Technical Report

UCAM-CL-TR-579
ISSN 1476-2986

Number 579

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Practical lock-freedom

Keir Fraser

February 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2003 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Summary

Mutual-exclusion locks are currently the most popular mechanism for interprocess synchronisation, largely due to their apparent simplicity and ease of implementation. In the parallel-computing environments that are increasingly commonplace in high-performance applications, this simplicity is deceptive: mutual exclusion does not scale well with large numbers of locks and many concurrent threads of execution. Highly-concurrent access to shared data demands a sophisticated 'fine-grained' locking strategy to avoid serialising non-conflicting operations. Such strategies are hard to design correctly and with good performance because they can harbour problems such as deadlock, priority inversion and convoying. Lock manipulations may also degrade the performance of cache-coherent multiprocessor systems by causing coherency conflicts and increased interconnect traffic, even when the lock protects read-only data.

In looking for solutions to these problems, interest has developed in *lock-free* data structures. By eschewing mutual exclusion it is hoped that more efficient and robust systems can be built. Unfortunately the current reality is that most lock-free algorithms are complex, slow and impractical. In this dissertation I address these concerns by introducing and evaluating practical abstractions and data structures that facilitate the development of large-scale lock-free systems.

Firstly, I present an implementation of two useful abstractions that make it easier to develop arbitrary lock-free data structures. Although these abstractions have been described in previous work, my designs are the first that can be practically implemented on current multiprocessor systems.

Secondly, I present a suite of novel lock-free search structures. This is interesting not only because of the fundamental importance of searching in computer science and its wide use in real systems, but also because it demonstrates the implementation issues that arise when using the practical abstractions I have developed.

Finally, I evaluate each of my designs and compare them with existing lock-based and lock-free alternatives. To ensure the strongest possible competition, several of the lock-based alternatives are significant improvements on the best-known solutions in the literature. These results demonstrate that it is possible to build useful data structures with all the perceived benefits of lock-freedom and with performance better than sophisticated lock-based designs. Furthermore, and contrary to popular belief, this work shows that existing hardware primitives are sufficient to build practical lock-free implementations of complex data structures.

Table of contents

Chapter 1

Introduction

This dissertation is concerned with the design and implementation of practical lock-free data structures. By providing effective abstractions built from readily-available hardware primitives, I show that a range of useful data structures can be built. My results demonstrate that existing hardware primitives are sufficient to implement efficient data structures that compete with, and often surpass, state-of-the-art lock-based designs.

In this chapter I outline the background issues that motivated this work, and state the contributions that are described in this dissertation. I then summarise the contents of each chapter and describe the language conventions used in the pseudocode examples throughout this document.

1.1 Motivation

Mutual-exclusion locks are one of the most widely used and fundamental abstractions for interprocess synchronisation. This popularity is largely due to their apparently simple programming model and their efficient implementation on shared-memory systems. Unfortunately these virtues frequently do not scale to systems containing more than a handful of locks, which may suffer a range of problems:

- Care must be taken to avoid *deadlock*. To do so, locks are usually taken in some global order, but this can affect the efficiency of some algorithms. For example, a lock may be held for longer than would otherwise be necessary, or a write lock may be taken even though updates are rarely required.
- Unfortunate scheduler interactions can cause critical operations to be delayed. A classic example is *priority inversion*, in which a process is pre-

empted while holding a lock which a higher-priority process requires to make progress.

- Even when a data structure is accessed through a sequence of 'fine-grained' locks, processes will tend to form *convoys* as they queue at a sequence of locks. This behaviour is exacerbated by the increased queuing time at a lock when it is attended by a convoy.

Furthermore, system designers increasingly believe that *parallelism* is necessary to satisfy the demands of high-performance applications. For example, high-end servers often consist of a physically-distributed set of processing and memory nodes which communicate via a cache-coherent interconnect; in some cases this architecture is applied to a conventionally-networked cluster of computers. Meanwhile, on the desktop, simultaneous multithreading (SMT) is being introduced, which allows a single processor core to harness parallelism among a set of threads [Tullsen95].

Unfortunately, locks can compromise the reliability and performance of highly-parallel systems in a number of ways:

- Mutual exclusion can needlessly restrict parallelism by serialising non-conflicting updates. This can be greatly mitigated by using fine-grained locks, but lock convoying and cache performance may then become an issue, along with the extra cost of acquiring and releasing these locks.

- Even when an operation does not modify shared data, the required lock manipulations can cause memory coherency conflicts, and contribute to contention on the memory interconnect. This can have enormous impact on system performance: Larson and Krishnan [Larson98] observe that "reducing the frequency of access to shared, fast-changing data items" is critical to prevent *cache bouncing*[1] from limiting system throughput.

- In a loosely-coupled system, such as a cluster, deadlock can occur if a processing node fails or stalls while holding a lock. The problem of ensuring system-wide progress in these situations has led to work on *leased locks*, which preempt ownership when a lock is held for too long.

There is a growing interest in *lock-free* data structures as a way of sidestepping these problems. By eschewing mutual exclusion it is hoped that more efficient and robust systems can be built. Unfortunately the current reality is that most lock-free algorithms are complex, slow and impractical. This dissertation ad-

[1] *Cache bouncing*, or *cache ping-pong*, occurs when exclusive ownership of a cache line moves rapidly among a set of processors.

dresses this situation by presenting and evaluating practical abstractions and data structures that facilitate the development of large-scale lock-free systems.

1.2 Terminology discussion

This dissertation makes frequent use of a number of technical terms which I define and discuss here for clarity and simplicity of reference. Many of these are discussed in greater depth in the next chapter — this section is intended as a glossary.

I use the term **lock-free** to describe a system which is guaranteed to make forward progress within a finite number of execution steps. Rather confusingly, a program which uses no mutual-exclusion locks is not *necessarily* lock-free by this definition: in fact the term applies to any system of processes which is guaranteed never to experience global deadlock or livelock, irrespective of the progress of individual processes. Note that systems satisfying this property are sometimes referred to as **non-blocking**. Following more recent usage, I instead reserve this term for the more general property that a stalled process cannot cause all other processes to stall indefinitely [Herlihy03a].

Since locks are disallowed in lock-free algorithms I instead use the **compare-&-swap (CAS)** primitive to execute atomic read-modify-write operations on shared memory locations. This primitive takes three arguments: a memory location, a value that is expected to be read from that location, and a new value to write to the location if the expected value is found there. The most common use of CAS is to read from a memory location, perform some computation on the value, and then use CAS to write the modified value while ensuring that the location has not meanwhile been altered. CAS is supported in hardware by most modern multiprocessor architectures. Those that do not implement CAS provide alternative machine instructions that can be used to emulate CAS with very little overhead.

Another commonly-assumed primitive is **double-word compare-&-swap (DCAS)** which effectively executes two CAS operations simultaneously: both memory locations are atomically updated if and only if both contain the expected initial values. It is usually easier to build lock-free algorithms using DCAS because it simplifies 'tying together' updates to multiple memory locations. However, DCAS is not supported in hardware by any modern processor architecture.

The most common technique for handling update conflicts in lock-free algorithms is **recursive helping**. If the progress of an operation A is obstructed by a conflicting operation B, then A will help B to complete its work. This is usually implemented by recursively reentering the operation, passing the invocation parameters specified by B. Operation B is responsible for making available sufficient information to allow conflicting processes to determine its invocation parameters. When the recursive call is completed, the obstruction will have been removed and operation A can continue to make progress.

When discussing lock-based algorithms I make use of two common locking protocols. The first is the conventional **mutual-exclusion lock** which supports only the operations *acquire* and *release*. These operations are used in pairs to protect a critical region: only one process at a time may acquire the lock and thus enter its critical region. There are many situations in which most operations are read-only, yet these *readers* are forced to acquire locks to gain protection from occasional updates. In these cases, where the strict serialisation enforced by mutual-exclusion locks is overkill, **multi-reader locks** are commonly used. These can be acquired either for reading or for writing: multiple readers may hold the lock simultaneously, but writers must acquire exclusive ownership of the lock. Multi-reader locks are not a panacea, however, since the *acquire* and *release* operations themselves create contention between otherwise read-only operations. As I will show in Chapter 6, this contention can be a major bottleneck in large-scale systems. It is generally preferable to find an algorithm which synchronises readers without resorting to locks of any kind.

1.3 Contribution

It is my thesis that lock-free data structures have important advantages compared with lock-based alternatives, and that tools for applying lock-free techniques should be placed within reach of mainstream programmers. Existing lock-free programming techniques have been too complex, slow, or assume too much of the underlying hardware to be of practical use — in this dissertation I introduce lock-free programming abstractions and real-world data structures which run on a wide range of modern multiprocessor systems and whose performance and simplicity can surpass sophisticated lock-based designs.

My first contribution is the design and implementation of two programming abstractions which greatly reduce the complexity of developing lock-free data structures. Based on the requirements of the data structure, a programmer can choose between *multi-word compare-&-swap* (MCAS) and *software transactional memory* (STM) to enforce serialisation of complex shared-memory operations. Although both these abstractions have been discussed in previous work, I present results which show that my implementations are not only practical to deploy, in contrast with existing lock-free designs, but also competitive with locking strategies based on high-performance mutexes.

A further contribution is a suite of efficient lock-free designs for three search structures: skip lists, binary search trees (BSTs), and red-black trees. These are interesting not only because of the fundamental importance of searching in computer science and its wide use in real systems, but also because they demonstrate the issues involved in implementing non-trivial lock-free data structures. Performance results show that these lock-free search structures compare well with high-performance lock-based designs. To ensure the fairest possible competition I derive a competing lock-based BST design directly from my lock-free algorithm, in which only update operations need to acquire locks. This, together with a novel lock-based design for concurrent red-black trees, represents a further contribution to the state of the art.

By fully implementing each of my designs I am able to evaluate each one on real multiprocessor systems. These results indicate that, contrary to popular belief, existing hardware primitives are sufficient to build practical lock-free implementations of complex data structures. Furthermore, the discussion of this implementation work illustrates the issues involved in turning abstract algorithm designs into deployable library routines.

1.4 Outline

In this section I describe the organisation of the remainder of this dissertation.

In Chapter 2 I describe previous work which relates to and motivates this dissertation. A recurring issue is that, except for a few simple data structures, existing lock-free algorithms are complex and slow.

In Chapter 3 I motivate and present efficient lock-free designs for two easy-to-use programming abstractions: multi-word compare-&-swap, and software transactional memory.

In Chapter 4 I introduce lock-free designs for three well-known dynamic search structures. Not only are these structures interesting in their own right, but they also illustrate non-trivial uses of the abstractions presented in Chapter 3.

In Chapter 5 I discuss the implementation issues that are faced when turning the pseudocode designs from Chapters 3 and 4 into practical implementations for real hardware.

In Chapter 6 I explain how I tested my lock-free implementations for correctness. I then present experimental results that demonstrate the practicality of my new lock-free designs compared with competitive lock-based and non-blocking alternatives.

Finally, in Chapter 7 I conclude the dissertation and suggest areas for further research.

1.5 Pseudocode conventions

All the pseudocode fragments in this dissertation are written using a C-style programming language. C is a simple and transparent language, which prevents important design details from being hidden behind complex language-level constructs. However, for clarity I assume a *sequentially-consistent* memory model, and I introduce the following new primitives:

- A datatype bool, taking values TRUE and FALSE.
- An integer datatype word, representing a word of memory in the native machine architecture.
- A function CAS(word *address, word expected, word new) with the same semantics as the well-known hardware primitive. It returns the previous contents of address; if this differs from expected then the operation failed.
- A Java-style new operator which allocates a new instance of the specified datatype. Unless otherwise specified, I assume that memory is automatically garbage-collected.
- Tuples, first-class datatypes used to concisely represent non-scalar values. They are declared by (type$_1$, ..., type$_n$), denoted in use by (x$_1$, ..., x$_n$), and can be freely passed to and from functions.
- An underscore can be used in place of a variable name on the left-hand side of an assignment. This turns the assignment into a null operation, and is particularly useful for discarding unwanted components of a tuple.

Furthermore, I write some standard C operators using a clearer representation:

Operator class	**C representation**	**Pseudocode representation**
Assignment	$=$	$:=$
Equality	$==, !=$	$=, \neq$
Relational	$<, >, <=, >=$	$<, >, \leq, \geq$
Logical	$\|\|, \&\&, !$	$\vee, \wedge, \neg$
Point-to-member	$->$	$\rightarrow$

Chapter 2

Background

I begin this chapter by presenting in greater detail the technical terms that are generally used when discussing non-blocking systems. Some of these terms describe rather abstract performance guarantees: I therefore introduce some additional formal properties that are desirable in high-performance systems, and that are satisfied by all the lock-free algorithms in Chapters 3 and 4. In the remainder of the chapter I summarise previous work relating to non-blocking systems and discuss the limitations which place existing lock-free programming techniques beyond practical use.

