This allows the BlockWhileActive method to suspend the current thread until it is awaken by an abort. As part of the Abort() method, the signaling event is set, if it is non-null (indicating that another transaction is waiting on the event):

```
ABORT(me)
1  ▷ …
2  if me.signal ≠ NIL
3      then SET-SIGNAL(me.signal)
```

The code generated for atomic using blocks (i.e., using(new XAction()) { }) is conceptually very similar to that of atomic methods. However, because an abort causes the entire method to be re-executed, thereby discarding and reevaluating all local variables, the generated code must save the value of all local variables that might be used in the atomic block before the transaction begins, and then restore their value in the event of an abort. Luckily, because local variables are private to the thread, there is no danger of concurrent access during the backup or restore procedure. However, any non-local and

non-atomic objects that are accessed from within an atomic block cannot be handled safely, and thus they are not permitted in atomic *using* blocks.[7]

As described in § 3.4, atomic blocks can be executed based on a boolean condition. Peet automatically moves the evaluation of the condition inside the transaction's scope. If the condition evaluates to false, the BlockWhileActive() method is called to wait until the transaction is aborted.

## 4.6 The CASe primtive

The complexity involved with implementing a wait-free subsequent write in the blocking STM mode led us to identify a potential use for short hardware transactions. While much previous work has focused on the feasibility of efficiently implementing various types of multi-compare and swap (MCAS) operations [56], we observe that a very slight modification to the existing CAS primitive would greatly simplify the implementation of subsequent write operations in object-based STM systems: simply atomically comparing one location (the transaction's status) and swapping another (the target field) would obviate the need for our approach to delaying aborts during write operations. We call this new primitive Compare and Swap elsewhere (CASe). Its only real added complexity, from a hardware perspective, is the fact that the compare location and the swap location might fall in different cache lines, necessitating that both lines be locked.

This is the inlined subsequent write code sequence presently used by the compiler:

---

[7] Local variables are not a problem in atomic methods (in contrast to atomic blocks), because C# requires that they be explicitly initialized prior to first use. Thus, after an abort and restart, the locals will be reinitialized.

```
INLINED-SUBSEQUENT-WRITE(me, target, value)
     ▷ make sure target is valid
1    if target = NIL
2         then throw NULLREFERENCEEXCEPTION
     ▷ make sure we can't be aborted now
3    if ATOMIC-INCREMENT(me.abortCounter) < 1
4         then throw ABORTED-EXCEPTION        ▷ we've already been aborted!
5    target.field ← value                     ▷ do the update
6    ATOMIC-DECREMENT(me.abortCounter)        ▷ release the lock
```

With the proposed CASe primitive, the algorithm for subsequent writes shown above can be rewritten as shown here:

```
INLINED-SUBSEQUENT-WRITE(me, target, value)
1    if ¬CASE(me.state,          ▷ compare A
              ACTIVE,            ▷      with this
              target.field,      ▷ swap B
              value)             ▷      with this
2         then throw ABORTED-EXCEPTION
```

## 4.7 Array algorithms

One aspect of the runtime system that turned out to be a significant performance bottleneck for some benchmarks was the array implementation provided by the STM library. In the initial release of SXM [41], the basic array support was not typed (all array elements were of type *object*), and the backup and restore operations performed complete copies of the array, as shown below:

```
public class OriginalAtomicArray :
    IEnumerable, IRecoverable {
    object[] data;   // new data
    object[] shadow; // backup data

    public void Restore() {
        Array.Copy(shadow, 0, data, 0, data.Length);
    }

    public void Backup() {
        Array.Copy(data, 0, shadow, 0, data.Length);
```

```
    }
    ...
```

To optimize the performance of array-based data structures, we investigated several alternate array implementations. First, we parameterized the atomic array type using generics so that casts are not needed when reading or writing array elements. Second, in addition to the basic copying array described above, we developed a number of experimental array algorithms that are more efficient in terms of space or time.

One possible approach is to keep a log of array changes rather than simply duplicating the entire array with a backup version. This approach has much in common with the transactional boosting technique described in § 3.8, as it basically involves keeping a log of read and write operations to the array and then undoing all the write operations in the event the transaction aborts. Not surprisingly, the log array is typically more space efficient than a copying array. It defines a list of ArrayElement structures that store the value and index number of an array entry:

```
internal struct ArrayElement<T> {
    public readonly int index;
    public readonly T value;

    public ArrayElement(int index, T value) {
        this.index = index;
        this.value = value;
    }
}
```

The log array stores the backup log in a list of array elements. To limit the maximum size of the backup log to the size of the array, a bit array is used to determine whether the client has accessed this array element previously, as shown below:

```
private readonly T[] data;
private readonly List<ArrayElement<T>> arrayLog;
private readonly BitArray containsArray;
```

Each array write operation first checks the bit array to see whether the user has modified the specified element previously. If the index's bit is not set, the array element must be added to backup list; if the index's bit is set, the array element has already been backed-up previously. The main indexer algorithm is shown below:

```
// principal indexer
public T this[int index] {
   get {
      // reads go forward
      return data[index];
   }

   set {
      // check the bit array
      if(!containsArray[index]) {
         // log the old value
         arrayLog.Add(
             new ArrayElement<T>(index, data[index]));
         // mark the bit array
         containsArray[index] = true;
      }

      // do the update
      data[index] = value;
   }
}
```

For the log array, the backup operation simply clears the bit array and empties the backup log. The actual backup of array elements occurs lazily, on the first use of a specific index. The restore operation, however, called when a transaction aborts, must iterate the entire backup log, restoring the original values of all the modified elements. The backup and restore algorithms for the log array are shown below:

```
public void Backup() {
    // empty the log
    arrayLog.Clear();

    // clear the bit array
    containsArray.SetAll(false);
}

public void Restore() {
    // restore based on tx's log
    foreach(ArrayElement<T> le in arrayLog) {
        data[le.index] = le.value;
    }
}
```

The log array is more efficient than the copying array when accessing larger arrays, since the entire array is not copied on each backup operation (a far more common operation than a restore). However, when working with small arrays consisting of several elements, copying the array is can be more efficient compared with the overhead of creating the backup log in a piecemeal fashion as part of every update. The hybrid array combines the strengths of both approaches. For arrays consisting of ten elements or fewer, the hybrid array simply creates a copy of the entire array on each backup/restore. Larger arrays are handled in the manner of the log array. This array algorithm provides the best overall performance, although it does require a runtime check to determine whether it is in copying or logging mode.

Figure 4.5 compares the performance the copying, logging, and hybrid approaches in the context of the SkipList and HashTable benchmarks. Because the HashTable benchmark uses large arrays, the logging array algorithm works best overall, but the hybrid approach is a close second. In the case of the SkipList benchmark, which uses relatively small arrays, the copying algorithm outperforms logging by a wide margin, but, again, the hybrid approach is nearly as fast.

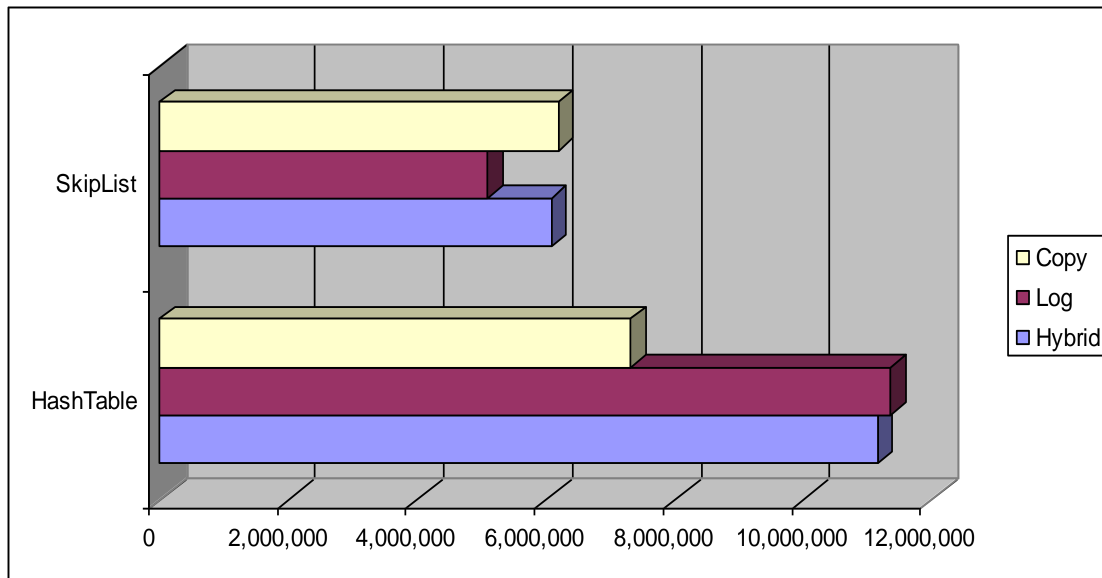



Figure 4.5: Performance comparison of atomic array algorithms

# Chapter 5

# Algorithms

## 5.1 Properties

This chapter introduces four obstruction-free transaction synchronization and validation algorithms designed for object-based STMs. Recent work in the area of STM [17, 23] has consistently suggested that lock-based STM systems offer better performance than those that support a nonblocking progress condition such as obstruction-freedom. This contention relies on the assumption that deadlock prevention is the only reason to build nonblocking algorithms. Our research challenges this conclusion. We show that, while it is true that obstruction-free STMs have more elegant theoretic

properties than locking systems, obstruction-free STMs can in fact equal or surpass the performance of systems using short locks.

We focus on three important properties of STM systems: a nonblocking progress condition, consistent reads, and support for contention management. Until now, no STM system has been able to achieve all three attributes efficiently. We argue that all three properties are important, if STM systems are to achieve widespread acceptance. In response to this challenge, we propose four transaction synchronization algorithms that satisfy these three properties simultaneously and efficiently. The table in Figure 5.1 compares several well-known transactional memory systems in terms of their support for these properties and shows that none support all three.

| **Algorithm** | **Non-Blocking?** | **Consistent Reads?** | **Contention Manager?** |
| --- | --- | --- | --- |
| **ASTM [63]** | ✓ | | ✓ |
| **Bulk [13]** | ✓ | ✓ | |
| **LogTM [67]** | | ✓ | |
| **McRT [74]** | | | ✓ |
| **RSTM [62]** | ✓ | | ✓ |
| **TCC [32]** | ✓ | ✓ | |
| **TL2 [16]** | | ✓ | |

Figure 5.1: Comparison of several TM algorithms

Furthermore, in contrast to many modern STM implementations that permit inconsistent reads and therefore violate the isolated property of transactional systems,

possibly leading to exceptions and infinite loops in client algorithms, our obstruction-free algorithms always return a consistent state that is linearizable. In systems that permit inconsistent reads, such problems are often dealt with through redundant checks inserted by compilers to detect and recover from these conditions [1], but, to our knowledge, no proof has been developed to show that these checks are sound in all cases.

In the algorithms we present, it is possible for a transaction as a whole to conflict with another already committed transaction, and thus have to be aborted, but at no point will a transaction have seen an inconsistent value written by an intervening transaction. We argue that such consistency is a critical property, if transactional memory systems are to be widely adopted.

Finally, the algorithms we present all support the use of orthogonal contention managers. Recent work on transaction synchronization algorithms [16], particularly in proposed hybrid software/hardware systems [78], does not mix well with flexible contention management policies. By contrast, we find that this is a valuable property even in locking STM systems, which are susceptible to livelock as well as deadlock and priority inversion. In addition, contention management has been shown to have a dramatic effect on the performance of transactional memory systems [76].

Our work builds upon several key attributes that define STM implementations. The algorithms rely on an object-based system that associates synchronization and recovery metadata with language-level objects, and thus is well suited to managed languages such as Java or C# [41, 48]. This distinguishes it from word-based STMs, which synchronize on uninterpreted regions of memory, and are better-suited for unmanaged languages, such as C or C++ [74]. An important advantage of object-based

STMs is their ability to enforce *strong atomicity* [5, 6, 80], ensuring that transactional and non-transactional accesses synchronize properly. Object-based STM systems are also particularly well-suited to obstruction-free implementations [46], as a single compare-and-swap (CAS) operation can atomically update an object's transactional state.

Section 5.2 presents the VisibleReaders algorithm, which is an optimization of the Locator-based obstruction-free synchronization algorithm originally introduced in the DSTM [48] and SXM [41] systems to accomplish efficient reader visibility. Section 5.3 describes the WarningWord algorithm, which outlines the basic building blocks of an obstruction-free reader-writer "lock." Section 5.4 expands the model outlined in § 5.3 and analyzes the behavior of an obstruction-free transaction synchronization algorithm based on a Bloom filter. Section 5.5 presents an alternative way of developing the model outlined in § 5.3 by introducing the WriterBins algorithm, which allows for an adjustable number of concurrent writers. Finally, Section 5.6 compares the four algorithms and describes the performance results. The complete a C# implementation for each algorithm presented is included in the Appendix.

## 5.2 The VisibleReaders algorithm

The VisibleReaders algorithm is based on the original obstruction-free approach described by the DSTM paper [48]. This algorithm employs a Locator object that contains references to the speculative and committed version of the object, as well as the current writer and a linked list of readers. While the DSTM model has been rightly criticized as inefficient [62] due to the extra level of indirection represented by the Locator, our work shows that its overhead can be effectively ameliorated by compiler

support that traverses the Locator object only on the initial read or write. Subsequent reads or writes of that object by the same transaction can use a cached reference directly to their private (in the case of writes), or shared (in the case of reads) copy of the object, effectively bypassing the Locator.

Figure 5.2 shows that, if an atomic object is opened for write access, the speculative “commit” version will be returned directly to the client algorithm, which can continue to use that copy of the object until it attempts to commit. If aborted by a competing transaction, the speculative copy becomes detached from the Locator and the writing transaction will not be permitted to commit. Thus, the extra level of indirection in the Locator object is an overhead that need only be traversed on the initial read or write.

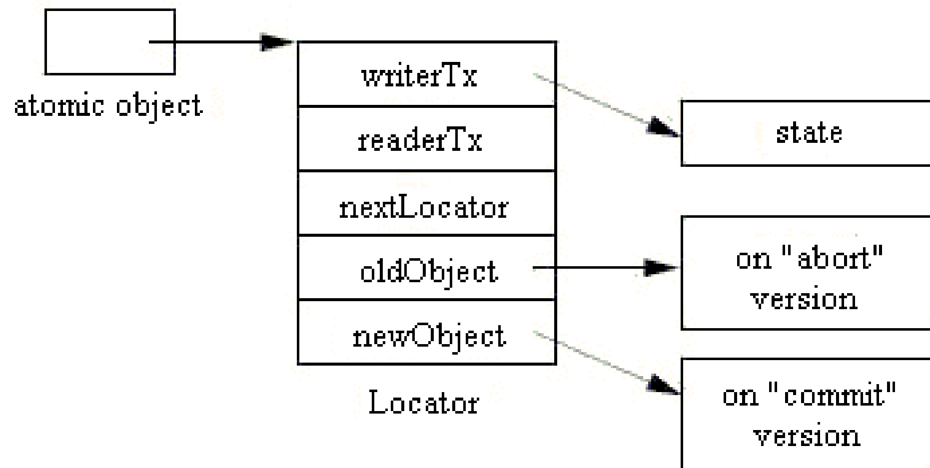



Figure 5.2: The transactional metadata model

In Figure 5.2, the nextLocator field points to a singly linked list of Locator objects; each represents a single reader transaction. This approach, which makes reader transactions visible to writers, works well with a contention management scheme, since a

writer that wants to take ownership of an atomic object knows which transactions are currently reading the object. The contention manager can then decide whether to wait for those reader transactions to complete or to abort them and allow the writer to go forward immediately.

The challenge posed by visible readers, however, is that the list of reader transactions must be updated and pruned in a nonblocking fashion. Adding new elements to the list is straightforward: a new locator is allocated, populated, and then swapped into the list's head position using a CAS operation. However, pruning completed (i.e., committed or aborted) readers from the list without locking is slightly more challenging. In the original DSTM algorithm the reader list was not pruned, leading to very inefficient scanning.

Our solution checks at each iteration whether the next entry represents a completed reader transaction. If so, the algorithm continues scanning forward until it detects either the end of the list (represented by a null value), or an active transaction. After each iteration, the starting pointer is updated with either a null value (indicating that the end of the list was reached) or a reference to the next transaction. This has the effect of pruning all the non-active transactions from the list.

```
ITERATE-READER-LIST(me, curentLocator)
1   while curentLocator ≠ NIL              ▷ iterate the list of readers
2       if currentLocator.reader = me
3           then return                    ▷ already a reader
4       currentLocator ← curentLocator.next ← PRUNE-READER-LIST(curentLocator.next)


PRUNE-READER-LIST(nextLocator)
    ▷ scan ahead and prune inactive transactions
1   while nextLocator ≠ NIL ∧ nextLoactor.reader.state ≠ ACTIVE
2       nextLocator ← nextLoactor.next
3   return nextLocator
```

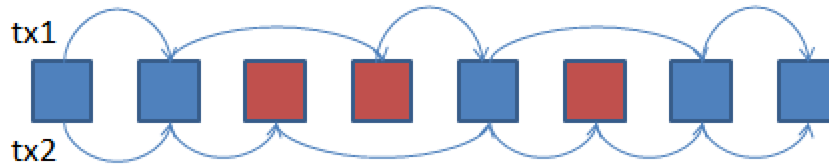



Figure 5.3: Two transactions concurrently prune the reader list safely without locks

The argument for the correctness of this optimization is that it preserves the following invariant: all active readers remain in the list; only inactive ones are pruned. As a result, concurrently running transactions may see and prune different inactive readers, but this has no effect on the invariant stated above. This is in contrast with the original DSTM and SXM approaches, which kept non-active (i.e., committed and aborted) transactions in the list. Because already completed (reader) transactions do not affect the linearizability of any other transactions, this optimization is safe. Furthermore, the pruning algorithm remains lock-free (i.e., it is safe to run concurrently and does not require a CAS operation to remove the dead readers). Threads concurrently pruning transactions from the same reader list do not interfere with one another. If a competing thread prunes some parts of the reader list at the same time that another thread is pruning other parts, the different transactions may observe a different set of readers depending on when those readers complete, but all threads will always see all other active readers (since only dead readers are discarded); see Figure 5.3. Nevertheless, the performance penalty of maintaining a nonblocking list of active readers, even where completed readers are pruned from the list, remains a significant impediment and thus limits the performance of this obstruction-free transaction synchronization algorithm.

*Definitions*: We now give the definitions of the locator list invariant and the associated rely and guarantee conditions; then we will prove the algorithm obeys their specifications. The locator list invariant is defined as follows:

$$\textit{LocatorListInv} \equiv \text{Locator(Head)} \wedge \text{Locator(Tail)} \wedge \forall_{\text{Locator}}\ \text{n. n.nextLocator = NIL} \vee \text{n.nextLoctor.reader} \neq \text{NIL}$$

The pre-condition is: $p \equiv \textit{LocatorListInv}$

The rely condition is: $R \equiv \forall_{\text{Locator}}\ \text{n. Preserve}(\textit{LocatorListInv}) \wedge \text{n.nextLocator.reader.state = ACTIVE} \Rightarrow \text{ID(n.nextLocator)} \wedge \text{n.reader = self} \Rightarrow \text{ID(n)}$

The guarantee condition is: $G \equiv \forall_{\text{Locator}}\ \text{n. Preserve}(\textit{LocatorListInv}) \wedge \text{n.nextLoctor.reader} \neq \text{NIL} \Rightarrow \text{n.nextLocator.reader.state = ACTIVE} \vee \text{n.reader = self} \Rightarrow \text{ID(n)}$

The post-condition is: $q \equiv \forall_{\text{Locator}}\ \text{n. Preserve}(\textit{LocatorListInv}) \wedge \text{n.nextLoctor.reader} \neq \text{NIL} \Rightarrow \text{n.nextLocator.reader.state = ACTIVE} \vee \text{n.reader = self} \Rightarrow \text{ID(n)}$

*Proof:* This is the reader list iterator algorithm annotated with pre- and post-conditions:

ITERATE-READER-LIST(*me*, *curentLocator*)
{ *LocatorListInv* }
1 **while** *curentLocator* ≠ NIL
{ *LocatorListInv* ∧ *currentLocator* ≠ NIL }
2 **if** *currentLocator.reader* = *me*
{ *LocatorListInv* ∧ *currentLocator* ≠ NIL ∧ *currentLocator.reader* = self }
3 **then return**

This exit branch satisfies the *guarantee* condition specifying that where the reader is the current transaction, the algorithm maintains the locator list invariant and does not modify the current locator, and in turn, satisifies the identical post-condition.

{ *LocatorListInv* ∧ *currentLocator* ≠ NIL ∧ *currentLocator.reader* ≠ self }
4 *currentLocator* ← *curentLocator.next* ← PRUNE-READER-LIST(*curentLocator.next*)
{ *LocatorListInv* ∧ (*currentLocator* = NIL ∨ *currentLocator.reader.state* = ACTIVE) }

This assignment modifies the structure of the list by eliding inactive readers. As the post-condition shows, however, the locator list invariant and the *guarantee* condition hold true.

When the algorithm returns because the *while* loop condition is false, the final post state can be stated as:

$\{$ *LocatorListInv* $\wedge$ *currentLocator* = NIL $\Rightarrow$ Locator(Tail).reader.state = ACTIVE$\}$

As shown above with pre- and post-conditions, at each step of the loop, the *guarantee* condition holds true. Thus, at each iteration of the loop, the current locator is active, proving, by induction, that the final locator is active when the loop terminates.

The prune reader list algorithm is invoked by the list iterator. Here is the pruning algorithm shown annotated with pre- and post-conditions:

PRUNE-READER-LIST(*nextLocator*)
$\{$ *LocatorListInv* $\wedge$ *nextLocator* $\neq$ NIL $\}$
1 **while** *nextLocator* $\neq$ NIL $\wedge$ *nextLoactor.reader.state* $\neq$ ACTIVE
$\{$ *LocatorListInv* $\wedge$ *nextLocator* $\neq$ NIL $\wedge$ *nextLocator.reader.state* $\neq$ ACTIVE $\}$
2 *nextLocator* $\leftarrow$ *nextLoactor.next*
$\{$ *LocatorListInv* $\}$

This assignment modifies only a local stack variable and so has no affect on the truth of the locator list invariant.

$\{$ *LocatorListInv* $\wedge$ (*nextLocator* = NIL $\vee$ *nextLocator.reader.state* = ACTIVE) $\}$
3 **return** *nextLocator*

## 5.3 The WarningWord algorithm

By dropping dead readers from the nonblocking readers list, the performance of the VisibleReaders algorithm described above greatly improves. However, as pointed out in [38], the overhead of maintaining a list of readers remains considerable. By contrast, better performance can be achieved by not maintaining such a record of readers, but, instead, relying on other mechanisms to detect and resolve read-write conflicts. This is the basis for the WarningWord algorithm, which relies on a single global 32-bit value we call the WarningWord. Each bit in the WarningWord corresponds to one transactional thread. Although by definition this limits the system to a maximum of 32 concurrent

threads, the approach could be easily expanded to use 64-bit or even larger values to support more threads.

In the WarningWord algorithm, write operations "warn" all other threads, and read operations check to see whether they have been warned. As a result, transactions become serialized as soon as there is an active writer present in the system. In effect, the WarningWord algorithm operates as a system-wide reader-writer "lock." Although potential concurrency is sacrificed for lower read/write overhead, this approach significantly outperforms the more conventional visible reader techniques. These results reinforce the argument advanced in [17], which suggests that the key to lowering overall STM overhead is to lower the cost of a single transaction.

In the WarningWord algorithm, and the BloomFilter and WriterBins algorithms described in §§ 5.4 and 5.5, the Locator object contains only the object's current writer and references to the new and old copies of the object. There is no longer any reader information stored, and thus no need for a linked list of Locator objects.

```
class Locator { // no reader info
   XState writer;
   ICloneable oldObject, newObject;
}
```

On transaction start, the WarningWord algorithm acquires the thread ID that will uniquely represent this transaction's warning bit. Then it clears the thread's bit in the global WarningWord value using a CAS operation. This ensures that the new transaction starts with a clean slate, even where the last transaction run by this thread may have aborted due to a warning.

```
INITIALIZE-TX(me)
1  me.threadId ← GETTHREADID
2  do        ▷ clear the warning bit for this thread
3      oldValue ← GLOBALWARNINGWORD
4      newValue ← oldValue ∧ me.threadId
5  until CAS(GLOBALWARNINGWORD, newValue, oldValue)
```

Writers warn other threads that they are the exclusive writer by setting the warning bits for all other threads:

```
WARN-OTHER-THREADS(me)
1  do
2      oldValue ← GLOBALWARNINGWORD
3      newValue ← oldValue ∨ ~(1 << me.threadId)
4  until CAS(GLOBALWARNINGWORD, newValue, oldValue)
```

Both transactional reads and writes check whether their thread has been "warned," in which case they must abort:

```
CHECK-WARNING-WORD(me)
1  if GLOBALWARNINGWORD ∧ (1 << me.threadId) ≠ 0
2      then throw ABORTED-EXCEPTION
```

The argument for the correctness of the WarningWord algorithm proceeds as follows. First each transactional thread is given a unique number that remains constant for the life of the process; this means that the set of all possible threads that may initiate transactions must be known in advance. Read sharing is permitted—initial reads simply check that their thread's "warning" bit is clear. Transactional writer threads atomically set all other threads' warning bit before acquiring any atomic object via the locator. Thus, following the same logic advanced in § 4.3, if the warning bit is clear at a time subsequent to the time a value was read, that value is linearizable. Since obstruction-free transactional objects are immutable (see § 4.4), a safe value, once read, remains safe for

later use, even if a writing transaction replaces the current version of the object in the locator. By way of contradiction, after a writer warns all other threads and acquires ownership of the atomic object, all later threads that check their warning bit are aborted.

*Definitions*: We now give the definitions of the warning word invariant and the associated rely and guarantee conditions; then we will prove the algorithm obeys their specifications. The warning word invariant is defined as follows:

$$WarningWordInv \equiv \forall_{\text{transaction}}\ \text{tx}.\ tx.threadId \leftarrow \text{GETTHREADID} \wedge \text{GLOBALWARNINGWORD} \wedge (1 << tx.threadId) \neq 0 \Rightarrow \text{ABORT}(tx)$$

The rely condition is: $R \equiv WarningWordInv$
$\wedge\ \forall_{\text{transaction}}\ \text{tx}.\ tx.Write(obj) \Rightarrow obj.Locator.writer \leftarrow tx$
$\wedge$ AllBitsSet(GLOBALWARNINGWORD), except bit number $tx.threadId$)

The guarantee condition is: $G \equiv WarningWordInv$
$\wedge\ me.Write(obj) \Rightarrow obj.Locator.writer \leftarrow me$
$\wedge$ AllBitsSet(GLOBALWARNINGWORD), except bit number $me.threadId$)

The pre-condition is: $p \equiv WarningWordInv \wedge$ Locator.writer = self

The post-condition is: $q \equiv WarningWordInv \wedge$ Locator.writer = self
$\wedge$ AllBitsSet(GLOBALWARNINGWORD), except bit number me.threadId

The WARN-OTHER-THREADS algorithm is executed immediately after a thread takes ownership of an atomic object for writing. Thus, the precondition is the starting point:

WARN-OTHER-THREADS(*me*)
{ $WarningWordInv \wedge$ Locator.writer = self }
1 **do**
2 $oldValue \leftarrow$ GLOBALWARNINGWORD
3 $newValue \leftarrow oldValue \vee \sim(1 << me.threadId)$
4 **until** CAS(GLOBALWARNINGWORD, *newValue*, *oldValue*)
{ $WarningWordInv \wedge$ Locator.writer = self
$\wedge$ AllBitsSet(GLOBALWARNINGWORD), except bit number me.threadId

The post-condition is satisfied by the CAS operation that sets all bits, except for that of the writing thread. The CheckWarningWord algorithm does not modify any values, and thus serves only to prevent reads and writes from proceeding on threads that have been warned, serving to satisfy the *WarningWordInv*.

However, several drawbacks remain: in addition to the obvious lack of write concurrency in the WarningWord model, a second challenge is the limited support for contention management in this algorithm. The contention manager can only be invoked when a reader or a writer encounters a direct conflict with another active writer. However, because readers are invisible, when a writer conflicts with an active reader there is no way for the writer to learn of the conflict. The conflicting readers will discover that they've been warned and then have to abort. But a contention management policy cannot be imposed to ensure that readers get a fair chance to complete.

Theoretically, the WarningWord algorithm is susceptible to livelock between reader and writer threads: if a writer does not complete within a reasonable period of time, it can only be aborted by another directly conflicting reader or writer; readers of other objects cannot abort the writer because they have no way of identifying it. In practice, however, the opposite is true. Because readers abort when they encounter an active writer, the WarningWord algorithm gives writers precedence over readers and thus exhibits excellent concurrency with short-lived transactions.

Another important feature is that the WarningWord approach can be applied to a blocking STM model as well, although we found that it worked best in the obstruction-free STM, probably because the blocking mode requires two added CAS operations per (initial) read and write (to acquire and release the lock).

## 5.4 The BloomFilter algorithm

Bloom filters [4] have become increasingly popular in STM systems as way of quickly checking whether an address is represented in a change log [17]. Bulk [13] uses a

related technique to represent a thread's access information. This is often an efficient way of solving the read-after-write problem in redo STMs. Bloom filters, of course, can return false positives, so the fact that a value appears in the Bloom filter does not necessarily mean that it actually exists in the change log. However, by quickly checking the Bloom filter, the STM can determine whether it is even necessary to consult the log.

One common problem with Bloom filters, however, is how to clean the filter when a transaction commits or aborts. Because multiple addresses may map to the same entry in the filter (i.e., multiple transactions may add the same objects to the filter), it can be dangerous to remove an entry. One common solution is to use a counting Bloom filter [24] in which each time a value is added to filter an associated counter is incremented; consequently, entries are removed only when their counter is decremented to zero.

Rather than implementing a counting Bloom filter, however, we observe that although multiple objects may hash to the same spot in the filter, the interesting conflicts in STM systems are those that occur between transactions. In other words, the question we ask is not how many times an object hashing to a specific entry has been added to the filter (this can be answered by a counting Bloom filter), but, rather, whether competing transactions have added an object mapping to this entry in the filter.

Based on this insight, we use a Bloom filter arranged as an array of 32-bit values—each bit representing one of thirty-two possible thread IDs. In this way, a thread that adds an object to the filter can tell whether a different transaction has already added it previously. Our current filter size is 256KB, or 65,536 entries of 32-bits each. A larger filter would make false positives less likely, but we have not found this to be a problem.

A global WarningWord value is used by writers to warn reader threads of a likely conflict. In this sense, it is similar to the WarningWord algorithm described in § 5.3, with one major difference: rather than any writing transaction warning all other threads, a writer now checks to see whether an active reader is registered at the filter entry for the target object. If so, the WarningWord is set to warn just the conflicting thread rather than all other threads.

Each new transaction obtains a thread ID and then a reference to that thread's read list:

```
INITIALIZE-TX(me)
1   me.threadId ← GETTHREADID
    ▷ per-tx read list for cleanup
2   me.txReadList ← GLOBALREADLISTS[me.threadId]
```

Read operations add the atomic object's hash to the transaction's read list and set the appropriate entry in the Bloom filter:

```
ADD-READ-OBJECT(me, hash)
    ▷ add the object hash to the tx's read list
1   me.txReadList.Add(hash)
    ▷ then add it to the Bloom filter
2   BLOOMFILTER-ADD(hash, me.threadId)
```

Before returning the atomic object opened for read access, the algorithm checks to see whether a writer thread has "warned" this transaction of a possible conflict. If so, the transaction must be aborted and the Bloom filter cleansed of all entries added by this transaction.

```
READ-OBJECT-VALIDATE(me)
1   if CHECK-WARNING-WORD(me.threadId) = TRUE
2       then FILTER-CLEANUP(me)
3           throw ABORTED-EXCEPTION
```

Similarly, transactional writes check the Bloom filter for possible reader conflicts and then “warn” any threads that may conflict:

```
WRITE-OBJECT-VALIDATE(me, hash)
1   conflict ← BLOOMFILTER[hash] ∧ ~(1 << me.threadId)
2   if conflict ≠ 0
3       then SET-WARNING-WORD(conflict)
4   if CHECK-WARNING-WORD(me.threadId) = TRUE
5       then FILTER-CLEANUP(me)
6           throw ABORTED-EXCEPTION
```

Our algorithm makes filter cleanup straightforward. Because each thread maintains a list of read objects, when a transaction commits or aborts that list can be iterated to remove those objects from the filter. Only the thread ID associated with the completing transaction is actually removed; other threads that read objects mapping to the same filter entry remain unaffected.

```
FILTER-CLEANUP(me)
1   for i ← 0 to me.txReadList.Count
2       do
3           oldValue ← BLOOMFILTER[me.txReadList[i]]
4           newValue ← oldValue ∧ me.threadId
5       until CAS(BLOOMFILTER[me.txReadList[i]], newValue, oldValue)
    ▷ clear the warning for this transaction
6   CLEAR-WARNING-WORD(me.threadId)
7   me.txReadList.Clear()          ▷ empty the read list for this transaction
```

The argument for the correctness of the BloomFilter algorithm proceeds as follows. Traditional Bloom filters are bitmaps of entries that map from larger sets via one or more hash functions. Our approach uses a two-dimensional Bloom filter, where each entry in the filter is represented not by a single bit, but, rather, by a warning word. This permits writer concurrency at the resolution of the Bloom filter’s size. Where a likely conflict is detected, the thread owning that entry in the filter is aborted by setting its

warning bit in the global warning word. Once at the level of the global warning word, the arguments for correctness advanced in § 5.3 apply. As shown in [4], Bloom filters can return false positives but not false negatives. In the context of transactional memory, false positives might occasionally abort a transaction unnecessarily, but transactional linearizability is never violated.