2.1 Terminology

In this dissertation I consider a shared data structure to be a set of memory locations that are shared between multiple *processes* and are accessed and updated only by a supporting set of *operations*. Mutual-exclusion locks are commonly used to ensure that operations appear to execute in isolation, act atomically and leave the structure in a consistent state. However, as I described in Chapter 1, locks have many drawbacks including a very weak progress guarantee: if a process never releases a lock that it has taken then it may be impossible for any concurrent operations to complete as all may stall waiting to take the failed lock.

To solve this problem, several *non-blocking* properties have been proposed which provide stronger progress guarantees by precluding the use of mutual exclusion. All the non-blocking properties described here guarantee that a stalled process cannot cause all other processes to stall indefinitely. The tradeoff which they explore is the range of assurances which may be provided to groups of conflicting non-stalled processes. In general, stronger progress guarantees can be provided at the cost of reduced overall performance.

2.1.1 Lock-freedom

The most popular non-blocking property, and the one that I consider in the remainder of this dissertation, is *lock-freedom*. A data structure is lock-free if and only if *some* operation completes after a finite number of steps system-wide have been executed on the structure. This guarantee of system-wide progress is usually satisfied by requiring a process that experiences contention to *help* the conflicting operation to complete before continuing its own work, thus ensuring that every executing process is always ensuring forward progress of some operation. This is a very different approach to that taken by lock-based algorithms, in which a process will either spin or block until the contending operation is completed.

Considerable software infrastructure is often required to allow lock-free helping: an incomplete operation must leave enough state in the shared structure to allow a consistent view of the structure to be constructed, and to allow any process to help it to complete; each operation must be carefully designed to ensure that each execution step can update shared memory at most once, no matter how many times it is 'replayed' by different processes; and reclamation of structures in shared memory is complicated by the fact that any process may access any shared location at any time.

2.1.2 Wait-freedom

Although lock-freedom guarantees system-wide progress it does not ensure that individual operations eventually complete since, in theory, an operation may continually be deferred while its process helps a never-ending sequence of contending operations. In some applications a fairer condition such as *wait-freedom* may be desirable. A data structure is wait-free if and only if *every* operation on the structure completes after it has executed a finite number of steps. This condition ensures that no operation can experience permanent livelock and, in principle, a worst-case execution time can be calculated for any operation.

It is very difficult to implement efficient wait-free algorithms on commodity hardware since fair access to memory is usually not guaranteed. Extensive software-based synchronisation is usually required to ensure that no process is starved. This is typically achieved by requiring each process to announce its current operation in a single-writer memory location. Processes which successfully make forward progress are required to periodically scan the announcements of other processes and help their operations to complete. Over time, the scanning algorithm should check every process in the system.

Note that the strict progress guarantees of wait-free algorithms are primarily of interest in hard real-time applications, whose requirements are typically not met by the types of multiprocessor system that I consider in this dissertation. The probabilistic elements found in such systems (e.g., memory caches) cannot provide the determinism required in hard real-time design.

2.1.3 Obstruction-freedom

Herlihy *et al.* have recently suggested a weak non-blocking property called *obstruction-freedom*, which they believe can provide many of the practical benefits of lock-freedom but with reduced programming complexity and the potential for more efficient data-structure designs [Herlihy03a]. Since efficiently allowing operations to help each other to complete is a major source of complexity in many lock-free algorithms, and excessive helping can generate harmful memory contention, obstruction-freedom can reduce overheads by allowing a conflicting operation to instead be *aborted* and retried later.

More formally, a data structure is obstruction-free if and only if every operation on the structure completes after executing a finite number of steps that do not contend with any concurrent operation for access to any memory location. Thus, although obstruction-freedom is strong enough to prevent effects such as deadlock or priority inversion, an out-of-band mechanism is required to deal with livelock (which might be caused by two mutually conflicting operations continually aborting each other). The cost of avoiding livelock in obstruction-free algorithms has not yet been investigated empirically — for example, if exponential backoff is used when retrying a contended operation then it is not certain that there will be a 'sweet spot' for the back-off factor in all applications. Evaluation of different livelock-avoidance mechanisms is the subject of ongoing research.

2.2 Desirable algorithmic features

In addition to the formal non-blocking properties that I describe above, there are other concerns which must be addressed by practical lock-free designs, such as performance and usability. To this end, all the lock-free algorithms that I present in this dissertation are both *disjoint-access parallel* and *linearisable*.

2.2.1 Disjoint-access parallelism

The performance guarantees of the various non-blocking properties are somewhat abstract: for instance, they don't promise that compliant operations will execute efficiently on real hardware. One property which attempts to bridge the gap between formal definition and real performance is *disjoint-access parallelism* [Israeli94]. A set of operations are disjoint-access parallel if and only if any pair of operation invocations which access disjoint sets of memory locations do not directly affect each others' execution.

This prohibits performance bottlenecks such as using 'ownership records' to serialise access to large regions of shared memory. However, it does not prevent an operation from indirectly affecting another's performance (perhaps via cache effects).

2.2.2 Linearisability

Besides performance, another metric by which algorithms can be evaluated is *usability*: in particular, does the algorithm behave as expected when it is deployed in an application? One property which is commonly considered desirable in concurrency-safe algorithms is *linearisability* [Herlihy90c]. This property is defined in terms of requests to and responses from a compliant operation: if the operation is implemented as a synchronous procedure then a call to that procedure is a request and the eventual return from that procedure is a response. An operation is linearisable if and only if it appears to execute instantaneously at some point between its request and response.

Linearisability ensures that operations have intuitively 'correct' behaviour. Concurrent invocations of a set of linearisable operations will have a corresponding sequence which could be executed by just one processor with exactly the same outcome. Another way of looking at this is that linearisable procedures behave as if the data they access is protected by a single mutual-exclusion lock which is taken immediately after a request and released immediately before a response.

2.3 Related work

In the following section I present previous work relating to non-blocking data structures and discuss why none of the proposed lock-free programming techniques are viable general-purpose alternatives to using locks.

Firstly, I introduce the primitives that have previously been used to build non-blocking data structures; unfortunately, a great deal of existing work is based on primitives that are not supported by current hardware. Secondly, I present universal constructions that can render concurrency-safe any suitable-specified sequential implementation of a data structure. Thirdly, I present programming abstractions that make it easier for programmers to directly implement non-blocking data structures. Fourthly, since existing constructions and programming abstractions are impractical for general use I present data structures that have instead been implemented by using hardware primitives directly. Finally, I discuss work in non-blocking memory management — an important yet often-ignored aspect of lock-free design.

2.3.1 Non-blocking primitives

An early paper by Herlihy demonstrates that various classical atomic primitives, including fetch-&-add and test-&-set, have differing levels of expressiveness [Herlihy88]. Specifically, a hierarchy is constructed in which primitives at a given level cannot be used to implement a wait-free version of any primitives at a higher level. Only a few of the well-known primitives discussed in the paper are *universal* in the sense that they can be used to solve the n-process consensus problem [Fischer85] in its general form.

One such universal primitive is compare-&-swap (CAS), which is used to build the lock-free algorithms described in this dissertation. Originally implemented in the IBM System/370 [IBM70], many modern multiprocessors support this operation in hardware.

Rather than implementing a read-modify-write instruction directly, some processors provide separate load-linked and store-conditional (LL/SC) operations. Unlike the *strong* LL/SC operations sometimes used when describing algorithms, the implemented instructions must form non-nesting pairs and SC can fail ‘spuriously’ [Herlihy93a]. Methods for building read-modify-write primitives from LL/SC are well known: for example, the Alpha processor handbook shows how to use them to construct atomic single-word sequences such as CAS [DEC92]. Such constructions, based on a simple loop that retries a LL/SC pair, are non-blocking under a guarantee that there are not infinitely many spurious failures during a single execution of the sequence.

It is widely believed that the design of efficient non-blocking algorithms is much easier if a more expressive operation such as DCAS is supported [Greenwald99, Detlefs00]. Unfortunately only the obsolete Motorola 680x0 family of processors supports DCAS directly in hardware [Motorola, Inc.], although Bershad describes how to implement CAS on architectures with weak atomic primitives using a technique that could easily be extended to DCAS [Bershad93]. This technique involves using a single shared lock which is known to the operating system, so contention will significantly affect performance under any memory-intensive workload.

2.3.2 Universal constructions

Universal constructions are a class of lock-free techniques that can be straightforwardly applied to a wide range of sequential programs to make them safe in parallel-execution environments. Indeed, most of these constructions are intended to be applied automatically by a compiler or run-time system.

Lamport was an early proponent of constructions that permit concurrent reading and writing of an arbitrary-sized data structure without requiring mutual exclusion [Lamport77]. His approach uses a pair of version counters, one of which is incremented before an update, and the other immediately after. Read operations read these variables in reverse order before and after accessing the data structure, and retry if they do not match. This approach is not lock-free, or even non-blocking, in the sense used in this dissertation, since a stalled writer can cause readers to retry their operation indefinitely. Lamport also assumes an out-of-band mechanism for synchronising multiple writers.

Herlihy describes a universal construction for automatically creating a non-blocking algorithm from a sequential specification [Herlihy90b, Herlihy93a]. This requires a snapshot of the entire data object to be copied to a private location where shadow updates can safely be applied: these updates become visible when the single 'root' pointer of the structure is atomically checked and modified to point at the shadow location. Although Herlihy describes how copying costs can be significantly reduced by replacing only those parts of the object that are modified, the construction still requires atomic update of a single root pointer. This means that concurrent updates will always conflict, even when they modify disjoint sections of the data structure.

Alemany and Felten extend Herlihy's work to avoid the useless work done by parallel competing processes accessing the same data structure [Alemany92]. They achieve this by including an in-progress reference count, and causing processes to defer their work if they attempt to start an operation when the count is above some threshold. If a process can make no further progress at this point then it may yield the processor to a process that is currently updating the data structure. If the threshold is set to one then update operations can update the structure in place rather than making a shadow copy; an update log must be maintained, however, so that these updates can be undone if the process is preempted. Unfortunately this approach still precludes disjoint-access parallelism, and the copy-avoidance optimisation requires OS support or scheduler activations [Anderson92].

Turek *et al.* address the problem of serialisation by devising a construction that may be applied to deadlock-free lock-based algorithms [Turek92]. Each lock in the unmodified algorithm is replaced by an ownership reference which is either *nil* or points to a continuation describing the sequence of *virtual instructions* that remain to be executed by the lock 'owner'. This allows conflicting operations to execute these instructions on behalf of the owner and then take ownership themselves, rather than blocking on the original process. Interpreting a continuation is cumbersome: after each 'instruction' is executed, a virtual program counter and a non-wrapping version counter are atomically modified using a double-width CAS operation which acts on an adjacent pair of memory locations. This approach permits parallelism to the extent of the original lock-based algorithm; however, interpreting the continuations is likely to cause significant performance loss.

Barnes proposes a similar technique in which mutual-exclusion locks are replaced by *operation descriptors* [Barnes93]. Lock-based algorithms are converted to operate on a private copy of the data structure; then, after determining the sequence of updates to apply, each required operation record is acquired in turn, the updates are performed, and finally the operation records are released. Copying is avoided if contention is low by observing that the private copy of the data structure may be cached and reused across a sequence of operations. This two-phase algorithm requires a nestable LL/SC operation, which has no efficient implementation on current processors.

Greenwald introduces 'two-handed emulation' to serialise execution of concurrent operations [Greenwald02]. This requires each operation to register its intent by installing an operation descriptor in a single shared location. As in the scheme by Turek *et al.*, the operation then uses DCAS to simultaneously update the shared structure and a 'program counter' within the operation descriptor. Processes which conflict with the current operation use the virtual program counter to help it to completion while preserving exactly-once semantics. This technique has limited applicability because of its dependence on DCAS. It is also not disjoint-access parallel: indeed, all operations are serialised.

2.3.3 Programming abstractions

Although the universal constructions described in Section 2.3.2 have the benefit of requiring no manual modification to existing sequential or lock-based programs, each exhibits some substantial performance or implementation problems on current systems which places it beyond practical use. Another class of techniques provides high-level programming abstractions which, although not automatic 'fixes' to the problem of constructing non-blocking algorithms, make the task of implementing non-blocking data structures much easier compared with using atomic hardware primitives directly. The two best-known abstractions are multi-word compare-&-swap (MCAS) and software transactional memory (STM), which have both received considerable treatment in the literature.