*Definitions*: We now give the definitions of the bloom filter invariant and the associated rely and guarantee conditions; then we will prove the algorithm obeys their specifications. The bloom filter invariant is defined as follows:

$$\textit{BloomFilterInv} \equiv \text{threadId} \leftarrow \textsc{GetThreadId} \wedge \exists \, \forall_{\text{threadId}} \, \text{id. txReadList} \in \textsc{GlobalReadLists}[\textit{id}]$$

The rely condition is: $R \equiv \textit{BloomFilterInv}$

$$\wedge \, \forall_{\text{transaction}} \, \text{tx. } \textit{tx}.\text{Write(txObj)} \Rightarrow \textsc{BloomFilter}[\textit{hash}] \wedge \sim(1 << \textit{tx.threadId}) \neq 0 \Rightarrow \textsc{Set-Warning-Word}(\textsc{BloomFilter}[\textit{hash}] \wedge \sim(1 << \textit{tx.threadId}))$$

$$\wedge \, \forall_{\text{transaction}} \, \text{tx. } \textit{tx}.\text{Read(txObj)} \Rightarrow \textit{hash} \in \textit{tx.txReadList} \wedge \textsc{BloomFilter}(\textit{hash, tx.threadId}) \wedge \textsc{Check-Warning-Word}(\textit{tx.threadId}) = \text{TRUE} \Rightarrow \text{ABORT}(\textit{tx})$$

This rule specifies that any transaction that writes an object where a conflicting reader is registered in the Bloom filter must set the warning word for that other thread; any transaction that reads an object must register its (hash, threadId) tuple in the Bloom filter.

The guarantee condition is: $G \equiv \textit{BloomFilterInv}$

$$\wedge \, \textit{me}.\text{Write(txObj)} \Rightarrow \textsc{BloomFilter}[\textit{hash}] \wedge \sim(1 << \textit{tx.threadId}) \neq 0 \Rightarrow \textsc{Set-Warning-Word}(\textsc{BloomFilter}[\textit{hash}] \wedge \sim(1 << \textit{me.threadId}))$$

$$\wedge \, \textit{me}.\text{Read(txObj)} \Rightarrow \textit{hash} \in \textit{me.txReadList} \wedge \textsc{BloomFilter}(\textit{hash, me.threadId}) \wedge \textsc{Check-Warning-Word}(\textit{me.threadId}) = \text{TRUE} \Rightarrow \text{ABORT}(\textit{me})$$

This rule specifies that the transaction that writes an object where a conflicting reader is registered in the Bloom filter must set the warning word for that other thread; the transaction that reads an object must register its (hash, threadId) tuple in the Bloom filter.

For the ADD-READ-OBJECT and READ-OBJECT-VALIDATE algorithms, which are executed sequentially, the pre- and post-conditions are defined as follows (the *rely-guarantee* conditions are the same as those given above):

The pre-condition is: $p \equiv BloomFilterInv$
$\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*]

The post-condition is: $q \equiv BloomFilterInv$
$\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*]
$\wedge$ CHECK-WARNING-WORD(*me.threadId*) ≠ TRUE ⇒
*hash* ∈ *me.txReadList* ∧ BLOOMFILTER(*hash, me.threadId*)

*Proof:* The read algorithms are executed only after the initialize step. Thus, the precondition is the starting point:

{ *BloomFilterInv* ∧ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*] }
1 *me.txReadList.Add*(*hash*)
{ *BloomFilterInv* ∧ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*] ∧
*hash* ∈ *me.txReadList* }
2 BLOOMFILTER-ADD(*hash, me.threadId*)
{ *BloomFilterInv* ∧ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*] ∧
*hash* ∈ *me.txReadList* ∧ BLOOMFILTER(*hash, me.threadId* }
3 **if** CHECK-WARNING-WORD(*me.threadId*) = TRUE
4 **then** FILTER-CLEANUP(*me*)
5 **throw** ABORTED-EXCEPTION
{ *BloomFilterInv* ∧ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*] ∧
CHECK-WARNING-WORD(*me.threadId*) ≠ TRUE ∧ *hash* ∈ *me.txReadList* ∧
BLOOMFILTER(*hash, me.threadId* }

This final post-condition also satisfies the reader portion of the guarantee property for the algorithm and does not violate the rely property on which other threads depend.

For the WRITE-OBJECT-VALIDATE algorithm, the pre- and post-conditions are defined as follows (the *rely-guarantee* conditions are the same as those given above):

The pre-condition is: $p \equiv BloomFilterInv$
$\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*]

The post-condition is: $q \equiv \mathit{BloomFilterInv}$

$\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*]

$\wedge$ BLOOMFILTER[*hash*] $\wedge$ ~(1 << *me.threadId*) $\neq 0 \Rightarrow$

SET-WARNING-WORD(BLOOMFILTER[*hash*] $\wedge$ ~(1 << *me.threadId*))

CHECK-WARNING-WORD(*me.threadId*) ≠ TRUE

*Proof:* The read algorithms are executed only after the initialize step. Thus, the precondition is the starting point:

{ *BloomFilterInv* $\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*] }
1 *conflict* ← BLOOMFILTER[*hash*] $\wedge$ ~(1 << *me.threadId*)
2 **if** *conflict* $\neq 0$
{ *BloomFilterInv* $\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*]
$\wedge$ *conflict* $\neq 0$ }
3 **then** SET-WARNING-WORD(*conflict*)
{ *BloomFilterInv* $\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*]
$\wedge$ *conflict* $\neq 0 \Rightarrow$ SET-WARNING-WORD(*conflict*) }
4 **if** CHECK-WARNING-WORD(*me.threadId*)
5 **then** FILTER-CLEANUP(*me*)
6 **throw** ABORTED-EXCEPTION
{ *BloomFilterInv* $\wedge$ *me.txReadList* ← GLOBALREADLISTS[*me.threadId*]
$\wedge$ *conflict* $\neq 0 \Rightarrow$ SET-WARNING-WORD(*conflict*)
$\wedge$ CHECK-WARNING-WORD(*me.threadId*) ≠ TRUE }

This final post-condition also satisfies the reader portion of the guarantee property for the algorithm and at no point does it violate the rely property on which other threads are entitled to depend. (Note that *conflict* ← BLOOMFILTER[*hash*] $\wedge$ ~(1 << *me.threadId*) from step 1).

Like the WarningWord algorithm, the BloomFilter algorithm provides somewhat limited support for contention management in cases of read-write conflicts. Because transactions are still notified of likely reader conflicts by way of a thread-specific warning bit, the contention manager cannot be used to resolve these conflicts. But, since the BloomFilter identifies reader conflicts only where they most likely actually exist, the likelihood of spurious aborts and hence the chance of livelock is greatly reduced.

## 5.5 The WriterBins algorithm

The obstruction-free algorithms described in the preceding sections are transaction synchronization algorithms in the sense that they don't permit a nonlinearizable operation to occur. Conflicts are handled as they occur by aborting the current transaction or calling a contention manager to resolve the conflict. As a result, commit operations are extremely simple: just a single CAS switches the transaction's state from *active* to *committed*.

By contrast, the WriterBins algorithm loosens this restriction in order to permit some nonlinearizable transactions to proceed. These transactions are then caught by the validation algorithm prior to commit, and aborted. It is important to emphasize, however, that the WriterBins algorithm does not allow inconsistent reads. That is, at no point can the same copy of the atomic object be read and written by different transactions. Unlike the WarningWord algorithm described in § 5.3, which permitted only one system-wide writer, the WriterBins algorithm is configurable. It can permit anywhere from 1 to *n* concurrent writers. In our tests, we have worked with a maximum of 32 concurrent writers in order to make the comparisons among the algorithms more meaningful (see § 5.6).

The WriterBins algorithm's data structures are arranged as follows. First, a global array of 32 integers is created to store the version number of each bin. Second, a global array of 32 transactions is created to store the transaction that is the current owner of each bin. Having an array of the actual transactions rather than simply a bitmap enables two transactions that conflict over a bin to invoke the contention manager to help resolve their differences.

```
XState GlobalWriterBins[32];
int GlobalBinVerCounters[32];
```

In addition to these two global arrays, each transaction maintains three pieces of private data: a 32-bit map of readers, writers, and a copy of the bin version numbers that were current when the transaction started.

```
// 1 bit per bin
int txReadBins, txWriteBins;

// local cache of version counters
int txBinVerCounters[32];
```

Transactional read operations first check whether the bin that the target object maps to has been read previously. If not, the transaction's read bins value (txReadBins) is updated. Then the read operation checks the transaction's local bin version number against the global one. This check is not required, as transactions are validated at commit-time, but is effective in aborting doomed transactions early.

```
READ-OBJECT(me, bin)
1  if (me.txReadBins ∧ 1 << bin) = 0                ▷ have we read this bin?
2      then me.txReadBins ← me.txReadBins ∨ 1 << bin   ▷ no, set the bin bit
3      ▷ early check for conflicts
4      if GLOBALBINVERCOUNTERS[bin] ≠ me.txBinVerCounters[bin]
5          then thrown ABORTED-EXCEPTION
```

The transactional write operation checks txWriteBins to determine whether the writing transaction has previously written an object mapping to this bin. If not, the algorithm will acquire ownership of the global bin, if necessary with the assistance of the contention manager.

```
WRITE-OBJECT(me, bin)
1   if (me.txWriteBins ∧ 1 << bin) = 0                ▷ have we written this bin?
2       then do                                       ▷ no
3           oldTx ← GLOBALWRITERBINS[bin]
            ▷ check bin version numbers
4           if GLOBALBINVERCOUNTERS[bin] ≠ me.txBinVerCounters[bin]
5               then thrown ABORTED-EXCEPTION
6           if oldTx.state = ACTIVE
7               then RESOLVE-CONFLICT(me, oldTx)      ▷ conflict – call ConMan
8                   continue
9       until CAS(GLOBALWRITERBINS[bin], me, oldTx)
10      me.txWriteBins ← me.txWriteBins ∨ 1 << bin    ▷ set the written bin bit
```

The most interesting part of the WriterBins algorithm, however, is the commit-time validation. This is performed in two phases, both accomplished without any CAS operations. The first pass iterates through the bins and increments both the local and global version counters for all bins written by this transaction. The second pass compares the local and global version counters for all read and written bins. If all bin counters match during the second pass, the transaction can commit; otherwise it must abort.

```
VALIDATE-TX(me)
    ▷ first pass – increment version numbers for written bins
1   for bin ← 0 to 32
2       if (1 << bin ∧ me.txWriteBins) ≠ 0        ▷ did the tx write this bin?
3           then GLOBALBINVERCOUNTERS[bin]++
4               me.txBinVerCounters[bin]++
    ▷ second pass – check all touched bin version numbers
5   txRWBins ← me.txReadBins ∨ me.txWriteBins
6   for bin ← 0 to 32
7       if (1 << bin ∧ me.txRWBins) ≠ 0           ▷ did the tx read/write this bin?
8           then if GLOBALBINVERCOUNTERS[bin] ≠ me.txBinVerCounters[bin]
9               then ABORT-TX(me)
10                  return FALSE
11  return TRUE
```

The argument for the correctness of the WriterBins algorithm proceeds as follows. First, multiple writers can now run in tandem, so long as they don't conflict over

individual atomic objects (managed by a locator object—see § 5.2). Nonlinearizable conflicts among transactions are detected by the commit-time validation algorithm shown above. At time $t_1$, when the transaction begins, the then current version number for each bin is recorded in a private transaction-specific array. At commit-time, say time $t_{1+n}$, the algorithm makes two passes over the bins. First, for each bin *written* by the committing transaction, the validation algorithm increments both the local and global bin version numbers. Second, at time $t_{1+n+1}$, the algorithm compares the local and global bin version numbers for all bins *either read or written* by the transaction. If all checked bin numbers match, we argue that no competing transaction could have written to any of the bins that the validating transaction read or wrote between time $t_1$ and $t_{1+n}$. Since, at the subsequent time $t_{1+n+1}$ the bin numbers still matched, and bin numbers are only incremented, never decremented, they must have matched at the earlier time as well, and thus the transaction is safe to try to commit.[1]

Once this second check completes successfully, the transaction has been validated and can attempt to commit by flipping its state from *Active* to *Committed* via a CAS operation, at time $t_{1+n+2}$. Of course, it is always possible that a competing transaction has stolen one of this transaction's objects between time $t_{1+n+1}$ and $t_{1+n+2}$, and, in the process, aborted us. If so, the CAS operation will fail and all the current transaction's mutations are discarded. The unnecessarily incremented bin version numbers, however, are not rolled back; at worst, this causes the rare spurious abort.

---

[1] Rollover will happen when the bin counters reach their maximum value. For this to be a problem, however, precisely $2^{32}$ transactions would have to write to that specific bin (and either no others or else all others read and/or written by the current transaction) and attempt to commit between times $t_1$ and $t_{1+n+1}$. Since we assume that transactions are relatively short-lived, we do not believe this to be a problem in practice. The overhead of an empty transaction is currently approximately 1µs, so a single transaction would have to run for more than one hour (4295 seconds) in order for this to be even a slight possibility. Increasing the bin version numbers to 64 bits would make this a practical impossibility.

*Definitions*: We now give the definitions of the writer bins invariant and the associated rely and guarantee conditions; then we will prove the algorithm obeys their specifications. The writer bin invariant is defined as follows:

$$\mathit{WriterBinInv} \equiv \forall_{\text{bin, transaction}}\ \text{b, tx.}\ (1 << b \wedge (\mathit{tx.txReadBins} \vee \mathit{tx.txWriteBins})) \neq 0 \Rightarrow \\ \text{GLOBALBINVERCOUNTERS}[b] \neq \mathit{tx.txBinVerCounters}[b] \Rightarrow \\ \text{ABORT}(\mathit{tx})$$

This rule says that for any given transaction that reads or writes atomic objects, the transaction's bin counters for the read/written bins must match the global bin counters, or else the transaction must abort.

The rely condition is: $R \equiv \mathit{WriterBinInv}$

$$\wedge\ \forall_{\text{bin, transaction}}\ \text{b, tx.}\ \mathit{tx}.\text{Read}(b) \Rightarrow \mathit{tx.txReadBins} \leftarrow \mathit{tx.txReadBins} \vee 1 << b \\ \wedge\ \forall_{\text{bin, transaction}}\ \text{b, tx.}\ \mathit{tx}.\text{Write}(b) \Rightarrow \text{GLOBALWRITERBINS}[b] \leftarrow \mathit{tx} \\ \wedge\ \mathit{tx.txWriteBins} \leftarrow \mathit{tx.txWriteBins} \vee 1 << b \\ \wedge\ \forall_{\text{transaction}}\ \text{tx.}\ \mathit{tx}.\text{Commit}() \Rightarrow \forall_{\text{bin}}\ \text{b.}\ (1 << b \wedge \mathit{tx.txWriteBins}) \neq 0 \Rightarrow \\ (\text{GLOBALBINVERCOUNTERS}[b]{+}{+} \wedge \mathit{tx.txBinVerCounters}[b]{+}{+})$$

This condition says that any transaction that wants to read or write an object belonging to a specific bin must set the appropriate bit in its txReadBins or txWriteBins map, respectively; that any bin written by a transaction must also be "owned" by assigning the owning transaction in the GLOBALWRITERBINS[*b*] array; and that any transaction attempting to commit will increment GLOBALBINVERCOUNTERS[*b*] and its local counters for each bin that was written.

The guarantee condition is: $G \equiv \mathit{WriterBinInv}$

$$\wedge\ \forall_{\text{bin}}\ \text{b. Read(b)} \Rightarrow \mathit{self.txReadBins} \leftarrow \mathit{self.txReadBins} \vee 1 << b \\ \wedge\ \forall_{\text{bin}}\ \text{b. Write(b)} \Rightarrow (\text{GLOBALWRITERBINS}[b] = \mathit{self} \\ \wedge\ \mathit{self.txWriteBins} \leftarrow \mathit{self.txWriteBins} \vee 1 << b) \\ \wedge\ \text{Commit}() \Rightarrow \forall_{\text{bin}}\ \text{b.}\ (1 << b \wedge \mathit{self.txWriteBins}) \neq 0 \Rightarrow \\ (\text{GLOBALBINVERCOUNTERS}[b]{+}{+} \wedge \mathit{self.txBinVerCounters}[b]{+}{+})$$

This condition guarantees that current the transaction will set the appropriate bit in its txReadBins or txWriteBins map, respectively, for each bin read or written; that the GLOBALWRITERBINS[*b*] array will be set to the current transaction for all bins written; and that, when attempting to commit, the transaction will increment GLOBALBINVERCOUNTERS[*b*] and its local counters for each bin that was written.

For the READ-OBJECT algorithm, the pre- and post-conditions are defined as follows (the *rely-guarantee* conditions are the same as those given above):

The pre-condition is: $p \equiv$ none

The post-condition is: $q \equiv \mathit{WriterBinInv} \wedge \mathit{me.txReadBins} \leftarrow \mathit{me.txReadBins} \vee 1 << \mathit{bin}$

*Proof:* The read algorithm is shown below annotated with pre- and post-conditions. The final post-condition satisfies the *WriterBinInv* by ensuring that transacion where the local bin counters don't match the global ones are aborted.

1 **if** $(\mathit{me.txReadBins} \wedge 1 << \mathit{bin}) = 0$
$\{ (\mathit{me.txReadBins} \wedge 1 << \mathit{bin}) = 0) \}$
2 **then** $\mathit{me.txReadBins} \leftarrow \mathit{me.txReadBins} \vee 1 << \mathit{bin}$
$\{ \mathit{me.txReadBins} \leftarrow \mathit{me.txReadBins} \vee 1 << \mathit{bin} \}$
3 **if** GLOBALBINVERCOUNTERS[*bin*] $\neq$ *me.txBinVerCounters*[*bin*]
$\{ \mathit{me.txReadBins} \leftarrow \mathit{me.txReadBins} \vee 1 << \mathit{bin}$
$\wedge$ GLOBALBINVERCOUNTERS[*bin*] $\neq$ *me.txBinVerCounters*[*bin*]}
4 **then thrown** ABORTED-EXCEPTION
$\{ \mathit{WriterBinInv} \wedge \mathit{me.txReadBins} \leftarrow \mathit{me.txReadBins} \vee 1 << \mathit{bin} \}$

For the WRITE-OBJECT algorithm, the pre- and post-conditions are defined as follows (the *rely-guarantee* conditions are the same as those given above):

The pre-condition is: $p \equiv$ none

The post-condition is: $q \equiv \mathit{WriterBinInv} \wedge$ GLOBALWRITERBINS[*bin*] $\leftarrow$ *me*
$\wedge\ \mathit{me.txWriteBins} \leftarrow \mathit{me.txWriteBins} \vee 1 << \mathit{bin}$

*Proof:* The write algorithm is shown below annotated with pre- and post-conditions; there is no pre-condition and the post-condition matches that given above.

1 **if** $(\mathit{me.txWriteBins} \wedge 1 << \mathit{bin}) = 0$
$\{ (\mathit{me.txWriteBins} \wedge 1 << \mathit{bin}) = 0) \}$
2 **then do**
3 *oldTx* $\leftarrow$ GLOBALWRITERBINS[*bin*]
4 **if** GLOBALBINVERCOUNTERS[*bin*] $\neq$ *me.txBinVerCounters*[*bin*]
$\{ (\mathit{me.txWriteBins} \wedge 1 << \mathit{bin}) = 0)$
$\wedge$ GLOBALBINVERCOUNTERS[*bin*] $\neq$ *me.txBinVerCounters*[*bin*] }
5 **then thrown** ABORTED-EXCEPTION
$\{ \mathit{WriterBinInv} \wedge (\mathit{me.txWriteBins} \wedge 1 << \mathit{bin}) = 0) \}$
6 **if** *oldTx.state* = ACTIVE

{ $WriterBinInv \wedge (me.txWriteBins \wedge 1 << bin) = 0)$
  $\wedge$ GLOBALWRITERBINS[*bin*].*state* = ACTIVE }
7 **then** RESOLVE-CONFLICT(*me, oldTx*)
8 **continue**
{ $WriterBinInv \wedge (me.txWriteBins \wedge 1 << bin) = 0)$
  $\wedge$ GLOBALWRITERBINS[*bin*].*state* $\neq$ ACTIVE }
9 **until** CAS(GLOBALWRITERBINS[*bin*], *me, oldTx*)
{ $WriterBinInv \wedge (me.txWriteBins \wedge 1 << bin) = 0)$
  $\wedge$ GLOBALWRITERBINS[*bin*] $\leftarrow$ *me* }
10 $me.txWriteBins \leftarrow me.txWriteBins \vee 1 << bin$
{ $WriterBinInv \wedge$ GLOBALWRITERBINS[*bin*] $\leftarrow$ *me*
  $\wedge\ me.txWriteBins \leftarrow me.txWriteBins \vee 1 << bin$ }

For the VALIDATE-TX algorithm, the pre- and post-conditions are defined as follows (the *rely-guarantee* conditions are the same as those given above):

The pre-condition is: $p \equiv$ none

The post-condition is: $q \equiv WriterBinInv$
  $\wedge\ \forall_{bin}$ b. $(1 << b \wedge me.txWriteBins) \neq 0 \Rightarrow$
    (*me*. GLOBALBINVERCOUNTERS[*b*]++ $\wedge$ *me.txBinVerCounters*[*b*]++)
  $\wedge\ \forall_{bin}$ b. $(1 << b \wedge me.txRWBins) \neq 0 \Rightarrow$
    GLOBALBINVERCOUNTERS[*b*] = *me.txBinVerCounters*[*b*]

*Proof:* The transaction validation algorithm is shown below annotated with pre- and post-conditions; there is no pre-condition and the post-condition matches that given above.

1 **for** $bin \leftarrow 0$ **to** 32
{ $bin \leftarrow 0{:}32$ }
2 **if** $(1 << bin \wedge me.txWriteBins) \neq 0$
{ $bin \leftarrow 0{:}32 \wedge (1 << bin \wedge me.txWriteBins) \neq 0$ }
3 **then** GLOBALBINVERCOUNTERS[*bin*]++
4 *me.txBinVerCounters*[*bin*]++
{ $bin \leftarrow 0{:}32$
  $\wedge\ \forall_{bin}$ b. $(1 << b \wedge me.txWriteBins) \neq 0 \Rightarrow$
    (*me*. GLOBALBINVERCOUNTERS[*b*]++ $\wedge$ *me.txBinVerCounters*[*b*]++) }
5 $txRWBins \leftarrow me.txReadBins \vee me.txWriteBins$
6 **for** $bin \leftarrow 0$ **to** 32
7 **if** $(1 << bin \wedge me.txRWBins) \neq 0$

{ $bin \leftarrow 0{:}32 \wedge (1 << bin \wedge me.txRWBins) \neq 0$
$\wedge \forall_{bin}$ b. $(1 << b \wedge me.txWriteBins) \neq 0 \Rightarrow$
$(me.$ GLOBALBINVERCOUNTERS$[b]{+}{+} \wedge me.txBinVerCounters[b]{+}{+})$ }
8 **then if** GLOBALBINVERCOUNTERS$[bin] \neq me.txBinVerCounters[bin]$
{ $bin \leftarrow 0{:}32 \wedge (1 << bin \wedge me.txRWBins) \neq 0$
$\wedge \forall_{bin}$ b. $(1 << b \wedge me.txWriteBins) \neq 0 \Rightarrow$
$(me.$ GLOBALBINVERCOUNTERS$[b]{+}{+} \wedge me.txBinVerCounters[b]{+}{+})$
$\wedge$ GLOBALBINVERCOUNTERS$[bin] \neq me.txBinVerCounters[bin]$ }
9 **then** ABORT-TX(*me*)
{ *WriterBinInv* }
10 **return** FALSE
11 **return** TRUE
{ *WriterBinInv* $\wedge$
$\forall_{bin}$ b. $(1 << b \wedge me.txWriteBins) \neq 0 \Rightarrow$
$(me.$ GLOBALBINVERCOUNTERS$[b]{+}{+} \wedge me.txBinVerCounters[b]{+}{+})$
$\wedge \forall_{bin}$ b. $(1 << b \wedge me.txRWBins) \neq 0 \Rightarrow$
GLOBALBINVERCOUNTERS$[b] = me.txBinVerCounters[b]$ }

Figure 5.4 shows the type of nonlinearizable transactions detected and aborted by the WriterBins validation algorithm. In this example, either tx1 or tx2 may commit, but not both. If tx1 commits first, it will increment the version number of the bins for obj2. This, in turn, will cause the validation algorithm to fail for tx2, since its version numbers will no longer match the global ones. Because the WriterBins validation algorithm is lock-free, it is possible that tx1 and tx2 will attempt to commit simultaneously. In such cases, tx1 and tx2 will both increment their local version numbers and the global version numbers concurrently. As a result, the local and global version numbers validated in the second pass might not match for either transaction. In such rare cases, both transactions will abort. This also explains why a CAS operation is not required to increment the GlobalBinVerCounters value in the WriterBins validation algorithm. While the effect of some increments on the global counters might be lost due to the non-atomic ++ operation,

these are caught in the second pass when the global version numbers do not match the safely incremented transaction's local version numbers.

In contrast to all the other algorithms presented in this chapter, such conflict is only possible in the WriterBins algorithm, where readers are invisible and writers are nonexclusive. In the VisibleReaders algorithm, the write operations detect the previously registered readers and invoke the contention manager to resolve the conflict. In the WarningWord algorithm the second writer is warned by the first writer to abort. In the BloomFilter algorithm, reads are noted in the filter and the first writer would warn the prior readers.

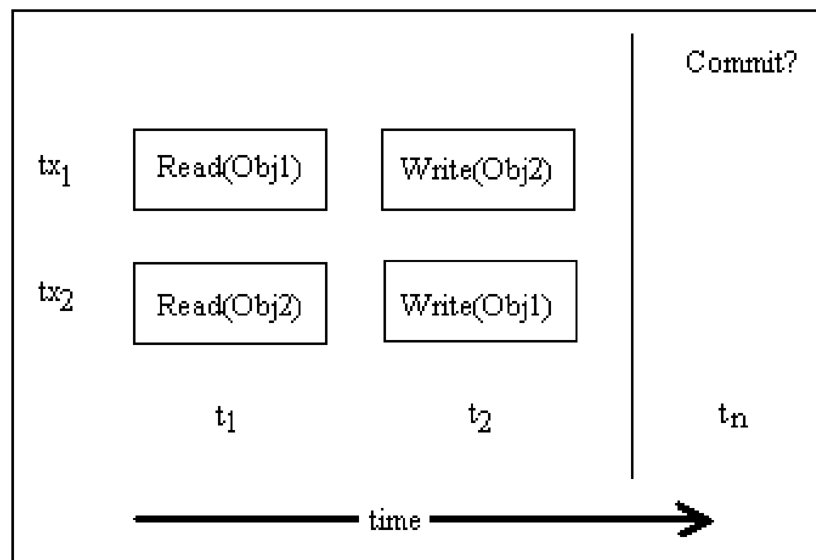



Figure 5.4: Nonlinearizable transactions

In transaction validation algorithms like WriterBins, how much validation to do during transactional reads and writes is an open question. Obviously, the less validation done incrementally during the execution of a transaction, the faster read and write

operations will execute. The downside, of course, is that some percentage of those transactions will need to be aborted when they attempt to commit. Our approach currently takes the middle ground by checking bin version number only on the initial read of a bin, but the tradeoffs involved require further research.

## 5.6 Comparisons

Figure 5.5 compares the four algorithms by their space requirements, the number of CAS operations required on first read, first write, and at commit, and whether validation is required at commit time.

| | **Visible Readers** | **Warning Word** | **Bloom Filter** | **Writer Bins** |
|---|---|---|---|---|
| **Space** | 20 bytes / reader | 32 bits | 256KB | 256 bytes |
| **# of CAS for 1st read** | 1 | 0 | 1 | 0 |
| **# of CAS for 1st write** | 1 | 2 | 2 | 2 |
| **# of CAS on commit** | 1 | 2 | num of reads + 2 | 1 |
| **Validation required?** | No | No | No | Yes |

Figure 5.5: Comparison of the algorithms

The table shown in Figure 5.6 summarizes the level of contention management support provided by each of the four algorithms presented.

| | **Contention management** |
|---|---|
| **VisibleReaders** | Conflicting transaction resolved by the contention manager. |
| **WarningWord** | Writers proceed causing all readers to abort. Writer-to-writer conflicts are resolved by the contention manager. |
| **BloomFilter** | Writers proceed causing only highly probable conflicting readers to abort. Writer-to-writer conflicts are resolved by the contention manager. |
| **WriterBins** | Some nonlinearizable transactions allowed to proceed to commit-time validation. Whoever commits first wins. Reader/writer-to-writer conflicts are resolved by the contention manager. |

Figure 5.6: Support for contention management

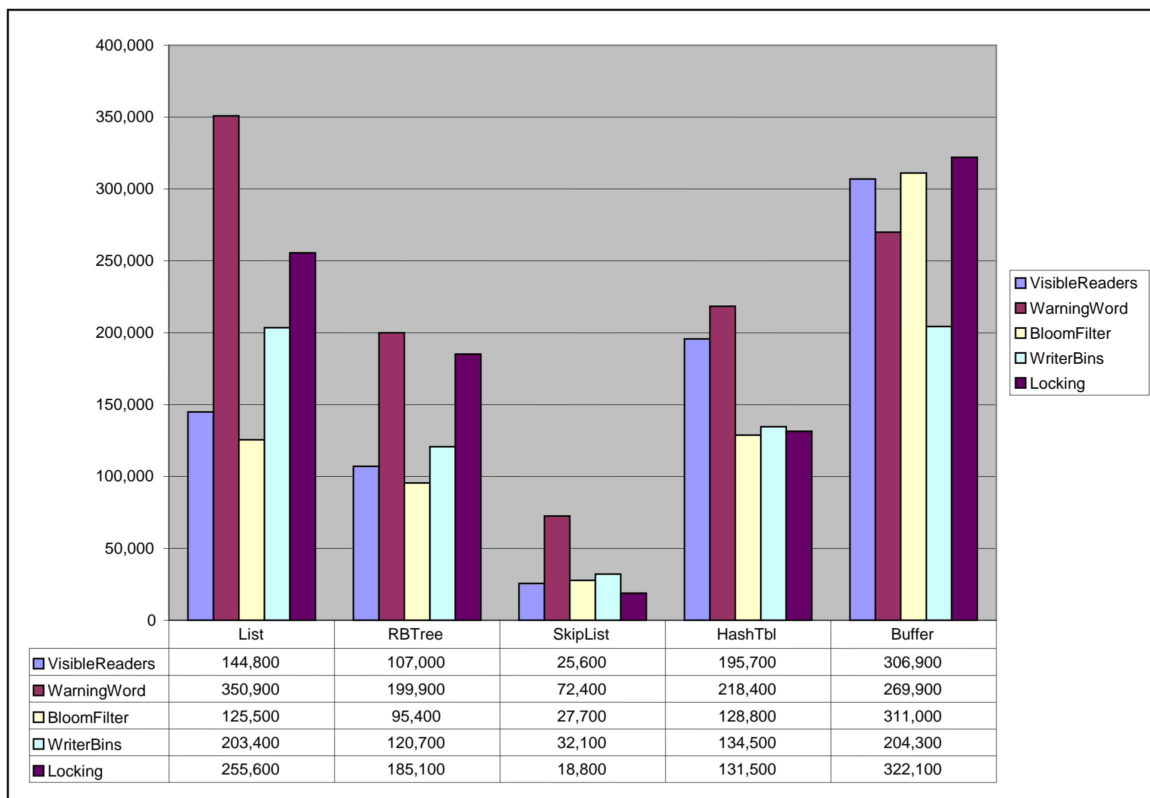
The graphs shown in Figures 5.7 and 5.8 compare the performance of our transaction synchronization algorithms when executing several standard benchmarks. The results in Figure 5.7 show the results of five standard algorithms running with low contention (2 threads with 30% write operations). The results are also compared against an STM mode using short locks. Figure 5.8 compares the performance of the List benchmark at increasing levels of concurrency with all four obstruction-free algorithms and the blocking model. These tests were all run on a 4-way machine.

Figure 5.8 shows the performance of the List benchmark under increasing levels of contention. We picked this specific benchmark because there is no transactional concurrency that can be exploited in a sorted list: a single writer will by necessity conflict with all other transactions as it will have read all list entries up to the point of the entry it intends to add. Thus, this figure shows that though performance degrades as contention

increases, the WriterBins algorithm performs at least as well as the STM mode that uses short lived locks.

Overall, Figures 5.7 and 5.8 show that the obstruction-free WarningWord algorithm performs slightly better than the blocking mode using short-lived locks. This is due to its extremely low overhead, and the fact that the blocking STM mode requires an additional CAS operation to acquire the lock. Despite the fact that the WarningWord algorithm performs best, and is more scalable than we expected, it nevertheless still suffers from considerable drawbacks because of its lack of any write concurrency. By contrast, the WriterBins algorithm comes closest to the goal of an innovative way of solving the invisible readers problem in a highly efficient way without sacrificing write concurrency or consistency. The WriterBins algorithm comes in second in terms of performance, but, on a system with more cores, the additional write concurrency enabled by the WriterBins algorithm may make it significantly more competitive.