These abstractions are not intended for direct use by application programmers. Instead it is expected that programmers with parallel-systems experience will implement libraries of support routines. Another possibility is to use MCAS or STM to implement run-time support for higher-level programming-language constructs such as monitors [Hoare74], atomic statements [Liskov83] or conditional critical regions [Hoare85]. Existing implementations of these constructs have generally been pessimistic in terms of the parallelism they exploit; for example, critical regions are serially executed on a single processor [Brinch Hansen78], or conservative locking is employed [Lomet77]. Efficient implementations of MCAS and STM may allow these constructs to be revisited and implemented with improved performance characteristics.

2.3.3.1 *Multi-word compare-&-swap (MCAS)*

MCAS is a straightforward extension of the well-known CAS operation to update an arbitrary number of memory locations simultaneously. An MCAS operation is specified by a set of tuples of the form (*address*, *expected*, *new*); if each *address* contains the *expected* value then all locations are atomically updated to the specified *new* values. The costs of the algorithms described here frequently depend on the maximum number of processes that may concurrently attempt an MCAS operation, which I denote by N. Many also require a strong form of LL/SC that can arbitrarily nested: this form of LL/SC is not supported by existing hardware.

Israeli and Rappaport describe a layered design which builds a lock-free MCAS from strong LL/SC primitives [Israeli94]. They describe a method for building the required LL/SC from CAS that reserves N bits within each updated memory location; the MCAS algorithm then proceeds by load-locking each location in turn, and then attempting to conditionally-store each new value in turn. The cost of implementing the required strong LL/SC makes their design impractical unless the number of concurrent MCAS operations can be restricted to a very small number.

Anderson and Moir present a wait-free version of MCAS that also requires strong LL/SC [Anderson95]. Their method for constructing the required LL/SC requires at least $\log N$ reserved bits per updated memory location, which are used as a version number to detect updates which conflict with an LL/SC pair. Although this bound is an improvement on previous work, considerable bookkeeping is required to ensure that version numbers are not reused while they are still in use by some process. A further drawback is that the accompanying *Read* operation, used to read the current value of a location that may concurrently be subject to an MCAS, is based on primitives that acquire exclusive cache-line access for the location. This may have a significant performance cost if *Read* is executed frequently.

Moir developed a stream-lined version of this algorithm which provides 'conditionally wait-free' semantics [Moir97]. Specifically, the design is lock-free but an out-of-band helping mechanism may be specified which is then responsible for helping conflicting operations to complete. This design suffers many of the same weaknesses as its ancestor; in particular, it requires a strong version of LL/SC and a potentially expensive *Read* operation.

Anderson *et al.* provide specialised versions of MCAS suitable for both uniprocessor and multiprocessor priority-based systems [Anderson97]. Both algorithms store a considerable amount of information in memory locations subject to MCAS updates: a valid bit, a process identifier ($\log N$ bits), and a 'count' field (which grows with the logarithm of the maximum number of addresses specified in an MCAS operation). Furthermore, the multiprocessor algorithm requires certain critical sections to be executed with preemption disabled, which is not feasible in many systems.

Greenwald presents a simple design in his PhD dissertation [Greenwald99], which constructs a record describing the entire operation and installs it into a single shared location which indicates the sole in-progress MCAS operation. If installation is prevented by an already-running MCAS, then the existing operation is helped to completion and its record is then removed. Once installed, an operation proceeds by executing a DCAS operation for each location specified by the operation: one update is applied to the *address* concerned, while the other updates a progress counter in the operation record. Note that Greenwald's design is not disjoint-access parallel, and that it requires DCAS.

2.3.3.2 *Software transactional memory (STM)*

Herlihy and Moss first introduced the concept of a *transactional memory*, which allows shared-memory operations to be grouped into atomic transactions [Herlihy93b]. They present a hardware design which leverages existing multiprocessor cache-coherency mechanisms. Transactional memory accesses cause the appropriate cache line to be loaded into a private *transactional cache*, the contents of which are written back to main memory at the end of a successful transaction. The transactional cache snoops memory operations from other processors, and fails a remote transaction if it attempts to obtain exclusive access to a cache line that is currently 'in use' by a local transaction. Although this means that the protocol is not non-blocking, in practice only a faulty processor will fail remote transactions indefinitely. Starvation of individual processors can be dealt with cooperatively in software; for example, by 'backing off' when contention is experienced. The major practical drawback of this design is that it requires hardware modifications — convincing processor designers that a new untrialled mechanism is worth the necessary modifications to the instruction-set architecture and increased bus-protocol complexity is likely to be a significant battle.

Shavit and Touitou address this problem by proposing a software-based lock-free transactional memory [Shavit95]. A notable feature is that they abort contending transactions rather than recursively helping them, as is usual in lock-free algorithms; non-blocking behaviour is still guaranteed because aborted transactions help the transaction that aborted them before retrying. Their design supports only 'static' transactions, in which the set of accessed memory locations is known in advance — this makes it difficult to implement certain common operations, such as traversal of linked structures. A further limitation is that the algorithm requires a nestable LL/SC operation.

Moir presents lock-free and wait-free STM designs [Moir97] which provide a dynamic programming interface, in contrast with Shavit and Touitou's static interface. The lock-free design divides the transactional memory into fixed-size blocks which form the unit of concurrency. A header array contains a word-size entry for each block in the memory, consisting of a block identifier and a version number. Unfortunately arbitrary-sized memory words are required as there is no discussion of how to handle overflow of the version number. The design also suffers the same drawbacks as the conditionally wait-free MCAS on which it builds: bookkeeping space is statically allocated for a fixed-size heap, and the read operation is potentially expensive. Moir's wait-free STM extends his lock-free design with a higher-level helping mechanism based around a 'help' array which indicates when a process i has interfered with the progress of some other process j: in this situation i will help j within a finite number of execution steps.

Recently, Herlihy *et al.* have implemented an obstruction-free STM with many desirable properties [Herlihy03b]. Firstly, the memory is dynamically sized: memory blocks can be created and destroyed on the fly. Secondly, an implementation is provided which builds on a readily-available form of the CAS primitive (this is at the cost of an extra pointer indirection when accessing the contents of a memory block, however). Finally, the design is disjoint-access parallel, and transactional reads do not cause writes to occur in the underlying STM implementation. These features serve to significantly decrease contention in many multiprocessor applications, and are all shared with my own lock-free STM which I describe in the next chapter. This makes Herlihy *et al.*'s design an ideal candidate for comparison in Chapter 6. The major difference is that my STM is lock-free, and so does not require an out-of-band mechanism for relieving contention to guarantee progress. As noted in Section 2.1, investigation of effective contention-avoidance strategies in obstruction-free algorithms is still an area of active research.

Harris and Fraser present an obstruction-free STM with a very different programming interface [Harris03]. By storing the heap *in the clear*, without dividing it into transactional objects, they avoid the overhead of copying an entire object when it is accessed by a transaction. This may be particularly beneficial when just a few locations are accessed within a large object. The interface is further motivated by the desire to incorporate transactional techniques into existing run-time environments. The in-the-clear representation allows non-transactional reads and writes to be implemented as usual, reducing the number of required modifications to the compiler or run-time system. The direct heap representation is implemented by maintaining *out-of-band* ownership records to manage synchronisation between concurrent transactions. A hash function can be used to map heap locations to a smaller set of ownership records. To achieve obstruction-freedom, transactions are allowed to *steal* ownership records from each other. This requires careful merging of the existing transaction's state, which is complicated by the fact that multiple heap locations are likely to map to the same record. Stealing an ownership record requires a double-width CAS primitive to allow atomic update of the reference count and transaction pointer contained within each ownership record. Unlike DCAS however, double-width CAS is supported efficiently by most modern architectures. An empirical evaluation of the in-the-clear interface compared with traditional object-based APIs is the subject of ongoing work: the former eliminates per-object overheads but trades this for increased overhead on every transactional memory access.

It is interesting to note the similarities between recent STM designs and work on optimistic concurrency control in transactional database systems [Herlihy90a, Kung81, Wu93]. Care is needed when designing database systems to ensure that disc-access times and network latencies (in distributed environments) do not cripple performance. These potential bottlenecks can be avoided by optimistically permitting overlapping execution of possibly-conflicting transactions. However, unlike transactional memories, the intention is not necessarily to execute these transactions simultaneously on different processors, but to harness available bandwidth to support a higher transaction rate[1]. Since CPU and memory speeds are not usually a bottleneck in database systems, simpler mechanisms can be used to enforce the ACID database properties when transactions attempt to commit. For example, a single *concurrency manager* might generate a global timestamp for each transaction and use this to safely resolve conflicts. This can work well in a database system, but using such a centralised approach to implement a transactional memory would defeat the object of achieving a high degree of CPU parallelism with minimal inter-processor communication overheads.

2.3.4 Ad hoc data structures

Although there are many universal constructions and programming abstractions that seek to ease the task of implementing complex data structures, practical concerns have caused most designers to resort to building non-blocking algorithms directly from machine primitives such as CAS and LL/SC. Consequently there is a large body of work describing *ad hoc* designs for fairly simple data structures such as stacks, deques, and lists. It is worth noting that more complex structures, such as binary search trees, are not represented at all, which indicates just how difficult it is to build data structures directly from single-word primitives.

Massalin and Pu describe the implementation of the lock-free *Synthesis* kernel for a multiprocessor system based on the Motorola 68030 [Massalin91]. The use of the 68030 processor means that several of the lock-free algorithms used in key kernel components can safely depend on architectural support for the DCAS primitive. However, this does mean that the kernel is not directly portable to any other processor architecture.

The Cache Kernel, described by Greenwald and Cheriton [Greenwald96], suffers from the same limitation. However, they note that in certain limited circumstances, DCAS may be implemented in software.

Although non-blocking designs exist for many simple data structures, such as queues, stacks and deques, the only search structure which has received significant attention is the singly-linked list. Valois [Valois95] introduced the first lock-free list design based on CAS. Although his algorithm allows a high degree of parallelism, its implementation is very involved. Indeed, several later papers describe errors relating to the management of reference-counted storage [Michael95, Harris01]. Harris presented a simpler and significantly more

[1] Disc packs and networked servers usually support overlapping (or *pipelined*) requests. This increases system bandwidth because the latencies of several data requests can be overlapped.

efficient design which uses ‘pointer marking’ to indicate when a node is logically deleted [Harris01]. In Chapter 4 I apply the same technique to skip lists.

Greenwald notes that the availability of DCAS makes linked lists, and a great number of other data structures, much easier to implement [Greenwald99]. However, no modern architecture implements this primitive.

2.3.5 Memory management

Many non-blocking algorithms in the literature are presented in pseudocode which assumes that automatic garbage collection is provided as a run-time service. This ignores the problem that many languages do not provide this support and, furthermore, that most general-purpose garbage collectors are not non-blocking or are unsuitable for highly-parallel applications. To deal with this, a range of non-blocking memory-management techniques have been suggested.

Herlihy and Moss present a lock-free copying garbage collector [Herlihy92]. Each time a process updates an object it creates a new copy within a per-process *to* region. Periodically each process will create a new *to* region; the old region is added to a set of *from* regions to be reclaimed when it is safe to do so. Each process occasionally runs a scan routine which scans for objects in a *from* region, copies them to its own *to* region, and updates references to that object to point at the new copy. When a *from* region contains no live objects, and each process has subsequently passed a ‘safe’ point, then that region can safely be reclaimed. Apart from the copying overhead, the major drawback of this scheme is the numerous parameters which are not fully explored in the paper. For example, the frequency with which *to* regions are retired, and the rate at which objects are scanned, are both likely to significantly affect heap size and execution time. Experimental analysis is required to determine satisfactory values for these parameters.

Valois uses reference counts to ensure that an object is not reused while any thread still holds a pointer to it [Valois95]. As there may be an arbitrary delay between obtaining a reference to an object and incrementing the reference count, objects reclaimed via reference counts must retain their type forever. Detlefs *et al.* solve this by using DCAS to increment the counter while simultaneously checking that the object remains globally accessible [Detlefs01]. However, all reference counting schemes suffer two serious drawbacks. Firstly, they can incur a considerable cost in maintaining the reference counts. This is particularly true for operations which read many objects: updates to reference counts may cause read-only objects to become a contention bottleneck. Secondly, even when using DCAS it is not safe to garbage-collect virtual address space as this may cause the program to fault if an address which previously contained a reference count becomes inaccessible (e.g., due to reclamation of page-table entries within the OS, resulting in a *page-not-present* fault).

Kung and Lehman describe a system in which garbage objects are placed on a temporary 'limbo list' [Kung80]. Periodically this list is copied by the garbage collector and the status of each process in the system is noted. When all processes have completed their current operation then all objects in the copied list can safely be reclaimed: no references to these objects exist in the shared structure so newly-started operations cannot reach them. This technique, which has subsequently been applied to several parallel algorithms [Manber84, Pugh90a], has the advantage that it does not require extra atomic updates to safely access objects. However, the limbo list will grow until all memory is consumed if a process stalls for any reason, such as an unfortunate scheduling decision. Cache utilisation may also be harmed because objects are likely to be evicted from processor caches before they are reclaimed.

Limbo lists have recently been applied to memory reclamation in the read-copy update (RCU) scheme, developed for lock-free data structures in the Linux kernel [Arcangeli03]. This method of garbage collection is particularly suited to the non-preemptive environment in which the Linux kernel executes. Since no operation can block, and all processors can be trusted to cooperate, garbage lists can be reclaimed within a reasonable time before they become excessively long.

Recent work by Michael describes a scheme in which processes 'publish' their private references in a shared array of hazard pointers [Michael02]. The garbage collector must not free any object referenced by this array. By reclaiming storage only when a sufficient number of objects are on the garbage list, the cost of scanning the array is amortised. However, the cost of updating a hazard pointer when traversing objects can be significant. On modern processors a memory barrier must be executed after updating a hazard pointer: implementing this barrier in my own binary tree implementation increased execution time by around 25%.

Herlihy *et al.* formalise object use in a concurrent system by formulating the Repeat Offender Problem [Herlihy02]. They present a solution to this problem, called Pass the Buck, which is similar in many respects to Michael's hazard pointers. The primary difference is that the cost of scanning the hazard array is not amortised across a number of garbage objects. However, better cache utilisation may be achieved by not delaying reuse of idle objects.

2.4 Summary

I began this chapter by introducing the standard terminology used to classify non-blocking systems. By themselves, these terms do not describe all the desirable features that a practical lock-free data structure is likely to possess — I therefore introduced *disjoint-access parallelism* and *linearisability* which provide additional performance and usability guarantees.

In the remainder of the chapter I described previous work relating to non-blocking data structures. A recurring problem is that existing methods for simplifying lock-free programming — universal constructions and high-level programming abstractions — are *impractical* for general use. Either they restrict parallelism, they require large amounts of memory for internal data structures, or they make unreasonable assumptions about the underlying hardware (requiring unsupported atomic primitives or arbitrary-width memory words). Using hardware primitives directly is not a feasible alternative: apart from very simple data structures it is too difficult to construct complex operations from single-word atomic primitives.

If lock-free programming is to be a viable alternative to using locks, we need tools that simplify the implementation of practical lock-free systems. In the next chapter I present new designs for two programming abstractions that provide this necessary support.

Chapter 3

Practical lock-free programming abstractions

In Chapter 2 I described how existing implementations of lock-free programming abstractions are impractical for general use. In this chapter I introduce the first practical lock-free designs for two easy-to-use abstractions: multi-word compare-&-swap, and software transactional memory. These new designs make it easier for programmers to implement efficient lock-free data structures — this is demonstrated in Chapters 4 and 6 which present the design and evaluation of lock-free versions of three real-world search structures.

3.1 Introduction

Usability is perhaps the biggest obstacle to wider use of lock-free programming. Although algorithms for simple data structures, such as array-based queues and stacks, have been known for many years, it seems that existing hardware primitives are too difficult to apply directly to more complex problems.

The limitations of single-word atomic operations have caused many researchers to suggest that better hardware support is necessary before lock-free design can be considered as a general-purpose alternative to using mutual exclusion. Although a range of possible hardware extensions have been suggested, it is unlikely that any one of them will be accepted by processor designers unless it demonstrably improves the performance of existing programs. This impasse is unlikely to be resolved unless a new style of concurrent programming appears that can immediately benefit from better hardware support.

One way to tackle the complexity of lock-free programming on current hardware is to build a more intuitive programming abstraction using existing primitives. A range of abstractions have been proposed which present a tradeoff between performance and ease of use.

Multi-word compare-&-swap (MCAS) extends the well-known hardware CAS primitive to operate on an arbitrary number of memory locations simultaneously. This avoids the greatest difficulty in using single-word primitives directly: ensuring that a group of related updates occurs atomically.

Although MCAS ensures consistency between groups of update operations, some data structures also require consistency guarantees for read-only operations. To this end, *software transactional memory* (STM) provides a higher-level transactional interface for executing groups of reads and writes to shared memory. Despite these advantages, STM may not be the best abstraction to use in all situations: as I will show later, the easier-to-use interface and stronger synchronisation often results in reduced performance compared with MCAS.

The main obstacle to wider use of these abstractions in lock-free programs is that existing designs are impractical. Firstly, their performance is lacklustre: either per-operation overhead is very high, or non-conflicting operations are unable to proceed in parallel. In both cases, the designs look very unattractive compared with lock-based solutions. A second problem is that non-existent primitives are assumed, such as DCAS or a 'strong' form of LL/SC that allows nesting. In this chapter I introduce the first practical lock-free MCAS design, and an efficient object-based transactional memory called FSTM.

3.2 Multi-word compare-&-swap (MCAS)[1]

MCAS extends the single-word CAS primitive to operate on multiple locations simultaneously. More precisely, MCAS is defined to operate on N distinct memory locations (a_i), expected values (e_i), and new values (n_i): each a_i is updated to value n_i if and only if each a_i contains the expected value e_i before the operation.

```
atomically bool MCAS (int N, word *a[ ], word e[ ], word n[ ]) {
   for ( int i := 0; i < N; i++ ) if ( *a[i] ≠ e[i] ) return FALSE;
   for ( int i := 0; i < N; i++ ) *a[i] := n[i];
   return TRUE;
}
```

[1] The work described in this section was conducted jointly with Dr T L Harris.

[2] The **atomically** keyword indicates that the function is to be executed as-if instantaneously.

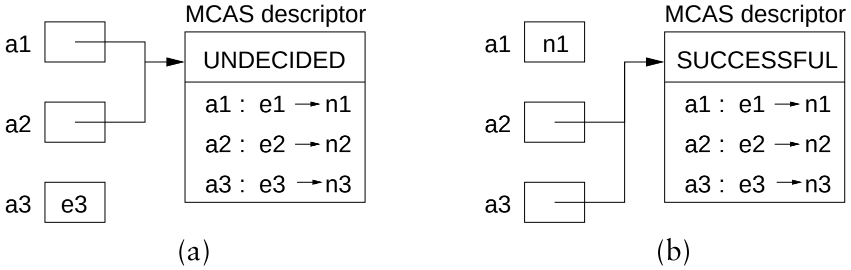

Figure 3.1: Two snapshots of a successful MCAS operation attempting to update three memory locations (a_1, a_2 and a_3). Snapshot (a) occurs just before location a_3 is acquired. The operation is *undecided* at this point, so all three locations still have their original 'logical' value, e_i. Snapshot (b) occurs after the operation is deemed successful: all locations are already logically updated to n_i, and the release phase has just physically updated location a_1.

3.2.1 Design

Consider the following two requirements that must be satisfied by any valid MCAS design. Firstly, it should appear to execute atomically: a successful MCAS must instantaneously replace the expected value in each specified memory location with its new value. Secondly, if an MCAS invocation fails because some location does not contain the expected value then it must be able to 'undo' any updates it has made so far and leave all memory locations as they were before it began executing. It is not immediately obvious how these requirements can be satisfied by a design based on atomic read-modify-write updates to individual memory locations (as provided by CAS). How can multiple locations appear to be updated simultaneously, and how can an in-progress operation be undone or rolled back?

I satisfy both these requirements by using a *two-phase* algorithm. The first phase gains ownership of each data location involved in the operation. If the operation successfully gains ownership of every location then it is deemed successful and all updates atomically become visible. This *decision point* is followed by a second phase in which each data location is updated to its new value if the first phase succeeded, or reverted to its old value if the first phase failed. Illustrative snapshots of an example MCAS operation are shown in Figure 3.1. Note that this two-phase structure has a further advantage in that an in-progress operation can arrange to 'describe itself' to any remote operation that it may obstruct: this allows recursive helping to be used to obtain the required lock-free progress guarantee.

Each MCAS operation creates a *descriptor* which fully describes the updates to be made (a set of (a_i, e_i, n_i) tuples) and the current status of the operation

```
typedef struct {
   int N;
   word *a[ ], e[ ], n[ ], status;
} MCASDesc;
bool MCAS (int N, word *a[ ], word e[ ], word n[ ]) {
   MCASDesc *d := new MCASDesc;
   (d→N, d→a, d→e, d→n, d→status) := (N, a, e, n, UNDECIDED);
   AddressSort(d); /* Memory locations must be sorted into address order. */
   return MCASHelp(d);
}
word MCASRead (word *a) {
   word v;
   for ( v := CCASRead(a); IsMCASDesc(v); v := CCASRead(a) )
      MCASHelp((MCASDesc *)v);
   return v;
}
bool MCASHelp (MCASDesc *d) {
   word v, desired := FAILED;
   bool success;
   /* PHASE 1: Attempt to acquire each location in turn. */
   for ( int i := 0; i < d→N; i++ )
      while ( TRUE ) {
         CCAS(d→a[i], d→e[i], d, &d→status);
         if ( ((v := *d→a[i]) = d→e[i]) ∧ (d→status = UNDECIDED) ) continue;
         if ( v = d ) break; /* move on to next location */
         if ( ¬IsMCASDesc(v) ) goto decision_point;
         MCASHelp((MCASDesc *)v);
      }
   desired := SUCCESSFUL;
decision_point:
   CAS(&d→status, UNDECIDED, desired);
   /* PHASE 2: Release each location that we hold. */
   success := (d→status = SUCCESSFUL);
   for ( int i := 0; i < d→N; i++ )
      CAS(d→a[i], d, success ? d→n[i] : d→e[i]);
   return success:
}
```

Figure 3.2: Two-phase multi-word CAS (MCAS) algorithm. MCASRead is used by applications to read from locations which may be subject to concurrent MCAS operations. Conditional CAS (CCAS) is used in phase one to ensure correctness even when memory locations are subject to 'ABA' updates.

(*undecided*, *failed*, or *successful*). The first phase of the MCAS algorithm then attempts to update each location a_i from its expected value, e_i, to a reference to the operation's descriptor. This allows processes to distinguish currently-owned memory locations, enables recursive helping of incomplete operations, and permits atomic update of 'logical' memory values at the MCAS decision point. Note that only owned locations can contain references to descriptors; furthermore, descriptors themselves will never be subject to MCAS updates.

The current logical (or application) value of an owned location is found by interrogating the MCAS descriptor that is installed there. The descriptor is searched to find the appropriate address, a_i. The current logical value is then either e_i, if the descriptor status is currently *undecided* or *failed*, or n_i, if the status is *successful*. All locations that are not currently owned store their logical value directly, allowing direct access with no further computation or memory accesses. To determine whether a location is currently owned, IsMCASDesc(p) is used to discover whether the given pointer p is a reference to an MCAS descriptor. Various implementations of IsMCASDesc are discussed in Chapter 5.

Pseudocode for MCAS and MCASRead is shown in Figure 3.2. Note that MCAS must acquire update locations in address order. This ensures that recursive helping eventually results in system-wide progress because each level of recursion must be caused by a conflict at a strictly higher memory address than the previous level. Recursive helping is therefore bounded by the number of memory locations in the shared heap. To ensure that updates are ordered correctly it sorts the update locations before calling MCASHelp (lines 8–9). The sort can be omitted if the caller ensures that addresses are specified in some global total order. If addresses are not ordered then a recursive loop may be entered.

The CCAS operation used at line 23 in the first phase of the algorithm is a *conditional compare-&-swap*. CCAS uses a second *conditional* memory location to control the execution of a normal CAS operation. If the contents of the conditional location are zero then the operation proceeds, otherwise CCAS has no effect.

```
atomically void CCAS (word *a, word e, word n, word *cond) {
   if ( (*a = e) ∧ (*cond = 0) ) *a := n;
}
```

The use of CCAS in MCAS requires that the *undecided* status value is represented by zero, thus allowing shared-memory locations to be acquired only if the outcome of the MCAS operation is not yet decided. This prevents a phase-one update from occurring 'too late': if a normal CAS were used then each memory location might be updated more than once because a helping process could incorrectly reacquire a location after the MCAS operation has already succeeded. This can happen if a CAS to install the descriptor is delayed, and in the meantime the memory location is modified *back* to the expected value: this is commonly called the *ABA problem* [IBM70].