All the algorithms presented in this chapter support three key properties: they include a nonblocking progress condition, allow for consistent reads, and support contention management. Together, they show that obstruction-free transactional memory systems can equal or surpass the performance of those that use short locks.




| | List | RBTree | SkipList | HashTbl | Buffer |
|---|---|---|---|---|---|
| VisibleReaders | 144,800 | 107,000 | 25,600 | 195,700 | 306,900 |
| WarningWord | 350,900 | 199,900 | 72,400 | 218,400 | 269,900 |
| BloomFilter | 125,500 | 95,400 | 27,700 | 128,800 | 311,000 |
| WriterBins | 203,400 | 120,700 | 32,100 | 134,500 | 204,300 |
| Locking | 255,600 | 185,100 | 18,800 | 131,500 | 322,100 |

Figure 5.7: Performance comparison with low contention

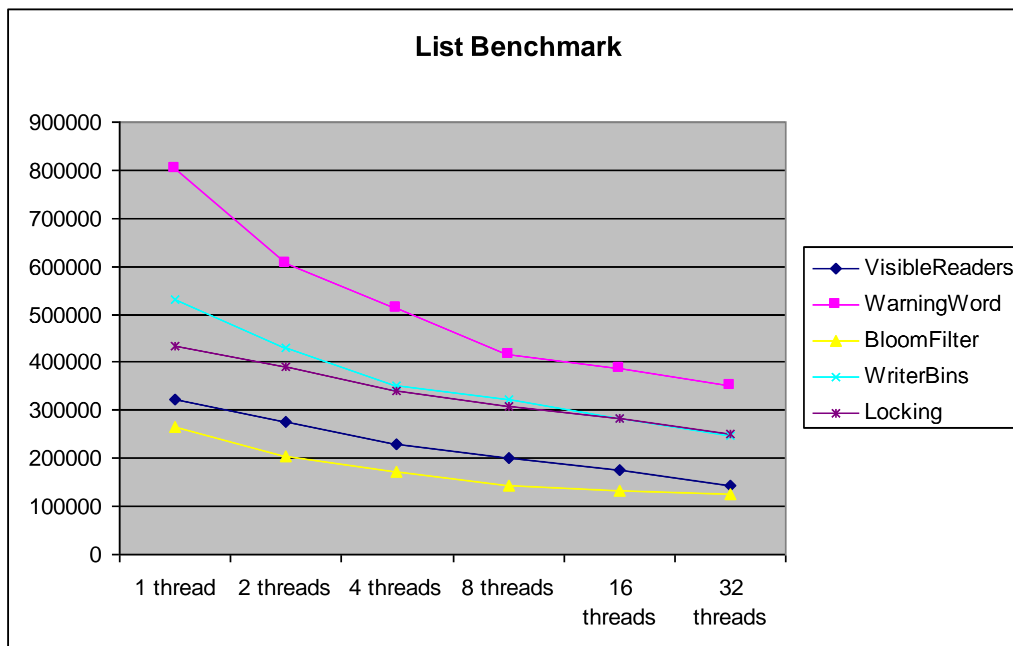



Figure 5.8: List benchmark at increasing levels of contention

# Chapter 6

# Evaluation

This thesis claims two fundamental results with respect to an optimizing compiler designed to support STM. First, we argue that STM systems can be made easier for programmers to use via an attribute-based language interface. Second, we claim that with a good mix of library and compiler optimizations, the performance of object-based, obstruction-free STMs with always consistent reads, do not perform worse than other types of STMs that do not support these properties. Both claims are difficult to prove. Ease-of-use in a programming model is notoriously amorphous and consequently difficult to measure. And, while performance can be more easily quantified, most current STMs are closed worlds that are difficult to compare with one another. For example, comparing

the STAMP [61] benchmarks originally written in C to run on the TL2 STM system to a C# port of those benchmarks running on our framework is not an apples-to-apples comparison. With these caveats in mind, this chapter evaluates our framework in the area of programmability; most of the performance results of this thesis are reported in relation to the specific benchmarks described in Chapter 4.

We are not the first to point out the challenges of library-based STM programming. In [14], Michael Scott et al. describe their frustration with a C++ STM library that relies on smart pointers for trapper transactional memory accesses. All library-based solutions to parallel programming suffer to some extent from this mismatch with languages fundamentally designed for single-threaded programming. Java was the first mainstream programming language to introduce synchronization directly in the programming language; even in Java, however, threads are library-based objects. While our approach to transactional memory does not obviate the need for explicit thread management, it does completely relieve the programmer of responsibility for any data protection beyond the declarative (i.e., marking objects as atomic).

As a starting point, below is the Insert method of the List benchmark as written for the Peet compiler. Note that other than the atomic annotation indicating that this method begins a new transaction, there is nothing else that the programmer must do to use the STM.

```
// Add a value to the set
[Atomic(XKind.Starts)]
public override bool Insert(int v) {
    Neighborhood hood = Find(v);
    if(hood.currNode != null)
        return false;

    Node newNode = new Node(v);
    Node prevNode = hood.prevNode;
```

```
        newNode.next = prevNode.next;
        prevNode.next = newNode;
        return true;
    }
```

Compare the code above to that the programmer would right using the SXM library alone:

```
    // method takes an array of objects and
    //    returns an object
    public override object Insert(params object[] _v) {
        // transaction creation is handled by the XStart
        //    delegate
        int v = (int)_v[0];     // unpack the parameters

        // instantiation not done with new -
        //    special factory creator method
        Node newNode = (Node)factory.Create(v);

        Neighborhood hood = Find(v);
        if(hood.currNode != null)
            return false;

        Node prevNode = hood.prevNode;

        // use field wrappers
        //    ('Next' instead of 'next')
        newNode.Next = prevNode.Next;

        prevNode.Next = newNode;
        return true;
    }
```

While this is not very different, certainly the convenience of controlling method signatures and not having to unpack parameters from a variable length array of objects is a significant ease-of-use feature. Furthermore, in order to wrap this method is a transaction, the programmer must, use the XStart delegate as shown here:

```
    // initialize the transaction object
    insertXStart = new XStart(this.Insert);
```

```
// run the tx
bool retval = (bool)XAction.Run(insertXStart,
                                value);
```

Starting and ending (committing or aborting) the transaction, and retrying the transaction in case of an abort, is a separate procedure that library manages, or the programmer can customize:

```
// run a method in a transaction
//    start - XStart Delegate to run
//    args  - array of arguments to pass to delegate
//    returns result returned by delegate (may be null)
    public static object Run(XStart start,
                     params object[] args) {
        object result = null;

        while(true) {
            // start the new transaction
            TxStart(XKind.Requires);
            try {
                // invoke the user's delegate
                result = start(args);
                if(current.Commit())
                    return result;
            }
            catch(AbortedException) {
                // aborted by synch conflict, retry
                TxHandleAbort(current);
            }
            catch(Exception e) {
                me.Abort();  // abort the transaction
                throw e;     // rethrow the errror
            }
            finally {
                TxFinally(current);
            }
        }
    }
```

In the Peet model, all of these basic transactional chores are handled automatically by the compiler, and customized for the types and method signature of the

user's code. Furthermore, when compared with earlier model like DSTM, the programmability of the Peet model because even more apparent. For comparison, below is the insert method written for the DSTM library (this algorithm is described in detail in § 2.5).

```
public boolean insert(int v) {
    // (1) List must implement TMCloneable
    List newList = new List(v);

    // (2) need to wrap the target object in a TMObject
    TMObject newNode = new TMObject(newList);

    TMThread thread = (TMThread)Thread.currentThread();

    // (3) loop to retry the transaction in case it fails
    while(true) {

        // (4) start the transaction
        thread.beginTransaction();

        boolean result = true;
        try {

            // (5) open the objects in read/write mode
            List prevList =
                (List)first.open(TMObject.WRITE);
            List currList =
                (List)prevList.next.open(TMObject.WRITE);

            while(currList.value < v) {
                prevList = currList;
                currList =
                    (List)currList.next.open(
                                               TMObject.WRITE);
            }

            if(currList.value == v) {
                result = false;
            }
            else {
                result = true;
                newList.next = prevList.next;
                prevList.next = newNode;
            }
```

```
        }
        catch(Denied d) { }

        // (6) attempt to commit the transaction
        if(thread.commitTransaction())
            return result;
    }
    return false;
}
```

It is important to realize that both the first generation DSTM model and the second generation SXM/DSTM2 models each has certain strength and weaknesses. The weakness of the original DSTM model is its challenging programming model which exposes much of the underlying transactional machinery to the programmer. The advantage of this approach, however, is that it makes clever optimizations possible. For example, the programmer can open a transactional object for reading or writing, and then hold onto that reference for later use in the method. So long as the programmer is certain that later accesses in the code first pass through an "open" operation, this subsequent use is safe in the obstruction-free model. In the STM model that uses short-locks, however, this approach is safe only for reads, and even then only where the programmer subsequently checks that the transaction is still active before using read the value. Finally the DSTM model supports techniques like *early release*, which allows read objects to be dropped from the transaction's read set, where the programmer is certain that object will not be used again in the same transaction.

These are powerful tools in the expert programmer's arsenal, but certainly do not change the general perception that concurrent programming is challenging and tends to yield fragile code that is difficult to maintain. By contrast, the SXM and DSTM2 models significantly simplify programming with transactional memory. At the same time,

however, this simplified programming model excludes nearly any possible optimizations by the programmer. All accesses to atomic objects are routed through the full transactional memory machinery, whether it is necessary or not. This happens because SXM requires that all fields of atomic objects be implemented as property procedures with getter and setter methods. For example, a simple integer field of atomic object in SXM must be implemented by the programmer as shown here:

```
protected int value;

public virtual int Value {
    get {
        return this.value;
    }
    set {
        this.value = value;
    }
}
```

The Peet model, by contrast, further improves on the programmability of the SXM and DSTM2 models. Indeed, it offers what can be thought of as the third generation of transactional programming models. At the same time, the compiler model automatically optimizes the use of the STM library, generating code that is in most cases more efficient than that written by programmers, while guaranteeing that those optimizations are correct (i.e., safe). Although number of lines of code is a crude measure of programmability, all other things being equal, a more concise expression of an equivalent algorithm is generally considered superior. By that measure, our STM programming model is also superior to its predecessors. The RBTree algorithm as implemented in DSTM was 776 lines of code; in SXM it was 705 lines of code; in Peet, the RBTree algorithm is just 538 lines of code, more than 30% shorter than the DSTM version.

# Chapter 7

# Conclusions

This thesis presents language support and compiler optimizations designed specifically for an object-based STM library. We show that compiler support is extremely beneficial in terms of improving performance and programmability. By introducing atomic types, methods, and blocks as annotation-based language features, the compiler supports a natural and nearly transparent language interface, thus easing the burden an STM library places on the programmer. By applying object-based dataflow optimizations and improving the underlying STM library, the compiler achieves significant performance gains over a purely library-based approach. Our work reveals that in STM models that use short locks, only the initial access to an atomic object requires the

acquisition of a lock; subsequent accesses can be made wait-free. We also show that, contrary to recent thinking [23], obstruction-free STM systems do not perform worse than those that employ short locks. Finally, we introduce the first compiler support for transactional boosting, a methodology for transforming highly concurrent linearizable objects into highly-concurrent transactional objects. Based on these results, we conclude that appropriate language support and high quality compiler optimizations are necessary for the success of any STM system.

## 7.1 Future work and open questions

STM is an area of active research. The very semantics of STM are still being defined and the correct programming language model has only recently begun to be considered. It is clear, however, that STM systems will require some degree of programmer involvement, and thus some language constructs will be necessary. High performance STM systems will require the cooperation of compilers, execution environments and possibly even hardware, either via basic instruction-level support or perhaps awareness and optimization of STM execution patterns. This thesis proposes simple object-based language constructs for STM and static compiler optimizations that build on the language proposal. Further optimization is almost certainly possible at the runtime level, which has dynamic information about the actual execution of the program. Besides these very general areas of work, some specific area of STM semantics, including exceptions and nesting, are still somewhat open despite the existence of several proposals. While it is ultimately unlikely that transactional memory will replace all uses of mutual exclusion, it seems probable that it can more efficiently and safely replace

most. Getting to that point will likely require addressing these and other, yet to be asked, questions. Just as virtual memory and garbage collection took many years of research to get to the point of widespread commercial use, transactional memory will likely have to go through a similar process of development and refinement. The result will undoubtedly make concurrent programming easier, safer, and faster.

# Bibliography

[1] Adl-Tabatabai, Ali-Reza; Lewis, Brian T.; Menon, Vijay; Murphy, Brian R.; Saha, Bratin and Shpeisman, Tatiana. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Programming Language Design and Implementation* (PLDI) (June 2006).

[2] Attiya, Hagit; Guerraoui, Rachid; Handler, Denny and Kouznetsoc, Petr. Synchronizing without Locks is Inherently Expensive. In *Principles of Distributed Computing* (July 2006).

[3] Barnes, Greg. A Method for Implementing Lock-Free Shared Data Structures. In *Symposium on Parallel Algorithms and Architectures* (1993).

[4] Bloom, Burton H. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM 13*, 7 (July 1970), 422–426.

[5] Blundell, Colin; Lewis, Christopher E. and Martin, Milo M. K. Subtleties of Transactional Memory Atomicity Semantics. *Manuscript*.

[6] Blundell, Colin; Lewis, Christopher E. and Martin, Milo M. K. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[7] Carey, Michael J.; DeWitt, David J. and Naughton, Jeffrey F. The OO7 Benchmark. In *ACM Special Interest Group on Management of Data* (SIGMOD), 1993.

[8] Carlstrom, Brian D.; McDonald, Austen; Kozyrakis, Christos and Olukotun, Kunle. Transactional Collection Classes. In *Principles and Practice of Parallel Programming* (2007)

[9] Choi, Jong-Deok; Grovel, David; Hind, Michael and Sarkar, Vivek. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pp. 21-31, 1999.

[10] Choi, Jong-Deok; Gupta, Manish; Serrano, Mauricio J.; Sreedhar, Vugranam C. and Midkiff, Samuel P. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. In *ACM Transactions on Programming Languages and Systems* (November 2003).

[11] Chow, Fred; Chan, Sun; Kennedy, Robert; Liu, Shin-Ming; Lo, Raymond and Tu, Peng. A New Algorithm for Partial Redundancy Elimination based on SSA Form. In *Programming Language Design and Implementation* (1997).

[12] Cole, Christopher and Herlihy, Maurice. Snapshots and Software Transactional Memory. In *Concurrency and synchronization in Java programs* (2005).

[13] Ceze, Luis; Tuck, James; Cascaval, Calin and Torrellas, Josep. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *International Symposium on Computer Architecture*, 2006.

[14] Dalessandro, L., Marathe, V., Spear, M. and Scott, Michael. Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR. August 2007.

[15] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid Transactional Memory. In *Proceedings of the 12th international Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) October 2006.

[16] Dice, David; Shalev, Ori and Shavit, Nir. Transactional Locking II. In *International Symposium on Distributed Computing* (DISC) (September 2006).

[17] Dice, David and Shavit, Nir. What Really Makes Transactions Faster? In *Transact* 2006.

[18] Discolo, Anthony; Harris, Tim; Jones, Simon Peyton and Singh, Satnam. Lock Free Data Structures using STM in Haskell. In the *Eighth International Symposium on Functional and Logic Programming* (FLOPS) (April 2006).

[19] Doherty, Simon; Detlefs, David, L.; Groves, Lindsay; Flood, Christine F.; Luchangco, Victor; Martin, Paul A.; Moir, Mark; Shavit, Nir and Steele, Guy L. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. In *Symposium on Parallelism in Algorithms and Architectures* (SPAA) 2004.

[20] Doherty, Simon; Herlihy, Maurice; Luchangco, Victor and Moir, Mark. Bringing Practical Lock-Free Synchronization to 64-Bit Applications. In *Symposium on Principles of Distributed Computing* (July 2004).

[21] Dwyer, Matthew B. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *Symposium on Foundations of Software Engineering* (1994).

[22] Ellen, Faith; Lev, Yossi; Luchangco, Victor and Moir, Mark. SNZI: Scalable NonZero Indicators. In *Principles of Distributed Computing* (August 2007).

[23] Ennals, Robert. Software Transactional Memory Should Not Be Obstruction-Free. Unpublished manuscript. Intel Research Cambridge, 2005.

[24] Fan, Li; Cao, Pai; Almeida, Jussara and Broder, Andrei Z. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proceedings of SIGCOMM '98*, 1998.

[25] Felber, Pascal; Fetzer, Christof; Muller, Ulrich; Riegel, Torvald; Subkraut, Martin and Sturzrehm, Heiko. Transactifying Applications using an Open Compiler Framework. In *Transact* (2007).

[26] Fich, Faith; Handler, Danny and Shavit, Nir. On the Inherent Weakness of Conditional Synchronization Primitives. In *Symposium on Principles of Distributed Computing* (July 2004).

[27] Genaim, Samir and Spoto, Fausto. Information Flow Analysis for Java Bytecode. 2005.

[28] Greenwald, Michael. Two-Handed Emulation: How to build nonblocking implementations of complex data-structures using DCAS. In *Principles of Distributed Computing* (2002).

[29] Grossman, Dan. The Transactional Memory / Garbage Collection Analogy. In *Object-Oriented Programming, Systems, Languages & Applications* (OOPSLA) (October 2007).

[30] Guerraoui, Rachid and Kapalka, Michal. On the Correctness of Transactional Memory. In *Principles and Practice of Parallel Programming* (PPoPP) (February 2008).

[31] Guerraoui, Rachid; Kapalka, Michal and Vitek, Jan. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2007 conference on EuroSys* (2007).

[32] Hammond, Lance; Wong, Vicky; Chen, Mike; Carlstrom, Brian D.; Davis, John D.; Hertzberg, Ben; Prabhu, Manohar K.; Wijaya, Honggo; Kozyrakis, Christos and Olukotun, Kunle. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International. Symposium on Computer Architecture*, June 2004.

[33] Harris, Tim. Exceptions and Side-Effects in Atomic Blocks. In *Workshop on Concurrency and Synchronization in Java Programs* (July 2004).

[34] Harris, Tim and Fraser, Keir. Language Support for Lightweight Transactions. In *Object-Oriented Programming Systems, Languages and Applications* (OOPSLA) (October 2003), pp. 388-402.

[35] Harris, Tim and Fraser, Keir. Revocable Locks for Non-Blocking Programming. In *Principles and Practice of Parallel Programming* (June 2005).

[36] Harris, Tim; Fraser, Keir and Pratt, I. A. A Practical Multi-word Compare-and-swap Operation. In *Proceedings of the 16th International Symposium on Distributed Computing* (Oct. 2002), pp. 265-279.

[37] Harris, Tim; Marlow, Simon; Jones, Simon Peyton and Herlihy, Maurice. Composable Memory Transactions. In *Principles and Practice of Parallel Programming* (PPoPP) (June 2005).

[38] Harris, Tim; Plesko, Mark; Shinnar, Avraham and Tarditi, David. Optimizing Memory Transactions. In *Programming Language Design and Implementation* (PLDI) (June 2006).