The CCAS design in Figure 3.3 makes the following simplifying assumptions:

- Location *cond* must not be updated by CCAS or MCAS.
- Memory locations which might be updated by CCAS should be accessed using CCASRead (or MCASRead, which is itself based on CCASRead).

```
1   typedef struct {
       word *a, e, n, *cond;
3   } CCASDesc;
    void CCAS (word *a, word e, word n, word *cond) {
5      CCASDesc *d := new CCASDesc;
       word v;
7      (d→a, d→e, d→n, d→cond) := (a, e, n, cond);
       while ( (v := CAS(d→a, d→e, d)) ≠ d→e ) {
9         if ( ¬IsCCASDesc(v) ) return;
          CCASHelp((CCASDesc *)v);
11     }
       CCASHelp(d);
13  }
    word CCASRead (word *a) {
15     word v;
       for ( v := *a; IsCCASDesc(v); v := *a )
17        CCASHelp((CCASDesc *)v);
       return v;
19  }
    void CCASHelp (CCASDesc *d) {
21     bool success := (*d→cond = 0);
       CAS(d→a, d, success ? d→n : d→e);
23  }
```

Figure 3.3: Conditional compare-&-swap (CCAS). CCASRead is used to read from locations which may be subject to concurrent CCAS operations.

The pseudocode design begins by installing a *CCAS descriptor* in the location to be updated (line 8). This ensures that the location's logical value is the expected value while the conditional location is tested, so that a successful CCAS operation linearises (atomically occurs) when the conditional location is read from. If the update location doesn't contain the expected value then CCAS fails (line 9); if it contains another CCAS descriptor then that operation is helped to complete before retrying (line 10).

If the update location is successfully acquired, the conditional location is tested (line 21). Depending on the contents of this location, the descriptor is either replaced with the new value, or with the original expected value (line 22). CAS is used so that this update is performed exactly once even when the CCAS operation is helped to complete by other processes.

3.3 Software transactional memory

Although MCAS eases the burden of ensuring correct synchronisation of updates, many data structures also require consistency among groups of read operations. Consider searching within a move-to-front list, in which a successful search promotes the discovered node to the head of the list. As indicated in Fig-

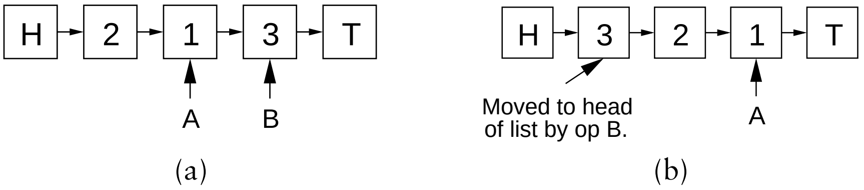

Figure 3.4: The need for read consistency: a move-to-front linked list subject to two searches for node 3. In snapshot (a), search A is preempted while passing over node 1. Meanwhile, in snapshot (b), search B succeeds and moves node 3 to the head of the list. When A continues execution, it will incorrectly fail.

ure 3.4, a naïve search algorithm which does not consider synchronisation with concurrent updates may incorrectly fail, even though each individual read from shared memory operates on a consistent snapshot of the list.

To deal with this problem I now turn to a higher-level abstraction, known as *software transactional memory* (STM) because it groups shared-memory operations into transactions that appear to succeed or fail atomically. As discussed in Chapter 2, existing STM designs vary greatly in the interface they provide to application programmers and the underlying transactional memory mechanisms.

In this section I present a new transactional-memory design called FSTM. It is the first lock-free design with all the following desirable features:

- Dynamic programming interface: it is not necessary to have precomputed a transaction before presenting it to the transactional memory.
- Small, fixed memory overhead per block of transactional memory, and per transactional read or write.
- Small number of shared-memory operations required to implement each transactional read or write, assuming reasonable levels of contention.

It is important to note that the advantages of FSTM over MCAS generally come at a cost. As I show in Chapter 6, a data structure implemented using MCAS will usually outperform an equivalent FSTM-based design. However, in many situations the extra time and complexity associated with programming using MCAS will not be justified by the run-time benefits.

Following several previous transactional memory designs [Moir97,Herlihy03b], FSTM groups memory locations into contiguous blocks, or *objects*, which act as the unit of concurrency and update. Rather than containing pointers, data structures contain opaque *object references*, which may be converted to directly-usable machine pointers by *opening* them as part of a transaction. Each object that is opened during a transaction is remembered as a consistency assumption to be checked before closing the object during the commit phase.

This section begins by introducing the object-based programming interface and a small example application that illustrates its use. Although the interface is similar in spirit to that of Herlihy *et al.*, there are both syntactic and semantic differences that can affect the implementation and execution of STM-based applications. I then proceed to describe in detail how FSTM implements this interface, and strives to hide concurrency issues from the application programmer. I conclude this section by discussing extra features and extensions that can be added to the basic FSTM design to make it more useful in a wider range of applications — several of these features are required by STM-based algorithms in the next chapter.

3.3.1 Programming interface

FSTM supports a dynamic programming interface in which transactions can be started and committed, and objects opened for access, at arbitrary program points. The following functions are provided by the API:

`stm *new_stm (int object_size)`

Creates a new transactional memory supporting objects of length object_size.

`void free_stm (stm *mem)`

Destroys a previously-created STM, and immediately releases all memory associated with it, including all objects, to the garbage collector.

`(stm_obj *, void *) new_object (stm *mem)`

Creates a new object with respect to an existing transactional memory, and returns two values: the first is an object reference that can be shared with other processes, and the second is a directly-usable machine pointer that can be used to initialise the object. This allows an application to initialise a new object outside any transaction, before sharing it with any other processes.

`void free_object (stm *mem, stm_obj *o)`

Frees a previously-allocated object to the garbage collector. This is used when a transaction has successfully removed all references to an object from shared memory.

`stm_tx *new_transaction (stm *mem)`

Starts a new transaction with respect to an existing transactional memory. Returns a transaction identifier which can be used to manage the transaction and open objects for access and update. Note that transactions cannot be nested; however, I discuss how this restriction might be lifted in Section 3.3.3.

void *open_for_reading (stm_tx *t, stm_obj *o)

Opens an object for read-only access with respect to an in-progress transaction. The returned pointer cannot be used for updates, but can be used for read accesses until the transaction commits. This function, along with open_for_writing, can safely be invoked multiple times with identical parameters.

void *open_for_writing (stm_tx *t, stm_obj *o)

Opens an object for reading and writing with respect to an in-progress transaction. The returned pointer is safe to use for any type of access until the transaction commits.This function is idempotent: if the object has already been opened for write access within transaction *t* then the same pointer will be returned again. This function, along with open_for_reading, can safely be invoked multiple times with identical parameters — a previously read-only object will be upgraded for write access on the first invocation of this function.

bool commit_transaction (stm_tx *t)

Attempts to commit a transaction by checking each opened object for consistency. If all open objects are consistent then the transaction succeeds and all updates atomically become visible. Otherwise the transaction fails. In all cases the transaction identifier becomes invalid.

void abort_transaction (stm_tx *t)

Aborts an in-progress transaction. The transaction identifier immediately becomes invalid, all opened objects are closed, and all updates are lost. This is useful if the application determines that a transaction cannot complete successfully and wants to avoid the expense of a commit operation that cannot possibly succeed.

bool validate_transaction (stm_tx *t)

Checks the consistency of an in-progress transaction to determine whether it can possibly complete successfully. If validation succeeds then the transaction may commit successfully; if validation fails then the transaction will certainly fail to commit. This is useful in a number of scenarios; for example, to check consistency before wasting time performing an expensive computation.

```
typedef struct { stm *mem; stm_obj *head; } list;
typedef struct { int key; stm_obj *next; } node;
list *new_list (void) {
    node *n;
    list *l := new list;
    l→mem := new_stm(sizeof node);
    (l→head, n) := new_object(l→mem);
    (n→key, n→next) := (0, NULL);
    return l;
}
void list_insert (list *l, int k) {
    stm_obj *prev_obj, *new_obj;
    node *prev, *new;
    (new_obj, new) := new_object(l→mem);
    new→key := k;
    do {
        stm_tx *tx := new_transaction(l→mem);
        (prev_obj, prev) := (l→head, open_for_reading(tx, l→head));
        while ( (prev→next ≠ NULL) ∧ (prev→key < k) )
            (prev_obj, prev) := (prev→next, open_for_reading(tx, prev→next));
        prev := open_for_writing(tx, prev_obj);
        (new→next, prev→next) := (prev→next, new_obj);
    } while ( ¬commit_transaction(tx) );
}
```

Figure 3.5: Linked-list creation and insertion, in which each list node is an STM object.

Figure 3.5 shows how this interface might be used to implement ordered linked-list creation and insertion. Each list node references its neighbour using an FSTM object reference rather than using a direct machine pointer. To ensure consistency, all shared-memory references in a transactional memory must be stored in this way and then opened for direct access within the scope of individual transactions. For this reason, and to aid code readability, local variables representing nodes in the list occur in pairs: one referring to the opaque object identifier, and the other to a directly-accessible (but transaction-specific) version of the node.

The above programming interface has several limitations that might be a nuisance in certain applications: for example, each instantiated transactional memory supports only a single size of object, and transactions cannot safely be nested. I discuss these restrictions in more detail, and describe how they might be lifted, in Section 3.3.3.

3.3.2 Design

When designing FSTM I attempted to minimise the total size of a transactional memory, and the number of shared-memory operations required to perform a transaction, by paying careful attention to heap layout. The current layout is chosen to be memory-efficient while supporting a very lightweight commit oper-

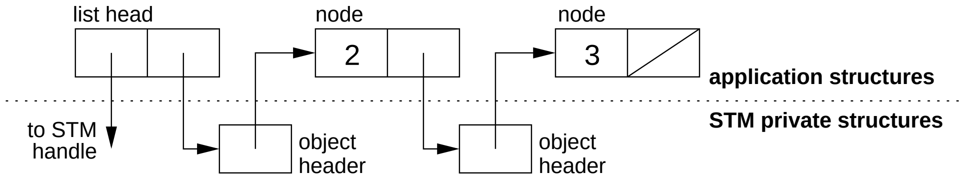

Figure 3.6: Example FSTM-based linked list structure created by pseudocode in Figure 3.5. List nodes are chained together via object headers, which are private to the STM. References to object headers are known as object references and must be converted to list-node references using open_for_reading or open_for_writing within the scope of a transaction.

ation. Making transactions commit quickly is particularly beneficial because it is only during their commit phase that they modify shared memory and become visible to other processes — a fast commit therefore reduces the window of opportunity for transactions to directly 'see' each other and incur the overheads of recursive helping.

I begin this section by describing the memory layout when no transactions are in progress. I then describe how transactions build a view of memory as objects are opened for reading and writing, how this view is tested for consistency during a transaction's commit phase, and the procedure for making apparently-atomic updates to the transactional memory.

3.3.2.1 *Transactional memory layout*

The current contents of an FSTM object are stored within a *data block*. Outside of a transaction context, shared references to an FSTM object point to a word-sized *object header* which tracks the current version of the object's data via a pointer to the current data block. This pointer is modified to point at a new data block each time an update is successfully committed. The object references introduced in Section 3.3.1 are implemented as pointers to object headers; however, object references are opaque to application programmers and can be used only to uniquely identify or name an object, and to open an object for reading or writing within a transaction via FSTM interface calls.

Figure 3.6 shows an example FSTM-based structure which might be created by the linked-list pseudocode described in Section 3.3.1. The nodes of the linked list are objects, so their contents are not directly accessible by the application. Instead, a reference to an object header must be converted by FSTM into a private pointer to the current data block.

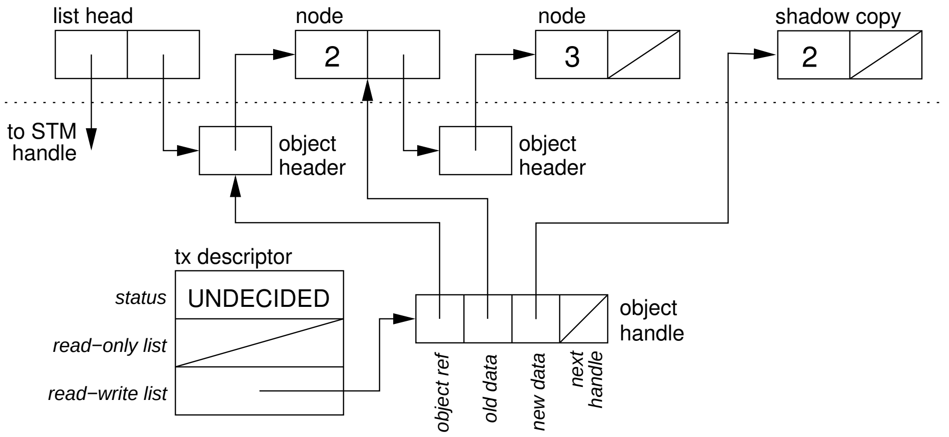

Figure 3.7: Example of a transaction attempting to delete node 3 from the list introduced in Figure 3.6. The transaction has accessed one object (node 2) which it has opened for writing. The read-only list is therefore empty, while the read-write list contains one object handle describing the modified node 2.