[39] Heffner, Kelly; Tarditti, David and Smith, Michael D. Extending Object-Oriented Optimizations for Concurrent Programs. In *Parallel Architecture and Compilation Techniques* (2007).

[40] Herlihy, Maurice. A Methodology for Implementing Highly Concurrent Data Objects. In *ACM Transactions on Programming Languages and Systems*, 15(5):745-770, 1993.

[41] Herlihy, Maurice. SXM 1.1 Software Transactional Memory Package for C#. http://www.cs.brown.edu/~mph.

[42] Herlihy, Maurice. Wait-Free Synchronization. In *ACM Transactions on Programming Languages and Systems,* 13(1):124-149, 1991.

[43] Herlihy, Maurice and Wing, Jeanette M. Axioms for Concurrent Objects. In *Symposium on Principles of Programming Languages* (1987).

[44] Herlihy, Maurice and Wing, Jeanette M. Linearizability: A Correctness Condition for Concurrent Objects. In *Transactions on Programming Languages and Systems* (July 1990).

[45] Herlihy, Maurice and Koskinen, Eric. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. Brown CS Technical Report CS-07-08 (July 2007).

[46] Herlihy, Maurice; Luchangco, Victor and Moir, Mark. Obstruction-free Synchronization: Double-ended Queues as an Example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems* (2003).

[47] Herlihy, Maurice; Luchangco, Victor and Moir, Mark. A Flexible Framework for Implementing Software Transactional Memory. OOPSLA 2006.

[48] Herlihy, Maurice; Luchangco, Victor; Moir, Mark and Scherer, William. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (July 2003), pp. 92-101.

[49] Herlihy, Maurice and Moss, J. Elliot B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), IEEE Computer Society Press, pp. 289-301.

[50] Herlihy, Maurice and Sun, Ye. Distributed Transactional Memory for Metric-Space Networks. In *Proceedings of the 19th International Symposium on Distributed Computing* (DISC) (Sep 2005), P. Fraigniaud, Ed., Springer, pp. 324-338.

[51] Hill, Mark D.; Hower, Derek; Moore, Kevin E.; Swift, Michael M.; Volos, Harris and Wood, David A. A Case for Deconstructing Hardware Transactional Memory Systems. In *University of Wisconsin Computer Sciences Technical Report CS-TR-2007-1594* (June 2007).

[52] Hindman, Benjamin and Grossman, Dan. Strong Atomicity for Java Without Virtual-Machine Support. In *Report 2006-05-01, University of Washington Dept. of Computer Science & Engineering*, May 2006.

[53] Hindman, Benjamin and Grossman, Dan. Atomicity via Source-to-Source Translation. In *Memory Systems Performance and Correctness* (October 2006).

[54] Hoare, Anthony. Towards a theory of parallel programming. In *Operating Systems Techniques, vol. 9 of A.P.I.C Studies in Data Processing,* Academic Press, pp. 61-71. (1972).

[55] Hudson, Richard L.; Saha, Bratin; Adl-Tabatabai, Ali-Reza and Hertzberg, Benjamin C. McRT-Malloc – A Scalable Transactional Memory Allocator. In *International Symposium on Memory Management* (June 2006).

[56] Kennedy, Robert; Chan, Sun; Liu, Shin-Ming; Lo, Raymond; Tu, Peng and Chow, Fred. Partial Redundancy Elimination in SSA Form. In *Transactions on Programming Languages and Systems* (TOPLAS) (May 1999).

[57] Kung, H. T. and Papadimitriou, Christos. An optimality theory of concurrent control for databases. In *Proceedings of the ACM CIGMOD international conference on Management of data.* May, 1979.

[58] Lamport, Leslie and Owicki, Susan. Proving Liveness Properties of Concurrent Programs. In *ACM Transactions on Programming Languages and Systems 4, 3* (July 1982), 455-495.

[59] Liskov, Barbara and Scheifler, R. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. In *ACM Transactions on Programming Languages* (July 1983), pp. 381-404.

[60] Luchangco, Victor; Moir, Mark and Shavit, Nir. Nonblocking k-compare-single-swap. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2003.

[61] Luchangco, Victor and Marathe, Virendra J. Transaction Synchronizers. In *Synchronization and Concurrency in Object-Oriented Languages* (2005).

[62] Marathe, Virendra J.; Spear, Michael F.; Heriot, Christopher; Acharya, Athul; Eisenstat, David; Scherer, William N. and Scott, Michael L. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, PLDI (June 2006).

[63] Marathe, Virendra J.; Scherer, William N. and Scott, Michael L. Adaptive Software Transactional Memory. *19th International. Symposium on Distributed Computing*, (September 2005).

[64] Michael, Maged M. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Symposium on Parallel Algorithms and Architectures* (SPAA) (August 2002).

[65] Michael, Maged M. and Scott, Michael L. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Symposium on Principles of Distributed Computing (PODC)*, pp. 267-275. 1996.

[66] Minh, Chi Cao; Trautmann, Martin; Chung, JaeWoong; McDonald, Austen; Bronson, Nathan; Casper, Jared; Kozyrakis, Christos and Olukotun, Kunle. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (June 2007).

[67] Moore, Kevin E.; Bobba, Jayaram; Moravan, Michelle J.; Hill, Mark D and Wood, David A. LogTM: Log-based Transactional Memory. In *High-Performance Computer Architecture* (February 2006).

[68] Morel, E. and Renvoise, C. Global Optimization by Suppression of Partial Redundancies. In *Communications of the ACM* (February 1979).

[69] Ni, Yang; Menon, Vijay; Adl-Tabatabai, Ali-Reza; Hosking, Antony L.; Hudson, Richard L.; Moss, J. Elliot B.; Saha, Bratin and Shpeisman, Tatiana. Open Nesting in Software Transactional Memory. In *Principles and Practice of Parallel Programming* (PPoPP) (March 2007).

[70] Riegel, Torvald; Felber, Pascal and Fetzer, Christof. A Lazy Snapshot Algorithm with Eager Validation. In *Symposium Parallel Algorithms and Architectures* (2007).

[71] Riegel, Torvald; Fetzer, Christof; Sturzrehm, Heiko and Felber, Pascal. From Causal to z-Linearizable Transactional Memory. In *Principles of Distributed Computing* (August 2007).

[72] Ringenburg, Michael F. and Grossman, David. AtomCaml: First-Class Atomicity via Rollback. In *International Conference on Functional Programming* (September 2005).

[73] Saha, Bratin; Adl-Tabatabai, Ali-Reza; Ghuloum, Anwar; Rajagopalan, Mohan; Hudson, Richard L.; Peterson, Leaf; Menon, Vijay; Murphy, Brian; Shpesisman, Tatiana; Fang, Jesse; Sprangle, Eric; Rohillah, Anwar and Carmean, Doug. Enabling Scalability and Performance in a Large Scale Chip Multiprocessor Environment. In *Proceedings of the 2007 conference on EuroSys* (2007).

[74] Saha, Bratin; Adl-Tabatabai, Ali-Reza; Hudson, Richard L.; Minh, Chi Cao and Hertzberg, Benjamin. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Principles and Practice of Parallel Programming* (PPoPP) (March 2006).

[75] Scherer, William N. and Scott, Michael L. Nonblocking Concurrent Data Structures with Condition Synchronization. In *Symposium on Distributed Computing* (October 2004).

[76] Scherer, William N. and Scott, Michael L. Advanced Contention Management for Dynamic Software Transactional Memory. In *Principles of Distributed Computing* (July 2005).

[77] Shavit, Nir and Touitou, Dan. Software transactional memory. *Distributed Computing,* Special Issue (10):99-116, 1997.

[78] Shriraman, Arrvindh; Marathe, Virendra J.; Dwarkadas, Sandhya; Scott, Michael L.; Eisenstat, David; Heriot, Christopher; Scherer, William N. and Spear, Michael F. Hardware Acceleration of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing* (June 2006).

[79] Singh, Satnam. Higher Order Combinators for Join Patterns using STM. In *Transact* (2006).

[80] Spear, Michael F.; Marathe, Virendra J.; Dalessandro, Luke and Scott, Michael L. Privatization Techniques for Software Transactional Memory. In *Department of Computer Science, University of Rochester, Technical Report #915* (February 2007).

[81] Spear, Michael F.; Shriraman, Arrvindh; Dalessandro, Luke; Dwarkadas, Sandhya and Scott, Michael L. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Symposium on Parallelism in Algorithms and Architectures* (June 2007).

[82] Stone, Stone, Heidelberger, and Turek. Multiple Reservations and the Oklahoma Update.

[83] Sutter, Herb and Lurus, James. Software and the Concurrency Revolution. In ACM Queue, Vol. 3, No. 7, pp 54–62, September 2005.

[84] Vafeiadas, Viktor; Herlihy, Maurice; Hoare, Tony and Shapiro, Marc. Proving Correctness of Highly-Concurrent Linearisable Objects. In *Principles and Practice of Parallel Programming* (PPoPP) pp. 129-136 (March 2006).

[85] Vafeiadas, Viktor; Herlihy, Maurice; Hoare, Tony and Shapiro, Marc. A safety proof of a lazy concurrent list-based set implementation. University of Cambridge Technical Report N. 659 (UCAM-CL-TR-659) (January 2006).

[86] VanDrunen, Thomas and Hosking, Antony L. Anticipation-based partial redundancy elimination for static single assignment form. In *Software—Practice and Experience,* 2002.

[87] Welc, Adam; Jagannathan, Suresh and Hosking, Antony. Safe Futures for Java. In *Object-Oriented Programming, Systems, Languages & Applications* (OOPSLA) (October 2005).

[88] Yen, Luke; Bobba, Kayaram; Marty, Michael R.; Moore, Kevin E.; Volos, Haris; Hill, Mark D.; Swift, Michael M. and Wood, David A. LogTM-SE: Decoupling

Hardware Transactional Memory from Caches. In *International Symposium on High Performance Computer Architecture* (Feb. 2007).

[89] Zilles, Craig and Rajwar, Ravi. Transactional Memory and the Birthday Paradox. In *Symposium on Parallelism in Algorithms and Architectures* (June 2007).

# Appendix

# Selected Algorithms

This appendix contains C# implementations of the synchronization algorithms for the STM model that uses short locks, as well as the four obstruction-free algorithms described in Chapter 5.