3.3.2.2 *Creating a transaction and accessing objects*

The state of incomplete transactions is encapsulated within a per-transaction *descriptor* structure which indicates the current status of the transaction and stores information about every object which has been opened for use within the transaction.

When an object is opened for read-only access, a new *object handle* is added to a 'read-only list' within the transaction descriptor. This list is indexed by object reference; each handle stores the object reference and data-block pointer (as read from the object header at the time of opening). The data pointer is returned to the application for use solely within the scope of the transaction.

The procedure is similar for objects that are opened for writeable access, except that a *shadow copy* of the data block is created. The address of this copy is stored, together with the object reference and the current data pointer, within a 'read-write list' inside the transaction descriptor. It is the shadow copy which is returned to the application: updates to this copy remain private until the transaction commits.

Figure 3.7 illustrates the use of transaction descriptors and object handles by showing a transaction in the process of deleting a node from an ordered linked list. The transaction descriptor indicates that the current status of the transaction is *undecided*, and contains pointers to the empty read-only list and the singleton read-write list. The sole object handle contains references to the only opened object, the version of the object that was up-to-date at that time of opening, and also the shadow copy containing pending updates by this transaction.

3.3.2.3 *Committing a transaction*

A transaction's commit stage has a two-phase structure that is very similar to the MCAS algorithm described in Section 3.2. Indeed, that algorithm can be used almost unmodified:

Acquire phase The header of each opened object is acquired in some global total order[1], by replacing the data-block pointer with a pointer to the transaction descriptor. If we see another transaction's descriptor when making the update then that transaction is recursively helped to complete.

Decision point Success or failure is then indicated by updating the status field of the transaction descriptor to indicate the final outcome (atomic update from *undecided* to *successful* or *failed*).

Release phase Finally, on success, each updated object has its data-block pointer updated to reference the shadow copy, while read-only objects have their original data pointers replaced.

Note that this algorithm slightly complicates finding the current data block of an object: when an object is opened we may have to search within a transaction descriptor to find the data-block pointer. For clarity, in pseudocode I use is_stm_desc(p) to determine whether the given pointer *p* is a reference to a transaction descriptor. As with MCAS, reading from an acquired object header does not need to involve recursive helping: the current logical version of the object can be determined from the contents of the transaction descriptor.

The main drawback of this algorithm is that read-only operations are implemented by acquiring and releasing objects within the STM. This may cause unnecessary conflicts between transactions, both at the software level when multiple transactions attempt to acquire a read-only object, and at the hardware level as the object headers are 'ping-ponged' between processor caches. Many data structures, particularly those used for searching, have a single root which is the 'entry point' for all operations on the structure. If care is not taken then an STM implementation of this type of structure will suffer a performance bottleneck at the object containing the entry point, from which all transactions must read.

I therefore modify the algorithm to only acquire objects that are on the transaction's read-write list (i.e., that were opened for update). This is followed by a *read phase* which compares the current data-block pointer of each object in the transaction's read-only list with the version that was seen when the object was first opened. If all pointers match then the transaction's status may be updated to indicate success, otherwise the transaction must fail. Note that if the

[1] It is typically most efficient to use arithmetic ordering of object references.

read phase sees an object header that is currently owned by another transaction then it will search within the owner's descriptor rather than helping the owner to complete (in fact, the latter approach may cause a recursive loop).

Unfortunately, incorporating a read phase creates a further problem: a successful transaction with non-empty acquire and read phases may not appear to commit atomically. This occurs when an object is updated after it is checked in the read phase, but before the transaction's status is updated to indicate success and its updates become visible to other transactions. More concretely, consider the following sequence of events which concerns two transactions, T_1 and T_2 (note that x_i denotes version i of object x):

1. T_1 opens x_i for reading, and y_j for writing.
2. T_2 opens y_j for reading, and x_i for writing.
3. T_1 acquires object y, then passes its read phase.
4. T_2 acquires object x, then passes its read phase (finds y_j in T_1's descriptor).
5. T_1 commits successfully, updating object y to y_{j+1}.
6. T_2 commits successfully, updating object x to x_{i+1}.

These transactions are not serialisable since T_1 ought to see T_2's update or vice versa. The inconsistency creeps in during step 5: T_1 invalidates T_2's read-check of object y, but T_2 is oblivious of the update and 'successfully' commits anyway.

This problem is handled by introducing two further changes:

- A new transaction status value, *read-checking*. This status value is observed only during a transaction's read phase.
- A transaction atomically commits or aborts when the descriptor status changes to *read-checking*.

The second modification may appear perverse: how can a transaction commit its changes before it has finished checking its assumptions? The key insight is that it doesn't matter that the final outcome is undecided if no other transaction will attempt to read an object header that is owned by a *read-checking* transaction T. I can arrange this by causing readers to do one of the following: (i) wait for T to reach its decision point, (ii) help T to reach its decision point, or (iii) abort T. The second option seems the best choice for a lock-free design, as the first option may stall indefinitely while careless use of abort can lead to livelock, which also invalidates the lock-free property.

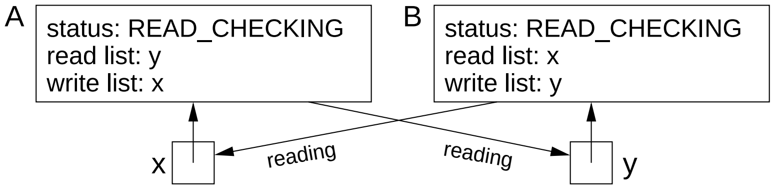

Figure 3.8: An example of a dependent cycle of two transactions, A and B. Each needs the other to exit its read phase before it can complete its own.

There is one final problem: what happens if there is a cycle of transactions, all in their read phase and each trying to read an object that is currently owned by the next? The simple example in Figure 3.8 shows that the algorithm enters a recursive loop because no transaction can progress until the next in the cycle has completed its read phase. The solution is to abort at least one of the transactions; however, care must be taken not to abort them all or livelock may occur as each transaction is continually retried and aborted. I ensure this by imposing a total order $\prec$ on all transactions, based on the machine address of each transaction's descriptor. The loop is broken by allowing a transaction T_1 to abort a transaction T_2 if and only if: (i) both are in their read phase; (ii) T_2 owns a location that T_1 is attempting to read; and (iii) $T_1 \prec T_2$. This guarantees that every cycle will be broken, but the 'least' transaction in the cycle will continue to execute.

3.3.2.4 *Pseudocode*

Figure 3.9 presents pseudocode for the open_for_writing and commit_transaction operations. Both operations use obj_read to find the most recent data block for a given object reference; I therefore describe this sub-operation first. In most circumstances the latest data-block reference can be returned directly from the object header (lines 7 and 18). If the object is currently owned by a committing transaction then the correct reference is found by searching the owner's read-write list (line 10) and selecting the old or new reference based on the owner's current status (line 16). If the owner is in its read phase then it must be helped to completion or aborted, depending on the status of the transaction that invoked its obj_read and its ordering relative to the owner (lines 11–15).

open_for_writing proceeds by checking whether the object is already open; if so, the existing shadow copy is returned (lines 21–22). If the object is present on the read-only list then the matching handle is removed (line 24). If the object is present on neither list then a new object handle allocated and initialised (lines 26–27). A shadow copy of the data block is made (lines 29–30) and the object handle is inserted into the read-write list (line 31).

```
typedef struct { word *data; } stm_obj;
typedef struct { stm_obj *obj; word *old, *new; } obj_handle;
typedef struct { word status;
                 obj_handle_list read_list, write_list;
                 int blk_size; } stm_tx;
static word *obj_read (stm_tx *t, stm_obj *o) {
  word *data := o→data;
  if ( is_stm_desc(data) ) {
    stm_tx *other := (stm_tx *)data;
    obj_handle *hnd := search(o, other→write_list);
    if ( other→status = READ_PHASE )
      if ( (t→status ≠ READ_PHASE) ∨ (t > other) )
        commit_transaction(other);
      else
        CAS(&other→status, READ_PHASE, FAILED);
    data := (other→status = SUCCESSFUL) ? hnd→new : hnd→old;
  }
  return data;
}
word *open_for_writing (stm_tx *t, stm_obj *o) {
  obj_handle *hnd := search(o, t→write_list);
  if ( hnd ≠ NULL ) return hnd→new;
  if ( (hnd := search(o, t→read_list)) ≠ NULL ) {
    remove(o, t→read_list);
  } else {
    hnd := new obj_handle;
    (hnd→obj, hnd→old) := (o, obj_read(t, o));
  }
  hnd→new := new bytes(t→blk_size);
  memcpy(hnd→new, hnd→old, t→blk_size);
  insert(hnd, t→write_list);
  return hnd→new;
}
bool commit_transaction (stm_tx *t) {
  word data, status, desired_status := FAILED;
  obj_handle *hnd, *ohnd;
  stm_tx *other;
  for ( hnd in t→write_list ) /* Acquire phase */
    while ( (data := CAS(&hnd→obj→data, hnd→old, t)) ≠ hnd→old ) {
      if ( data = t ) break;
      if ( ¬is_stm_desc(data) ) goto decision_point;
      commit_transaction((stm_tx *)data);
    }
  CAS(&t→status, UNDECIDED, READ_PHASE);
  for ( hnd in t→read_list ) /* Read phase */
    if ( (data := obj_read(t, hnd→obj)) ≠ hnd→old ) goto decision_point;
  desired_status := SUCCESSFUL;
decision_point:
  while ( ((status := t→status) ≠ FAILED) ∧ (status ≠ SUCCESSFUL) )
    CAS(&t→status, status, desired_status);
  for ( hnd in t→write_list ) /* Release phase */
    CAS(&hnd→obj→data, t, status = SUCCESSFUL ? hnd→new : hnd→old);
  return (status = SUCCESSFUL);
}
```

Figure 3.9: FSTM's open_for_writing and commit_transaction interface calls. Algorithms for *read* and *read-write* lists are not given here. Instead, search, insert, remove and for-in iterator operations are assumed to exist. These operations may store object information in a linked list, for example.

As I described in the previous section, commit_transaction is divided into three phases. The first phase attempts to acquire each object in the read-write list (lines 38–43). If a more recent data-block reference is found then the transaction is failed (line 41). If the object is owned by another transaction then the obstruction is helped to completion (line 42). The second phase checks that each object in the read-only list has not been updated since it was opened (lines 45–46). If all objects were successfully acquired or checked then the transaction will attempt to commit successfully (lines 49–50). Finally, each acquired object is released (lines 51–52); the data-block reference is returned to its previous value if the transaction failed, otherwise it is updated to its new value.

The following proof sketch demonstrates that commit_transaction, as shown in Figure 3.9, avoids both recursive loops and unbounded retry-abort cycles:

Definition 1. System-wide progress occurs whenever a transaction completes successfully.

Definition 2. A transaction T_1 in its read phase will abort a conflicting transaction T_2 also in its read phase if and only if $T_1 \prec T_2$. Otherwise T_1 will help T_2. $\prec$ is a well-founded total order on incomplete transactions.

Definition 3. $\sqsubseteq$ is the reflexive transitive closure of the 'aborts' relation: T_1 aborts $T_2 \iff T_1 \sqsubseteq T_2$.

Definition 4. S is the set of all transactions created during the lifetime of the system.

Lemma 1. *A transaction can only fail if system-wide progress has been made since the transaction began.* The transaction observed that a location changed during its execution. This can occur only as the result of a successful transaction, and thus system-wide progress (definition 1).

Lemma 2. *A recursive helping loop must contain at least one transaction in its read phase.* There cannot be a loop with only transactions in their write phase since a transaction gains ownership of locations in a global total order.

Lemma 3. *A recursive helping chain consists of two parts: a sequence of transactions in their write phase, followed by a sequence of transactions in their read phase.* A transaction in its read phase never helps a transaction in its write phase.

Lemma 4. *A recursive helping loop consists only of transactions in their read phase.* The loop contains at least one transaction in its read phase (lemma 2). This transaction will help another transaction only if it is also in its read phase.

Lemma 5. *Any sequence of recursive-helped transactions in their write phase has finite length.* First note that no such sequence can loop (lemma 2). Now, because each transaction gains ownership of its updated blocks in a global total order, no sequence can be greater than the number of objects in the system.

Lemma 6. *Any sequence of recursive-helped transactions in their read phase has finite length.* No such sequence can loop (definition 2). Furthermore, if T_1 helps T_2 in the chain then $T_2 \prec T_1$. But $\prec$ is well-founded so any sequence of helping must be bounded.

Lemma 7. *A process can recursively help only to a finite depth.* First I show that such a process can never enter a recursive loop. If this were possible then the loop must consist entirely of transactions in their read phase (lemma 4). But such a loop is impossible (lemma 6). Furthermore, recall that any recursive chain consists of two distinct sequences (lemma 3). Each of these sequences is finite in length (lemmas 5 and 6).

Lemma 8. *If T_1 aborts T_2 then T_1 either progresses to check the next location specified in its read phase, or fails.* Observe that obj_read contains no loops, and the read phase of commit_transaction immediately fails or checks the next location when obj_read returns (see Figure 3.9).

Lemma 9. *A transaction can abort only a finite number of other transactions.* A transaction can access only a finite number of objects in a system with bounded memory capacity. Assume a transaction T accesses N objects: it may therefore check up to N object headers during its read phase. Since every abort causes T to check the next pointer in its list or fail (lemma 8), a maximum of N transactions will be aborted.

Lemma 10. *$\sqsubseteq$ is a partial order.* Reflexivity and transitivity follow trivially from definition 3. Asymmetry follows from definitions 2 and 3:

$$T_1 \sqsubseteq T_2 \Rightarrow T_1 \prec T_2 \Rightarrow T_2 \not\prec T_1 \Rightarrow T_2 \not\sqsubseteq T_1 \quad (T_1 \neq T_2)$$

Lemma 11. *$\sqsubseteq$ is well-founded.* Note that $T_1 \sqsubseteq T_2 \Rightarrow T_1 \prec T_2$. Result follows from definition 2.

Lemma 12. *System-wide progress occurs within a finite number of aborts.* Consider an incomplete transaction T_1. If T_1 is aborted by T_2 then $T_2 \sqsubseteq T_1$. Thus, by lemma 11, there cannot be an endless chain of aborted transactions; a finite number of aborts must reach a minimal transaction in the poset $(S, \sqsubseteq)$. This minimal transaction cannot continually abort other transactions (lemma 9), so it must either succeed or fail, or help another transaction to succeed or fail (lemma 7 disallows endless helping), within a finite number of steps. Success and failure both imply global progress (definition 1 and lemma 1).

Theorem. *Commit is lock-free.* Let us consider the progress of a single process X. This is sufficient because, over a large enough sequence of system-wide steps, at least one process in the finite system-wide set must execute an arbitrarily large number of steps and, I assume, attempt an unbounded number of transactions. Whenever X executes a transactional commit then that transaction must progress to completion, progress a recursively-helped transaction to completion, abort some transaction, or be itself aborted, within a finite number of steps (note that lemma 7 disallows endless recursive helping, and all other code paths in commit_transaction lead directly to one of the preceding cases). If a transaction is successfully or unsuccessfully brought to completion then system-wide progress has occurred (definition 1 and lemma 1). If some transaction was aborted, then reapply the above argument a finite number of times to achieve system-wide progress (lemma 12).

3.3.3 Further enhancements

In this section I address some of the limitations of FSTM as described so far. The extensions described here should make it easier to use transactions in real-world applications. Briefly, these extensions deal with allowing multiple sizes of object, nested transactions, early release of opened objects, and automatically restarting inconsistent transactions.

3.3.3.1 *Arbitrary object sizes*

The basic FSTM design allows only a single fixed size of object per transactional memory. This makes it a poor fit for applications which consist of heterogeneous collections of objects, or data structures with variable-sized nodes (e.g., skip lists). The obvious simple fix, which creates a transactional memory specifying a single 'large enough' object size, is very wasteful of memory if most objects are small. Fortunately FSTM can easily be modified to handle arbitrary object sizes within the same transactional memory, at the cost of increased per-object overhead: (*i*) extend new_object to accept a *size* argument; (*ii*) extend object headers to contain a *size* field; and (*iii*) modify data-block allocations, deallocations, and copy operations to use the new *size* field or argument.

3.3.3.2 *Nested transactions*

In hierarchically-structured programs it would be useful to allow a sub-operation executed within a transaction to internally use transactions to commit shared-memory updates. However, the basic FSTM design disallows this style of programming because transactions cannot be nested. One solution is for sub-operations to accept an in-progress transaction handle as a parameter and execute shared-memory operations with reference to this ‘outer’ transaction. This manual fix complicates interfaces between application components and limits portability (for example, it is difficult to support legacy code written before the sub-operation interfaces were changed).

Allowing nested transactions raises questions about the precise semantics that should be supported: “When should nested transactions linearise?”, “Should a failing inner transaction cause outer transactions to fail?”, and so on. Here I limit myself to outlining a simple semantics which permits a straightforward extension to FSTM. In particular, nested transactions whose updates are committed appear to linearise at the same instant as the outermost transaction. This allows a design in which a completed inner transaction is merged with the smallest-enclosing transaction, thus ‘storing up’ the commit for later.

The basic idea is to maintain a list of in-progress transactions for each process, in nested order. When an inner transaction completes, its open objects are merged with the immediately enclosing transaction. When the outermost transaction completes, the updates of all merged transactions are validated and committed. When merging two transactions, if an object is open in both but with different data-block versions then the inner transaction is failed if its version is out-of-date; otherwise the merge is aborted and the outer transaction will eventually fail because its version is out-of-date. After merging, objects opened by the completed inner transaction are considered part of the outer transaction. This is a safe but conservative design choice: if an object opened by a completed inner transaction becomes out-of-date before the enclosing transaction attempts to commit then both transactions will have been retried even if the outer transaction has not directly accessed a stale object.

3.3.3.3 *Early release*

Herlihy *et al.* introduced the concept of an *early-release* operation, and provided a design limited to objects opened for read-only access [Herlihy03b]. This makes it possible to reduce conflicts between concurrent transactions by *releasing* an opened object before committing the transaction, if the programmer determines that it is safe to do so. A released object is removed from the transactional set; the transaction may then complete successfully even if the object is subsequently updated before the transaction attempts to commit. Unfortunately Herlihy *et al.*’s STM design, which acquires a transactional object as soon as it is opened for writing, limits the benefit that could be obtained by extending the operation to writeable objects.

Since FSTM does not acquire any objects until a transaction attempts to commit, it is easy to implement a fuller version of early release that will work for any opened object. Releasing an object with pending updates provides a number of performance benefits: not only does it reduce the possibility of the transaction failing, but it also obviates the need to acquire the object during commit and avoids the possibility of failing other concurrent transactions that have accessed the object. The red-black tree design in the next chapter provides an example where early release offers a significant performance benefit.

3.3.3.4 *Automatic validation*

A non-obvious but in practice rather serious complication arises when dealing with transactions which become inconsistent at some point during their execution. An inconsistent transaction cannot successfully commit: any attempt to validate or commit the transaction will fail and the transaction must then be restarted by the application. The problem is that an application making decisions based on inconsistent data may not get as far as attempting validation or commit. In practice there are two ways in which inconsistent data can prevent progress: the application may crash, or it may loop indefinitely.

An application which suffers these problems can be modified to validate the current transaction in appropriate places. This requires validation checks to be inserted immediately before critical operations which may cause the application to crash, and inside loops for which termination depends on transactional data. A failed validation causes the application to abort the current transaction and reattempt it, thus averting program failure or unbounded looping.

My experience when implementing transactional red-black trees was that determining where to add these validation checks in a complex algorithm is tedious and error-prone. I further observed that validation checks were only required in two types of situation: (*i*) to avoid a memory-protection fault, usually due to dereferencing a `NULL` pointer; and (*ii*) to prevent indefinite execution of a loop containing at least one FSTM operation per iteration. I therefore extended FSTM to automatically detect consistency problems in these cases.

Firstly, when a transaction is started FSTM saves enough state to automatically return control to that point if the transaction becomes invalid: in a C/UNIX environment this can be done portably using the POSIX `setjmp` and `longjmp` routines.

Secondly, a handler is installed which catches memory-protection faults and validates the in-progress transaction, if any. If the validation fails then the transaction is restarted, otherwise control is passed to the next handler in turn. When there is no other handler to receive control, FSTM uninstalls itself and re-executes the faulting instruction to obtain the system's default behaviour.

Finally, each STM operation checks the consistency of one object currently opened for access by the in-progress transaction. This avoids unbounded looping because the inconsistency will eventually be detected and the transaction automatically restarted. If necessary, the overhead of incremental validation can be reduced by probabilistically validating an open object during each STM operation. The probability of validation can be reduced to gain faster execution of STM operations at the expense of slower detection of inconsistencies. However, in my implementation of FSTM I found that very little overhead is added by checking a previously-opened object on every invocation of an STM operation; this ensures that looping transactions are detected swiftly.

3.4 Summary

In this chapter I have introduced the first lock-free programming abstractions that are practical for general-purpose use. As I will show in the next chapter, these abstractions are much simpler to use than the single-word primitives usually provided by hardware. Furthermore, the performance results I present in Chapter 6 show that the performance overhead compared with direct use of atomic hardware instructions is negligible when shared memory is moderately contended, and that performance frequently surpasses that of lock-based designs.

Chapter 4

Search structures

4.1 Introduction

Search structures are a key data-storage component in many computer systems. Furthermore, when an indexed data store is shared between multiple processors it is usually important that the data can be efficiently accessed and updated without excessive synchronisation overheads. Although lock-based search structures are often suitable, lock-free solutions have traditionally been considered wherever reentrancy (e.g., within a signal or interrupt handler), process failure, or priority inversion is a concern.

In this chapter I extend the argument for lock-free data structures by showing that they can also be *faster* and *easier to implement* compared with using mutual exclusion. As discussed in the Introduction, locks do not scale well to large parallel systems. Structures based on fine-grained locking suffer a number of problems:

- The cost of acquiring and then releasing a large number of small locks may be unreasonable, particularly when access to the structure is not heavily contended and a coarser-granularity scheme would suffice.
- Even when acquiring a read-only lock, updates in the acquire and release operations may cause memory coherency traffic.
- Efficiently avoiding problems such as deadlock can require tortuous programming; for example, to ensure that locks are acquired in some global order.

By avoiding these weaknesses, the lock-free data structures in this chapter achieve highly competitive performance and reduced complexity, despite relying on the seemingly elaborate MCAS and STM designs proposed in Chapter 3. The performance gains are largely due to eliminating lock contention and improving cache locality. This is particularly beneficial for tree structures: if read operations must acquire locks then the root of the tree can become a significant bottleneck, even if multi-reader locks are used. Improvements in code complexity can partly be attributed to the lack of convoluted 'lock juggling'. Also, more subjectively, MCAS and STM seem to fit well with the way we reason about concurrency problems, in terms of regions of atomically-executed code rather than regions of atomically-accessed data.

I present here lock-free designs for three well-known search structures: skip lists, binary search trees (BSTs), and red-black trees. As I discuss in the next chapter, I have implemented and tested these structures on a wide range of modern shared-memory multiprocessor architectures, and the experimental results show that my designs generally perform better than high-quality lock-based schemes.

4.2 Functional mappings

Each of the designs in this chapter can be viewed as implementing a functional mapping from a domain of keys to a range of values. All that is required is that the domain forms a totally-ordered set: that is, any pair of distinct keys has a relative ordering over $<$.

In the following description I represent a function as a set of key-value pairs. Each key is represented at most once in a set; any key which does not appear maps to some distinguished value, $\perp$. The abstract datatype supports *lookup*, *update* and *remove* operations, all of which take a set S and a key k and return the current mapping of k in S:

$$op(S,k) = \begin{cases} v & \text{if } \exists v \,.\, (k,v) \in S \\ \perp & \text{otherwise} \end{cases}$$
$$op \in \{\mathit{lookup}, \mathit{update}, \mathit{remove}\}$$

In addition, *update* and *remove* modify the set in place:

$$\mathit{remove}(S_i,k): \quad S_{i+1} = \begin{cases} S_i \setminus \{(k,v)\} & \text{if } \exists v \,.\, (k,v) \in S \\ S_i & \text{otherwise} \end{cases}$$

$$\mathit{update}(S_i,k,w): \quad S_{i+1} = \begin{cases} S_i \setminus \{(k,v)\} \cup \{(k,w)\} & \text{if } \exists v \,.\, (k,v) \in S \\ S_i \cup \{(k,w)\} & \text{otherwise} \end{cases}$$

In the following pseudocode I assume that key values are integers of type map-

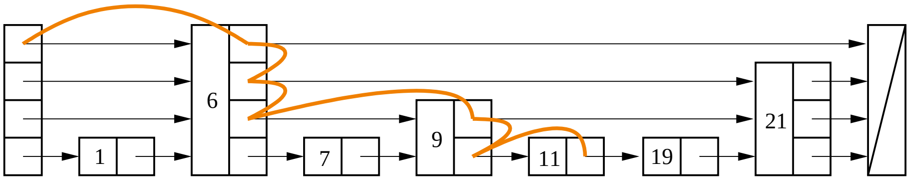

Figure 4.1: Searching in a skip list. This example illustrates the path taken when searching for the node with key 11.

key_t. Mapped values are pointers of type mapval_t, with ⊥ represented by NULL.