**The lock-based synchronization algorithm described in § 4.3**

```
// synchronization state for transactional objects
public struct TMemSynchState {
    // default size of the readers array
    private const int DEFAULT_NUM_START_READERS = 8;

    // this is the state maintaned for each atomic object
    private XState writer;          // the writer, if any
    private XState[] readers;       // resizable array of readers
    private int numReaders;         // curr # of readers
    private int lockState;          // a reader-writer lock
    private readonly IRecoverable atomicObj; // ref to the object

    enum lockStates {
        WriteExclusive = 1,
        Free = 0
    }

    #region constructors
    public TMemSynchState(IRecoverable atomicObject) {
        readers = new XState[DEFAULT_NUM_START_READERS];
        atomicObj = atomicObject;
        writer = XAction.ClosedTx;
        lockState = (int)lockStates.Free;
        numReaders = 0;
    }

    // overload that allows you to set the initial # of readers
    public TMemSynchState(IRecoverable atomicObject,
                          int numStartReaders) {
        readers = new XState[numStartReaders];
        atomicObj = atomicObject;
        writer = XAction.ClosedTx;
        lockState = (int)lockStates.Free;
        numReaders = 0;
    }
    #endregion
    #region debug functions
    // display for debugging
    public override string ToString() {
        return string.Format(
         "SynchState{0}[writer: {1}, readCount: {2}, object: {3}]",
           this.GetHashCode(), writer, numReaders,
           atomicObj.GetHashCode());
    }
    #endregion

    // never used by the compiler... (could be called manually)
    public void EarlyRelease(XState me) {
        for(int i = 0; i < numReaders; i++)
           if(readers[i] == me)
           Interlocked.CompareExchange(ref readers[i], null, me);
        // try to remove myself as a reader
        // no harm done if it didn't work...
    }
```

```
// for atomic fields
#region OpenReadIntegrated
public T OpenReadIntegrated<T>(XState me, ref T field) {
    if(me == null) { // for non-tx reads
        if(writer.State == XStates.ABORTED) {
            atomicObj.Restore();
             // restore object if last writer aborted
            writer = XAction.ClosedTx;
        }
        return field;
    }

    T value;
    if(me.root == writer.root) { // do i already own it?
        value = field; // do the read
        if((XStates)me.state == XStates.ABORTED)
            // check to make sure the read was "good"
            throw XAction.AbortedException;
        return value;
    } // post-state: i'm not the writer

Loop:
    if((XStates)me.state == XStates.ABORTED)
        throw XAction.AbortedException;

    if(Interlocked.CompareExchange(ref lockState,
      (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
         (int)lockStates.Free) {
        Thread.Sleep(0); // necessary to avoid livelock
        goto Loop;
    }

    switch((XStates)writer.state) {
        case XStates.ACTIVE:
            lockState = (int)lockStates.Free;
            XAction.Manager.ResolveConflict(me, writer);
            goto Loop;
        case XStates.ABORTED:
            atomicObj.Restore();
            break;
    }
    writer = XAction.ClosedTx; // always clean-up the writer

    value = field; // do the read

    // replaces first non-active reader with the current reader
    XState myRoot = me.root;
    for(int i = 0; i < numReaders; i++) {
        XState currReader = readers[i];
        if((XStates)currReader.state != XStates.ACTIVE ||
                  myRoot == currReader.root) {
            readers[i] = me; // install myself as a reader
            lockState = (int)lockStates.Free;
            return value;
        }
    }
```

```
        // if necessary, resize the readers buffer
        if(numReaders == readers.Length)
            Array.Resize(ref readers, readers.Length << 1);

        readers[numReaders++] = me;
        lockState = (int)lockStates.Free; // release the lock
        return value;
    }
    #endregion
    #region OpenWriteIntegrated
    public void OpenWriteIntegrated<T>(XState me, ref T field,
                                       T newValue) {
        if(me == null) { // for non-tx writes
            if(writer.State == XStates.ABORTED) {
                atomicObj.Restore();
                    // restore object if last writer aborted
                writer = XAction.ClosedTx;
            }
            field = newValue;
            return;
        }

        if(me.root == writer.root) { // am i the writer?
            if(Interlocked.Increment(ref me.abortLock) < 1) {
                // inc tx-wide abortLock
                // oh-oh, i've already been aborted
                throw XAction.AbortedException;
            }
            field = newValue; // do the write
            Interlocked.Decrement(ref me.abortLock);
                   // release the pro-life lock
            return;
        } // post-state: i'm not the writer

    Loop:
        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        // try to get the lock
        if(Interlocked.CompareExchange(ref lockState,
           (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
           (int)lockStates.Free) {
            Thread.Sleep(0); // necessary to avoid livelock
            goto Loop;
        }

        // am i in conflict with any other readers?
        XState myRoot = me.root;
        for(int i = 0; i < numReaders; i++) {
            XState currReader = readers[i];
            if((XStates)currReader.state == XStates.ACTIVE &&
               myRoot != currReader.root) {
                lockState = (int)lockStates.Free;
                    // release the lock
                XAction.Manager.ResolveConflict(me, currReader);
                goto Loop;
```

```
                }
            }
            numReaders = 0; // reset number of readers

            switch((XStates)writer.state) {
                case XStates.ACTIVE:
                    lockState = (int)lockStates.Free;
                        // release the lock
                    XAction.Manager.ResolveConflict(me, writer);
                    goto Loop;
                case XStates.COMMITTED:
                    atomicObj.Backup();
                    break;
                case XStates.ABORTED:
                    atomicObj.Restore();
                    break;
            }
            writer = me;          // install myself as the writer
            field = newValue;     // do the write
            lockState = (int)lockStates.Free; // release the lock
        }
        #endregion
        #region OpenWriteForReadIntegrated
        public T OpenWriteForReadIntegrated<T>(XState me, ref T field)
{
            if(me == null) { // for non-tx reads
                if((XStates)writer.state == XStates.ABORTED) {
                    // restore object if last writer aborted
                    atomicObj.Restore();
                    writer = XAction.ClosedTx;
                }
                return field;
            }

            T value;
            if(me.root == writer.root) { // am i the writer?
                value = field; // do the read
                if((XStates)me.state == XStates.ABORTED)
                    throw XAction.AbortedException;
                return value;
            } // post-state: i'm not the writer

    Loop:
            if((XStates)me.state == XStates.ABORTED)
                throw XAction.AbortedException;

            // try to get the write lock
            if(Interlocked.CompareExchange(ref lockState,
               (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
               (int)lockStates.Free) {
                Thread.Sleep(0); // necessary to avoid livelock
                goto Loop;
            }

            // am i in conflict with any other readers?
            XState myRoot = me.root;
            for(int i = 0; i < numReaders; i++) {
```

```
                XState currReader = readers[i];
                if((XStates)currReader.state == XStates.ACTIVE &&
                   myRoot != currReader.root) {
                    lockState = (int)lockStates.Free;
                    XAction.Manager.ResolveConflict(me, currReader);
                    goto Loop;
                }
            }
            numReaders = 0; // reset number of readers

            switch((XStates)writer.state) {
                case XStates.ACTIVE:
                    lockState = (int)lockStates.Free;
                       // release the lock
                    XAction.Manager.ResolveConflict(me, writer);
                    goto Loop;
                case XStates.COMMITTED:
                    atomicObj.Backup();
                    break;
                case XStates.ABORTED:
                    atomicObj.Restore();
                    break;
            }
            writer = me;             // install myself as the writer
            value = field;           // do the read
            lockState = (int)lockStates.Free; // release the lock
            return value;
        }
        #endregion

        // for atomic object pre-opens
        #region PreOpenReadIntegrated
        public void PreOpenReadIntegrated(XState me) {
            if(me == null) { // for non-tx reads
                if((XStates)writer.state == XStates.ABORTED) {
                      // restore object if last writer aborted
                    atomicObj.Restore();
                    writer = XAction.ClosedTx;
                }
                return;
            }

            if(me.root == writer.root) // do i already own the object?
                return;
        // post-state: i'm not the writer

    Loop:
            if((XStates)me.state == XStates.ABORTED)
                throw XAction.AbortedException;

            // try to get the lock
            if(Interlocked.CompareExchange(ref lockState,
               (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
               (int)lockStates.Free) {
                Thread.Sleep(0); // necessary to avoid livelock
                goto Loop;
            }
```

```
        switch((XStates)writer.state) {
            case XStates.ACTIVE:
                lockState = (int)lockStates.Free;
                XAction.Manager.ResolveConflict(me, writer);
                goto Loop;
            case XStates.ABORTED:
                atomicObj.Restore();
                break;
        }
        writer = XAction.ClosedTx; // always clean-up the writer

        // replaces first non-active reader with the current reader
        XState myRoot = me.root;
        for(int i = 0; i < numReaders; i++) {
            XState currReader = readers[i];
            if((XStates)currReader.state != XStates.ACTIVE ||
                 myRoot == currReader.root) {
                readers[i] = me; // install myself as a reader
                lockState = (int)lockStates.Free;
                return;
            }
        }

        // if necessary, resize the readers buffer
        if(numReaders == readers.Length)
            Array.Resize(ref readers, readers.Length << 1);

        readers[numReaders++] = me;  // install myself as a reader
        lockState = (int)lockStates.Free; // release the write lock
    }
    #endregion
    #region PreOpenWriteIntegrated
    public void PreOpenWriteIntegrated(XState me) {
        if(me == null) { // for non-tx writes
            if((XStates)writer.state == XStates.ABORTED) {
                     // restore object if last writer aborted
                atomicObj.Restore();
                writer = XAction.ClosedTx;
            }
            return;
        }

        if(me.root == writer.root) // am i the writer?
            return;

    Loop:
        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        // try to get the lock
        if(Interlocked.CompareExchange(ref lockState,
           (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
           (int)lockStates.Free) {
            Thread.Sleep(0); // necessary to avoid livelock
            goto Loop;
        }
```

```
    // am i in conflict with any other readers?
    XState myRoot = me.root;
    for(int i = 0; i < numReaders; i++) {
        XState currReader = readers[i];
        if((XStates)currReader.state == XStates.ACTIVE &&
               myRoot != currReader.root) {
            lockState = (int)lockStates.Free;
            XAction.Manager.ResolveConflict(me, currReader);
            goto Loop;
        }
    }
    numReaders = 0; // reset readCount - no active readers

    switch((XStates)writer.state) {
        case XStates.ACTIVE:
            lockState = (int)lockStates.Free;
                // release the lock
            XAction.Manager.ResolveConflict(me, writer);
            goto Loop;
        case XStates.COMMITTED:
            atomicObj.Backup();
            break;
        case XStates.ABORTED:
            atomicObj.Restore();
            break;
    }
    writer = me; // install myself as the writer
    lockState = (int)lockStates.Free; // release the lock
}
#endregion

// for atomic arrays
#region OpenReadArrayIntegrated
public T OpenReadArrayIntegrated<T>(XState me, int index) {
    IAtomicArray<T> castedAtomicArray =
          (IAtomicArray<T>)atomicObj;
    if(me == null) { // not in a transaction
        if((XStates)writer.state == XStates.ABORTED) {
                  // but maybe the last writer aborted
            atomicObj.Restore();
            writer = XAction.ClosedTx;
        }
        return castedAtomicArray[index];
    }

    T value;
    if(me.root == writer.root) {
               // do i already own the object?
        value = castedAtomicArray[index];
        if((XStates)me.state == XStates.ABORTED)
           // check to make sure that "value" has a good value
            throw XAction.AbortedException;
        return value;
    } // post-state: we're not related to the writer

Loop:
```

```
        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        // try to get the lock
        if(Interlocked.CompareExchange(ref lockState,
           (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
           (int)lockStates.Free) {
            Thread.Sleep(0); // necessary to avoid livelock
            goto Loop;
        }

        switch((XStates)writer.state) {
            case XStates.ACTIVE:
                lockState = (int)lockStates.Free;
                XAction.Manager.ResolveConflict(me, writer);
                goto Loop;
            case XStates.ABORTED:
                atomicObj.Restore();
                break;
        }
        writer = XAction.ClosedTx; // always clean-up the writer

        try {
            value = castedAtomicArray[index]; // do the read
        }
        catch(Exception e) {
            lockState = (int)lockStates.Free; // release the lock
            throw e;
        }

        // replaces first non-active reader with the current reader
        XState myRoot = me.root;
        for(int i = 0; i < numReaders; i++) {
            XState currReader = readers[i];
            if((XStates)currReader.state != XStates.ACTIVE ||
                      myRoot == currReader.root) {
                readers[i] = me; // install myself as a reader
                lockState = (int)lockStates.Free;
                return value;
            }
        }

        // if necessary, resize the readers buffer
        if(numReaders == readers.Length)
            Array.Resize(ref readers, readers.Length << 1);

        readers[numReaders++] = me;  // install myself as a reader
        lockState = (int)lockStates.Free; // release the write lock
        return value;
    }
    #endregion
    #region OpenReadArrayIntegrated // for GetEnumerator()
    // used by the array GetEnumerator calls
    public void OpenReadArrayIntegrated(XState me) {
        if(me == null) { // not in a transaction
            if((XStates)writer.state == XStates.ABORTED) {
               // but maybe the last writer aborted
```

```
                atomicObj.Restore();
                writer = XAction.ClosedTx;
            }
            return;
        }

        if(me.root == writer.root) // do i already own the object?
            return;
    // post-state: we're not related to the writer

Loop:
        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        // try to get the lock
        if(Interlocked.CompareExchange(ref lockState,
           (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
           (int)lockStates.Free) {
            Thread.Sleep(0); // necessary to avoid livelock
            goto Loop;
        }

        switch((XStates)writer.state) {
            case XStates.ACTIVE:
                lockState = (int)lockStates.Free;
                XAction.Manager.ResolveConflict(me, writer);
                goto Loop;
            case XStates.ABORTED:
                atomicObj.Restore();
                break;
        }
        writer = XAction.ClosedTx; // always clean-up the writer

        // replaces first non-active reader with the current reader
        XState myRoot = me.root;
        for(int i = 0; i < numReaders; i++) {
            XState currReader = readers[i];
            if((XStates)currReader.state != XStates.ACTIVE ||
               myRoot == currReader.root) {
                readers[i] = me; // install myself as a reader
                lockState = (int)lockStates.Free;
                return;
            }
        }

        // if necessary, resize the readers buffer
        if(numReaders == readers.Length)
            Array.Resize(ref readers, readers.Length << 1);

        readers[numReaders++] = me; // install myself as a reader
        lockState = (int)lockStates.Free;
    }
    #endregion
    #region OpenWriteArrayIntegrated
    public void OpenWriteArrayIntegrated<T>(XState me, int index,
                                            T newValue) {
        IAtomicArray<T> castedAtomicArray =
```

```
        (IAtomicArray<T>)atomicObj;
    if(me == null) { // not in a transaction
        if((XStates)writer.state == XStates.ABORTED) {
                // but maybe the last writer aborted
            atomicObj.Restore();
            writer = XAction.ClosedTx;
        }
        castedAtomicArray[index] = newValue;
        return;
    }

    if(me.root == writer.root) {
                // do i already own the object?
        if(Interlocked.Increment(ref me.abortLock) < 1) {
            // inc tx-wide abortLock
            // oh-oh, i've already been aborted
            throw XAction.AbortedException;
        }
        try {
            castedAtomicArray[index] = newValue;
            // do the write - can't be aborted during this time
        }
        finally {
            Interlocked.Decrement(ref me.abortLock);
            // release the pro-life lock
        }
        return;
    } // post-state: i'm not related to the writer

Loop:
    if((XStates)me.state == XStates.ABORTED)
        throw XAction.AbortedException;

    // try to get the lock
    if(Interlocked.CompareExchange(ref lockState,
       (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
       (int)lockStates.Free) {
        Thread.Sleep(0); // necessary to avoid livelock
        goto Loop;
    }

    // am i in conflict with any other readers?
    XState myRoot = me.root;
    for(int i = 0; i < numReaders; i++) {
        XState currReader = readers[i];
        if((XStates)currReader.state == XStates.ACTIVE &&
               myRoot != currReader.root) {
            lockState = (int)lockStates.Free;
                   // release the lock
            XAction.Manager.ResolveConflict(me, currReader);
            goto Loop;
        }
    }
    numReaders = 0; // reset numReaders - no active readers

    switch((XStates)writer.state) {
        case XStates.ACTIVE:
```

```
                lockState = (int)lockStates.Free;
                        // release the lock
                XAction.Manager.ResolveConflict(me, writer);
                goto Loop;
            case XStates.COMMITTED:
                atomicObj.Backup();
                break;
            case XStates.ABORTED:
                atomicObj.Restore();
                break;
        }
        writer = me; // install myself as the writer

        try {
            castedAtomicArray[index] = newValue;
        }
        finally {
            lockState = (int)lockStates.Free;
        }
    }
    #endregion
    #region OpenWriteForReadArrayIntegrated // unused
    public T OpenWriteForReadArrayIntegrated<T>(XState me,
                            int index) {
        IAtomicArray<T> castedAtomicArray =
                (IAtomicArray<T>)atomicObj;
        if(me == null) { // not in a transaction
            if((XStates)writer.state == XStates.ABORTED) {
                    // but maybe the last writer aborted
                atomicObj.Restore();
                writer = XAction.ClosedTx;
            }
            return castedAtomicArray[index];
        }

        T value;
        if(me.root == writer.root) {
            // don't need try-finally here because there is no lock
            // to release if an exception is thrown
            value = castedAtomicArray[index]; // do the read
            if((XStates)me.state == XStates.ABORTED)
                // check to make sure that "value" has a good value
                throw XAction.AbortedException;
            return value;
        } // post-state: i'm not related to the writer

Loop:
        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        // try to get the lock
        if(Interlocked.CompareExchange(ref lockState,
           (int)lockStates.WriteExclusive, (int)lockStates.Free) !=
           (int)lockStates.Free) {
            Thread.Sleep(0); // necessary to avoid livelock
            goto Loop;
        }
```

```
            // am i in conflict with any other readers?
            XState myRoot = me.root;
            for(int i = 0; i < numReaders; i++) {
                XState currReader = readers[i];
                if((XStates)currReader.state == XStates.ACTIVE &&
                     myRoot != currReader.root) {
                    lockState = (int)lockStates.Free;
                           // release the write lock
                    XAction.Manager.ResolveConflict(me, currReader);
                    goto Loop;
                }
            }
            numReaders = 0; // reset numReaders - no active readers

            switch((XStates)writer.state) {
                case XStates.ACTIVE:
                    lockState = (int)lockStates.Free;
                           // release the lock
                    XAction.Manager.ResolveConflict(me, writer);
                    goto Loop;
                case XStates.COMMITTED:
                    atomicObj.Backup();
                    break;
                case XStates.ABORTED:
                    atomicObj.Restore();
                    break;
            }
            writer = me; // install myself as the writer

            try {
                value = castedAtomicArray[index]; // do the read
            }
            finally {
                lockState = (int)lockStates.Free;
            }
            return value;
        }
        #endregion
    }
```

**The VisibleReaders algorithm described in § 5.2**

```
// A transactional object encapsulates an ICloneable object
public struct OFreeSynchState {
    private OFreeLocator rootLocator;

    public OFreeSynchState(ICloneable obj) {
        rootLocator = new OFreeLocator();
        rootLocator.newObject = obj;
        rootLocator.writer = XAction.ClosedTx;
        rootLocator.reader = XAction.ClosedTx;
    }

    // open object with intention to read
    // returns the shared version of object
    public ICloneable OpenRead(XState me) {
        OFreeLocator oldLocator = rootLocator;

        if(me == null) // not in a transaction, update in place
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                case XStates.ACTIVE:
                    throw new PanicException("Tx/not-tx conflict");
            }

        // are we the writer?
        if(me.root == oldLocator.writer.root)
            return oldLocator.newObject;

        // check whether we're already a reader
        OFreeLocator currentLocator = oldLocator;
        do {
            if(currentLocator.reader.root == me.root)
                return oldLocator.newObject;

            // prune dead transactions from the list
            OFreeLocator nextLocator = currentLocator.next;
            while(nextLocator != null &&
               (XStates)nextLocator.reader.state != XStates.ACTIVE)
                nextLocator = nextLocator.next;

            currentLocator.next = currentLocator = nextLocator;
        } while(currentLocator != null);

        OFreeLocator newLocator = new OFreeLocator();
             // allocate successor
        newLocator.writer = XAction.ClosedTx;
        newLocator.reader = me;

        do {
            if((XStates)me.state == XStates.ABORTED)
                throw XAction.AbortedException;
```

```
            // me.state == XStates.ACTIVE (probably)

            // check writer status
            ICloneable currentVersion = null;
            switch((XStates)oldLocator.writer.state) {
                case XStates.ACTIVE:
                    XAction.Manager.ResolveConflict(me,
                        oldLocator.writer); // abort or wait?
                    continue; // try again
                case XStates.COMMITTED:
                    currentVersion = oldLocator.newObject;
                    break;
                case XStates.ABORTED:
                    currentVersion = oldLocator.oldObject;
                    break;
            }

            newLocator.newObject = currentVersion;
            newLocator.next = oldLocator;

            if(Interlocked.CompareExchange(ref rootLocator,
                  newLocator, oldLocator) == oldLocator)
                return currentVersion;

            oldLocator = rootLocator;
        } while(true);
    }

    // open object with intention to modify
    // returns the private version of object
    public ICloneable OpenWrite(XState me) {
        OFreeLocator oldLocator = rootLocator;

        // not in a transaction, update in place
        if(me == null)
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                case XStates.ACTIVE:
                    throw new PanicException("Tx/not-tx conflict");
            }

        if(me.root == oldLocator.writer.root)
            return oldLocator.newObject;

        // allocate successor
        OFreeLocator newLocator = new OFreeLocator();
        newLocator.writer = me;
        newLocator.reader = XAction.ClosedTx;
        ICloneable oldVersion = null, newVersion;

    Loop:
        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;
```

```
    // me.state == XStates.ACTIVE (probably)

    // check for read conflicts
    XState myRoot = me.root;
    for(OFreeLocator currentLocator = oldLocator;
            currentLocator != null;
            currentLocator = currentLocator.next)
        if((XStates)currentLocator.reader.state ==
             XStates.ACTIVE &&
             myRoot != currentLocator.reader.root) {
            XAction.Manager.ResolveConflict(me,
                                     currentLocator.reader);
            goto Loop;
        }

    // check writer status
    switch((XStates)oldLocator.writer.state) {
        case XStates.ACTIVE:
            XAction.Manager.ResolveConflict(me,
                    oldLocator.writer); // abort or wait?
            goto Loop; // try again
        case XStates.COMMITTED:
            oldVersion = newLocator.oldObject =
                    oldLocator.newObject;
            break;
        case XStates.ABORTED:
            oldVersion = newLocator.oldObject =
                    oldLocator.oldObject;
            break;
    }

    // no conflict
    newVersion = newLocator.newObject =
                    (ICloneable)oldVersion.Clone();

    // try to install
    if(Interlocked.CompareExchange(ref rootLocator, newLocator,
                                oldLocator) == oldLocator)
        return newVersion;

    // conflict - try again...
    oldLocator = rootLocator;
    goto Loop;
}

// wrapper routines for array that mimic TMemSynchState
public T OpenReadArrayIntegrated<T>(XState me, int index) {
    return ((IAtomicArray<T>)OpenRead(me))[index];
}

public T OpenWriteForReadArrayIntegrated<T>(XState me,
                                              int index) {
    return ((IAtomicArray<T>)OpenWrite(me))[index];
}

public void OpenWriteArrayIntegrated<T>(XState me, int index,
                                        T newValue) {
```

```
        ((IAtomicArray<T>)OpenWrite(me))[index] = newValue;
    }
}

// keeps track of old and new object versions
// along with latest accessing transaction(s)
internal sealed class OFreeLocator {
    internal XState writer;
            // transaction that wrote this version, or null
    internal XState reader;
            // transaction that read this version, or null
    internal OFreeLocator next;       // previous reader, if any
    internal ICloneable oldObject;    // object version on abort
    internal ICloneable newObject;    // object version on commit

    // display for debugging
    public override string ToString() {
        return String.Format("Locator[writer: {0}, reader: {1}]",
            writer == null ? "null" : writer.ToString(),
            reader == null ? "null" : reader.ToString());
    }
}
```

**The WarningWord algorithm described in § 5.3**

```
// OFreeSynchState that uses a warning word instead
// of a linked list of readers
public struct OFreeSynchState {
    private OFreeLocator rootLocator;

    public OFreeSynchState(ICloneable obj) {
        rootLocator = new OFreeLocator();
        rootLocator.newObject = obj;
        rootLocator.writer = XAction.ClosedTx;
    }

    // open object with intention to read
    //  returns the shared version of object
    public ICloneable OpenRead(XState me) {
        OFreeLocator oldLocator = rootLocator;

        if(me == null) // not in a transaction, update in place
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                default: //XStates.ACTIVE:
                    throw new PanicException("Tx/not-tx conflict");
            }

        if(me.root == oldLocator.writer.root)
                    // yes, am i the writer?
            return oldLocator.newObject;

        ICloneable oldVersion = null;

    TryAgain:

        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        switch((XStates)oldLocator.writer.state) {
            case XStates.COMMITTED:
                oldVersion = oldLocator.newObject;
                break;
            case XStates.ABORTED:
                oldVersion = oldLocator.oldObject;
                break;
            default: //XStates.ACTIVE:
                XAction.Manager.ResolveConflict(me,
                     oldLocator.writer); // abort or wait?
                oldLocator = rootLocator;
                goto TryAgain;
        }

        // have we been warned?
        if((XAction.WarningWord & (1 << me.threadId)) != 0)
            throw XAction.AbortedException;
```

```
        return oldVersion;
    }

    // open object with intention to modify
    // returns the private version of object
    public ICloneable OpenWrite(XState me) {
        OFreeLocator oldLocator = rootLocator;

        // not in a transaction, update in place
        if(me == null)
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                default: //XStates.ACTIVE:
                    throw new PanicException("Tx/not-tx conflict");
            }

        // check whether we're already the writer
        if(me.root == oldLocator.writer.root)
            return oldLocator.newObject;

        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        // allocate successor
        OFreeLocator newLocator = new OFreeLocator();
        newLocator.writer = me;
        ICloneable newVersion;

        do {
            // me.state == XStates.ACTIVE (probably)
            ICloneable oldVersion = null;

            // check writer status
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    oldVersion = oldLocator.newObject;
                    break;
                case XStates.ABORTED:
                    oldVersion = oldLocator.oldObject;
                    break;
                default: //XStates.ACTIVE:
                    XAction.Manager.ResolveConflict(me,
                         oldLocator.writer); // abort or wait?
                    continue; // try again
            }

            // no conflict
            newLocator.oldObject = oldVersion;
            newVersion = (ICloneable)oldVersion.Clone();
            newLocator.newObject = newVersion;

            // try to install
            if(Interlocked.CompareExchange(ref rootLocator,
```

```
                    newLocator, oldLocator) == oldLocator)
                break;

            // conflict - try again...
            oldLocator = rootLocator;
        } while(true); // keep trying

        // set the thread warning bit to warn all other threads
        int conflict = ~(1 << me.threadId);
        int oldValue, newValue;
        do {
            oldValue = XAction.WarningWord;
            newValue = oldValue | conflict;
            if(newValue == oldValue)
                break;
        } while(Interlocked.CompareExchange(
         ref XAction.WarningWord, newValue, oldValue) != oldValue);

        if((XAction.WarningWord & (1 << me.threadId)) != 0)
            throw XAction.AbortedException;

        return newVersion;
    }

    // wrapper routines for array that mimic TMemSynchState
    public T OpenReadArrayIntegrated<T>(XState me, int index) {
        return ((IAtomicArray<T>)OpenRead(me))[index];
    }

    public T OpenWriteForReadArrayIntegrated<T>(XState me,
                                                  int index) {
        return ((IAtomicArray<T>)OpenWrite(me))[index];
    }

    public void OpenWriteArrayIntegrated<T>(XState me, int index,
                                              T newValue) {
        ((IAtomicArray<T>)OpenWrite(me))[index] = newValue;
    }
}

// keeps track of old and new object versions
// along with latest accessing transaction(s)
internal sealed class OFreeLocator {
    internal XState writer;
         // transaction that wrote this version, or null
    internal ICloneable newObject;    // object version on commit
    internal ICloneable oldObject;    // object version on abort

    // display for debugging
    public override string ToString() {
        return String.Format(
            "Locator[writer: {0}, new: {1}, old: {2}]",
            writer.ToString(),
            newObject.GetHashCode(), oldObject.GetHashCode());
    }
}
```

**The BloomFilter algorithm described in § 5.4**

```
// OFreeSynchState that uses a Bloom filter
public struct OFreeSynchState {
    private OFreeLocator rootLocator;
    private readonly int hash;

    // the global bloom filter
    internal static BloomFitler TheBFilter = new BloomFitler();

    public OFreeSynchState(ICloneable obj) {
        rootLocator = new OFreeLocator();
        rootLocator.newObject = obj;
        hash = obj.GetHashCode() & BloomFitler.SIZE - 1;
        rootLocator.writer = XAction.ClosedTx;
    }

    // open object with intention to read
    // returns the shared version of object
    public ICloneable OpenRead(XState me) {
        OFreeLocator oldLocator = rootLocator;
        ICloneable oldVersion = null;

        if(me == null) // not in a transaction, update in place
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                case XStates.ACTIVE:
                    throw new PanicException("Tx/not-tx conflict");
            }

        if(me.root == oldLocator.writer.root)
                 // yes, am i the writer?
            return oldLocator.newObject;

        // add it to the tx's readlog
        me.txReadList.Add(hash);

        // add it to the global bloom filter
        TheBFilter.Add(hash, me.threadId);

        do {
            if((XStates)me.state == XStates.ABORTED)
                throw XAction.AbortedException;
            // me.state == XStates.ACTIVE (probably)

            switch((XStates)oldLocator.writer.state) {
                case XStates.ACTIVE:
                    XAction.Manager.ResolveConflict(me,
                         oldLocator.writer); // abort or wait?
                    oldLocator = rootLocator;
                    continue; // try again
                case XStates.COMMITTED:
                    oldVersion = oldLocator.newObject;
```

```
                    break;
                case XStates.ABORTED:
                    oldVersion = oldLocator.oldObject;
                    break;
            }

            if(oldLocator == rootLocator)
                break;

            // conflict - try again...
            oldLocator = rootLocator;
        } while(true);

        if(TheBFilter.CheckWarningWord(me.threadId)) {
            TheBFilter.Remove(me);
            throw XAction.AbortedException;
        }

        return oldVersion;
    }

    // open object with intention to modify
    // returns the private version of object
    public ICloneable OpenWrite(XState me) {
        OFreeLocator oldLocator = rootLocator;

        // not in a transaction, update in place
        if(me == null)
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                case XStates.ACTIVE:
                    throw new PanicException("Tx/not-tx conflict");
            }

        // check whether we're already the writer
        if(me.root == oldLocator.writer.root)
            return oldLocator.newObject;

        // allocate successor
        OFreeLocator newLocator = new OFreeLocator();
        newLocator.writer = me;
        ICloneable newVersion, oldVersion = null;

        do {
            if((XStates)me.state == XStates.ABORTED)
                throw XAction.AbortedException;
            // me.state == XStates.ACTIVE (probably)

            // check writer status
            switch((XStates)oldLocator.writer.state) {
                case XStates.ACTIVE:
                    XAction.Manager.ResolveConflict(me,
                         oldLocator.writer); // abort or wait?
                    oldLocator = rootLocator;
```

```
                    continue; // try again
                case XStates.COMMITTED:
                    oldVersion = oldLocator.newObject;
                    break;
                case XStates.ABORTED:
                    oldVersion = oldLocator.oldObject;
                    break;
            }

            // no conflict
            newLocator.oldObject = oldVersion;
            newVersion = (ICloneable)oldVersion.Clone();
            newLocator.newObject = newVersion;

            // try to install
            if(Interlocked.CompareExchange(ref rootLocator,
                    newLocator, oldLocator) == oldLocator)
                break;

            // conflict - try again...
            oldLocator = rootLocator;
        } while(true);

        // bfilter stuff
        int conflict = TheBFilter.filter[hash] &
                                       ~(1 << me.threadId);
        if(conflict != 0)
            TheBFilter.SetWarningWord(conflict);

        if(TheBFilter.CheckWarningWord(me.threadId)) {
            TheBFilter.Remove(me);
            throw XAction.AbortedException;
        }

        return newVersion;
    }

    // wrapper routines for arrays that mimic TMemSynchState
    public T OpenReadArrayIntegrated<T>(XState me, int index) {
        return ((IAtomicArray<T>)OpenRead(me))[index];
    }

    public T OpenWriteForReadArrayIntegrated<T>(XState me,
                                                int index) {
        return ((IAtomicArray<T>)OpenWrite(me))[index];
    }

    public void OpenWriteArrayIntegrated<T>(XState me, int index,
                                            T newValue) {
        ((IAtomicArray<T>)OpenWrite(me))[index] = newValue;
    }
}

// keeps track of old and new object versions
// along with latest accessing transaction(s)
internal sealed class OFreeLocator {
    internal XState writer;
```

```
            // transaction that wrote this version, or null
        internal ICloneable oldObject;    // object version on abort
        internal ICloneable newObject;    // object version on commit

        // display for debugging
        public override string ToString() {
            return String.Format("Locator[writer: {0}, reader: {1}]",
                writer == null ? "null" : writer.ToString());
        }
    }

    // Bloom filter implementation of read set.
    internal class BloomFitler {
        internal const int SIZE = 0x10000; // 65536
        internal readonly int[] filter = new int[SIZE];
                                        // 262,144 bytes
        private int WarningWord = 0;

        // Add an object to the filter.
        public void Add(int hash, int threadId) {
            int oldValue, newValue;
            do {
                oldValue = filter[hash];
                newValue = oldValue | (1 << threadId);
                if(oldValue == newValue)
                    break;
            } while(Interlocked.CompareExchange(ref filter[hash],
                    newValue, oldValue) != oldValue);
        }

        // Remove an object from the filter.
        public void Remove(XState me) {
            int oldValue, newValue;
            int size = me.txReadList.Count;

            for(int i = 0; i < size; i++) {
                int h = me.txReadList[i];
                do {
                    oldValue = filter[h];
                    newValue = oldValue & ~(1 << me.threadId);
                    if(oldValue == newValue)
                        break;
                } while(Interlocked.CompareExchange(ref filter[h],
                        newValue, oldValue) != oldValue);
            }

            ClearWarningWord(me.threadId);

            // empty the list after removal
            me.txReadList.Clear();
        }

        // check whether reader has been warned of read/write conflict
        public bool CheckWarningWord(int threadId) {
            return (WarningWord & (1 << threadId)) != 0;
        }
```

```
    // Warn another thread that you are modifying object it
    // may have placed in filter.
    public void SetWarningWord(int mask) {
        int oldValue, newValue;
        do {
            oldValue = WarningWord;
            newValue = oldValue | mask;
            if(newValue == oldValue)
                break;
        } while(Interlocked.CompareExchange(ref WarningWord,
                newValue, oldValue) != oldValue);
    }

    public void ClearWarningWord(int threadId) {
        int oldValue, newValue;
        int mask = ~(1 << threadId);
        do {
            oldValue = WarningWord;
            newValue = oldValue & mask;
            if(newValue == oldValue)
                break;
        } while(Interlocked.CompareExchange(ref WarningWord,
                newValue, oldValue) != oldValue);
    }
}
```

### The WriterBins algorithm described in § 5.5

```
// OFreeSynchState that uses WriterBins
public struct OFreeSynchState {
    private OFreeLocator rootLocator;
    private readonly int bin;

    public OFreeSynchState(ICloneable obj) {
        rootLocator = new OFreeLocator();
        rootLocator.newObject = obj;
        rootLocator.writer = XAction.ClosedTx;
               // initialize the writer to ClosedTx
        bin = obj.GetHashCode() &
              XAction.GlobalNumberOfWriterBins - 1;
                    // what bin does this object belong to?
    }

    // open object with intention to read
    // returns the shared version of object
    public ICloneable OpenRead(XState me) {
        OFreeLocator oldLocator = rootLocator;

        if(me == null) // not in a transaction, update in place
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                default: //XStates.ACTIVE
                    throw new PanicException("Tx/not-tx conflict");
            }

        if(me.root == oldLocator.writer.root)
                  // yes, am i the writer?
            return oldLocator.newObject;

        ICloneable oldVersion;

    TryAgain:

        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;

        // check writer status
        switch((XStates)oldLocator.writer.state) {
            case XStates.COMMITTED:
                oldVersion = oldLocator.newObject;
                break;
            case XStates.ABORTED:
                oldVersion = oldLocator.oldObject;
                break;
            default: //XStates.ACTIVE:
                XAction.Manager.ResolveConflict(me,
                        oldLocator.writer); // abort or wait?
                oldLocator = rootLocator;
                goto TryAgain;
```

```
        }

        if((me.txReadBins & 1 << bin) == 0) {
            // is this the first time we're reading this bitCode?
            // set the bit in txReadFilter
            // this is required for commit-time validation
            me.txReadBins |= 1 << bin;

            // check that someone else hasn't committed a new value
            // here since this tx began. this check isn't required
            // it's an early conflict detector. without this, more
            // txs will abort at the commit stage. could also check
            // this on every read (not just the first read for this
            // bin), but that seems a little more expensive than
            // it's worth - tests inconclusive...
            if(XAction.GlobalBinVerCounters[bin] !=
                          me.txBinVerCounters[bin])
                throw XAction.AbortedException;

            // This prevents livelock...
            XState oldBinOccupier = XAction.GlobalWriterBins[bin];
            if((XStates)oldBinOccupier.state == XStates.ACTIVE) {
                XAction.Manager.ResolveConflict(me,
                                                 oldBinOccupier);
                goto TryAgain;
            }
        }

        return oldVersion;
    }

    // open object with intention to modify
    // returns the private version of object
    public ICloneable OpenWrite(XState me) {
        OFreeLocator oldLocator = rootLocator;

        // not in a transaction, update in place
        if(me == null)
            switch((XStates)oldLocator.writer.state) {
                case XStates.COMMITTED:
                    return oldLocator.newObject;
                case XStates.ABORTED:
                    return oldLocator.oldObject;
                default: //XStates.ACTIVE:
                    throw new PanicException("Tx/not-tx conflict");
            }

        // check whether we're already the writer
        if(me.root == oldLocator.writer.root)
            return oldLocator.newObject;

        // mandatory
        if((XStates)me.state == XStates.ABORTED)
            throw XAction.AbortedException;
        // me.state == XStates.ACTIVE (probably)

        // is this the first time we're writing this bitCode
```

```
if((me.txWriteBins & 1 << bin) == 0) {
    XState oldBinOccupier;
    do {
        oldBinOccupier = XAction.GlobalWriterBins[bin];

        // check that someone else hasn't committed a new
        // value here since this tx began. mandatory,
        // otherwise you might make yourself the writer of
        // this binNumber with an inconstitent read/write
        // problem
        if(XAction.GlobalBinVerCounters[bin] !=
                    me.txBinVerCounters[bin])
            throw XAction.AbortedException;

        // if the writer for this spot is active,
        // then call the contention manager
        if((XStates)oldBinOccupier.state == XStates.ACTIVE)
{
            XAction.Manager.ResolveConflict(me,
                     oldBinOccupier); // abort or wait?

            // have we been aborted now?
            if((XStates)me.state == XStates.ABORTED)
                throw XAction.AbortedException;

            continue;
        }
    } while(Interlocked.CompareExchange(
            ref XAction.GlobalWriterBins[bin], me,
            oldBinOccupier) != oldBinOccupier);

    // set the txWriteFilter bit to indicate that we
    // own this entry
    me.txWriteBins |= 1 << bin;
}

// ASSERT: (XStates)oldLocator.writer.state !=
//         XStates.ACTIVE

OFreeLocator newLocator = new OFreeLocator();
    // allocate the new locator
newLocator.writer = me; // make me the writer
// check previous writer status -- cannot be active;
// ConMan will have killed it above...
newLocator.oldObject = (XStates)oldLocator.writer.state ==
    XStates.COMMITTED ?
    oldLocator.newObject : // COMMITTED
    oldLocator.oldObject;  // ABORTED
newLocator.newObject =
       (ICloneable)newLocator.oldObject.Clone();
       // create a clone

// try to install
if(Interlocked.CompareExchange(ref rootLocator, newLocator,
            oldLocator) != oldLocator)
    throw XAction.AbortedException; // failed
```

```
            // no conflict
            return newLocator.newObject;
        }

        // wrapper routines for array that mimic TMemSynchState
        public T OpenReadArrayIntegrated<T>(XState me, int index) {
            return ((IAtomicArray<T>)OpenRead(me))[index];
        }

        public T OpenWriteForReadArrayIntegrated<T>(XState me,
                                                    int index) {
            return ((IAtomicArray<T>)OpenWrite(me))[index];
        }

        public void OpenWriteArrayIntegrated<T>(XState me, int index,
                                                T newValue) {
            ((IAtomicArray<T>)OpenWrite(me))[index] = newValue;
        }
    }

    // keeps track of old and new object versions
    // along with last writing transaction
    internal sealed class OFreeLocator {
        internal XState writer;
                // last transaction that wrote to this object
        internal ICloneable newObject;    // object version on commit
        internal ICloneable oldObject;    // object version on abort

        // display for debugging
        public override string ToString() {
            return String.Format(
               "Locator[writer: {0}, new: {1}, old: {2}]",
                writer.ToString(), newObject.GetHashCode(),
                oldObject.GetHashCode());
        }
    }
```