4.3 Skip lists

Skip lists are probabilistic search structures which provide improved execution-time bounds compared with straightforward binary search trees yet are much simpler to implement than any guaranteed-$O(\log n)$ search structure [Pugh90b]. A skip list comprises multiple levels, each of which is a linked list. Every skip-list node is present at the lowest level, and *probabilistically* present in each higher level up to some *maximum* level that is chosen independently and randomly for each node. This maximum is selected using a random number generator with exponential bias: for example, the probability of inserting into level x is often chosen to be 2^{-x}. In my pseudocode designs I use a function rand_level to assign a maximum level to a new node. Figure 4.1 shows how a node can be found efficiently by using higher levels of the skip list to quickly 'home in' on the area of interest.

A particularly useful property for parallel skip-list designs is that a node can be independently inserted at each level in the list. A node is visible as long as it is linked into the lowest level of the list: insertion at higher levels is necessary only to maintain the property that search time is $O(\log n)$. Pugh used this insight to design an efficient highly-parallel skip list implementation based on per-pointer locks [Pugh90a] which significantly influenced my own designs presented here.

I present lock-free designs built from three different atomic primitives with the aim of demonstrating the tradeoff between simplicity and, in the next chapter, efficiency. The FSTM-based design is so straightforward that a compiler could generate it automatically from the sequential algorithm. In contrast, the design which uses CAS directly is considerably more complicated but can be expected to execute faster. The MCAS-based design is a middle ground between these two extremes.

```
(stm_obj **, stm_obj **, node_t *) list_search (stm_tx *tx, list_t *l, mapkey_t k) {
    (x_obj, x) := (l→head, open_for_reading(tx, l→head));
    for ( i := NR_LEVELS-1; i ≥ 0; i-- ) {
        while ( TRUE ) {
            (y_obj, y) := (x→next[i], open_for_reading(tx, x→next[i]));
            if ( y→k ≥ k ) break;
            (x_obj, x) := (y_obj, y);
        }
        (left_objlist[i], right_objlist[i]) := (x_obj, y_obj);
    }
    return (left_objlist, right_objlist, y);
}
mapval_t list_lookup (list_t *l, mapkey_t k) {
    do {
        tx := new_transaction(l→memory);
        (_, _, succ) := list_search(tx, l, k);
        v := (succ→k = k) ? succ→v : NULL;
    } while ( ¬commit_transaction(tx) );
    return v;
}
mapval_t list_update (list_t *l, mapkey_t k, mapval_t v) {
    (new_obj, new) := new_object(l→memory);
    (new→level, new→k, new→v) := (rand_level(), k, v);
    do {
        tx := new_transaction(l→memory);
        (pred_objs, succ_objs, succ) := list_search(tx, l, k);
        if ( succ→k = k ) { /* Update value field of an existing node. */
            succ := open_for_writing(tx, succ_objs[0]);
            (old_v, succ→v) := (succ→v, v);
        } else {
            old_v := NULL;
            for ( i := 0; i < new→level; i++ ) {
                pred := open_for_writing(tx, pred_objs[i]);
                (pred→next[i], new→next[i]) := (new_obj, succ_objs[i]);
            }
        }
    } while ( ¬commit_transaction(tx) );
    if ( old_v ≠ NULL ) free_object(new_obj);
    return old_v;
}
mapval_t list_remove (list_t *l, mapkey_t k) {
    do {
        tx := new_transaction(l→memory);
        (pred_objs, succ_objs, succ) := list_search(tx, l, k);
        old_v := NULL;
        if ( succ→k = k ) {
            old_v := succ→v;
            for ( i := 0; i < succ→level; i++ ) {
                pred := open_for_writing(tx, pred_objs[i]);
                pred→next[i] := succ→next[i];
            }
        }
    } while ( ¬commit_transaction(tx) );
    if ( old_v ≠ NULL ) free_object(succ_objs[0]);
    return old_v;
}
```

Figure 4.2: Skip lists built from FSTM.

4.3.1 FSTM-based design

Skip lists can be built straightforwardly from FSTM by representing each list node as a separate transactional object (Figure 4.2). Each list operation is implemented by encapsulating the sequential algorithm within a transaction, and opening each node before directly accessing it. Every skip list contains a pair of sentinel nodes, respectively containing the minimal and maximal key values; this simplifies the search algorithm by eliminating code to deal with corner cases.

4.3.2 MCAS-based design

A nice feature of skip lists is that searches do not need to synchronise with carefully implemented update operations, because the entire structure can be made continuously consistent from their point of view. Pugh showed how to do this by updating the pointers in a deleted node to point *backwards*, causing searches to automatically backtrack when they follow a stale link [Pugh90a].

This technique can be used to build efficient skip lists from MCAS in a simple manner, as shown in Figure 4.3. Insertions batch all their individual memory writes and then perform a single MCAS operation (line 32), while searches check each shared-memory location that they read to ensure it is not currently 'owned' by an MCAS(line 5). Deletions invoke MCAS to update each predecessor node to point at its new successor (line 43). As described above, each pointer in a deleted node is updated to point backwards, so that searches backtrack correctly (line 44).

One case that deserves special mention is updating the mapping of a key that is already present in the list. Here I can update the value field in place, rather than deleting the node and inserting a new one. Since this is a single-word update, it is possible to use CAS directly and so avoid the overheads of MCAS (line 24). When a node is deleted its value field is set to NULL (line 46). This indicates to other operations that the node is garbage, and forces them to re-read the data structure.

4.3.3 CAS-based design

The direct-CAS design performs composite update operations using a sequence of individual CAS instructions, with no need for a dynamically-allocated per-operation 'descriptor'. This means that great care is needed to ensure that updates occur atomically and consistently. Figure 4.4 illustrates how conflicting insertions and deletions can otherwise cause inconsistencies.

```
(node_t **, node_t **) list_search (list_t *l, mapkey_t k) {
   x := &l→head;
   for ( i := NR_LEVELS-1; i ≥ 0; i-- ) {
      while ( TRUE ) {
         y := MCASRead(&x→next[i]);
         if (y→k ≥ k) break;
         x := y;
      }
      (left_list[i], right_list[i]) := (x, y);
   }
   return(left_list, right_list);
}
mapval_t list_lookup (list_t *l, mapkey_t k) {
   (_, succs) := list_search(l, k);
   return (succs[0]→k = k) ? MCASRead(&succs[0]→v) : NULL;
}
mapval_t list_update (list_t *l, mapkey_t k, mapval_t v) {
   new := new node_t;
   (new→level, new→k, new→v) := (rand_level(), k, v);
   do {
      (preds, succs) := list_search(l, k);
      if ( succs[0]→k = k ) { /* Update value field of an existing node. */
         do { if ( (old_v := MCASRead(&succs[0]→v)) = NULL ) break;
         } while ( CAS(&succs[0]→v, old_v, v) ≠ old_v );
         if ( old_v = NULL ) continue;
         return old_v;
      }
      for ( i := 0; i < new→level; i++ ) { /* Construct update list. */
         new→next[i] := succs[i]; /* New node can be updated directly. */
         (ptr[i], old[i], new[i]) := (&preds[i]→next[i], succs[i], new);
      }
   } while ( ¬MCAS(new→level, ptr, old, new) );
   return NULL; /* No existing mapping was replaced. */
}
mapval_t list_remove (list_t *l, mapkey_t k) {
   do {
      (preds, succs) := list_search(l, k);
      if ( (x := succs[0])→k ≠ k ) return NULL;
      if ( (old_v := MCASRead(&x→v)) = NULL ) return NULL;
      for ( i := 0; i < succs[0]→level; i++ ) {
         x_next := MCASRead(&x→next[i]);
         if ( x→k > x_next→k ) return NULL;
         (ptr[2*i], old[2*i], new[2*i]) := (&preds[i]→next[i], x, x_next);
         (ptr[2*i+1], old[2*i+1], new[2*i+1]) := (&x→next[i], x_next, preds[i]);
      }
      (ptr[2*i], old[2*i], new[2*i]) := (&x→v, old_v, NULL);
   } while ( ¬MCAS(2*succs[0]→level+1, ptr, old, new) );
   return old_v;
}
```

Figure 4.3: Skip lists built from MCAS.

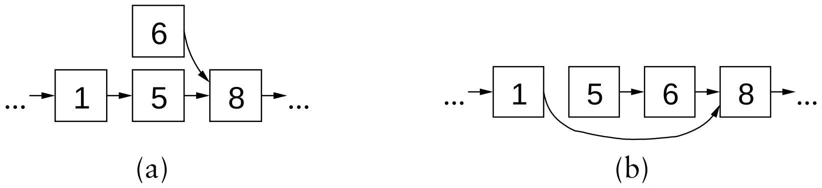

Figure 4.4: Unsynchronised insertion and deletion in a linked list. Snapshot (a) shows a new node, 6, about to be inserted after node 5. However, in snapshot (b) we see that node 5 is simultaneously being deleted by updating node 1 to point at node 8. Node 6 never becomes visible because it is linked from the defunct node 5.

4.3.3.1 *Pointer marking*

Harris solves the problem of 'disappearing' nodes, illustrated in Figure 4.4, for singly-linked lists by *marking* a node's forward pointer before physically deleting it from the list [Harris01]. This prevents concurrent operations from inserting directly after the defunct node until it has been removed from the list. Since, for the purposes of insertion and deletion, I treat each level of a skip list as an independent linked list, I use Harris's marking technique to logically delete a node from each level of the skip list in turn.

To implement this scheme I reserve a *mark bit* in each pointer field. This is easy if all list nodes are word-aligned; for example, on a 32-bit architecture this will ensure that the two least-significant bits of a node reference are always zero. Thus a low-order bit can safely be reserved, provided that it is masked off before accessing the node.

For clarity, the marking implementation is abstracted via a set of pseudocode operations which operate on pointer marks: is_marked(p) returns TRUE if the mark bit is set in pointer p, mark(p) returns p with its mark bit set, while unmark(p) returns p with its mark bit cleared (allowing access to the node that it references).

4.3.3.2 *Search*

The search algorithm searches for a *left* and a *right* node at each level in the list. These are adjacent nodes with key values respectively less-than and greater-than-or-equal-to the search key.

```
1   (node_t **, node_t **) list_search(list_t *l, mapkey_t k) {
     retry: left := &l→head;
3     for ( i := NR_LEVELS-1; i ≥ 0; i-- ) {
        left_next := left→next[i];
5       if ( is_marked(left_next) ) goto retry;
        /* Find unmarked node pair at this level. */
7       for ( right := left_next; ; right := right_next ) {
          /* Skip a sequence of marked nodes. */
9         while ( TRUE ) {
            right_next := right→next[i];
11          if ( ¬is_marked(right_next) ) break;
            right := unmark(right_next);
13        }
          if ( right→k ≥ k ) break;
15        left := right; left_next := right_next;
        }
17      /* Ensure left and right nodes are adjacent. */
        if ( (left_next ≠ right) ∧ (CAS(&left→next[i], left_next, right) ≠ left_next) )
19        goto retry;
        left_list[i] := left; right_list[i] := right;
21    }
      return(left_list, right_list);
23  }
```

The search loop interprets marked nodes and skips over them, since logically they are no longer present in the list (lines 7–16). If there is a sequence of marked nodes between a level's left and right nodes then these are removed by updating the left node to point directly at the right node (lines 17–19). Line 5 retries the entire search because if the left node at the previous level is now marked then the search result as constructed so far is now stale.

The publicly-exported lookup operation simply searches for the required key, and then returns the value mapping if a matching node exists.

```
mapval_t list_lookup(list_t *l, mapkey_t k) {
    (_, succs) := list_search(l, k);
    return (succs[0]→k = k) ? succs[0]→v : NULL;
}
```

4.3.3.3 *Deletion*

Removal begins by searching for the node with key k. If the node exists then it is *logically deleted* by updating its value field to NULL (lines 4–6). After this point any subsequent operations will see no mapping for k and, if necessary, they will remove the defunct node entirely from the list to allow their own updates to proceed. The next stage is to 'mark' each link pointer in the node; this prevents any new nodes from being inserted directly after the deleted node, and thus avoids the consistency problem in Figure 4.4. Finally, all references to the deleted node are removed. This is done by a single call to list_search, which guarantees that the node it matches is not preceded by a marked node.

```
mapval_t list_remove(list_t *l, mapkey_t k) {
    (_, succs) := list_search(l, k);
    if ( succs[0]→k ≠ k ) return NULL;
    /* 1. Node is logically deleted when the value field is NULL. */
    do { if ( (v := succs[0]→v) = NULL ) return NULL;
    } while ( CAS(&succs[0]→v, v, NULL) ≠ v );
    /* 2. Mark forward pointers, then search will remove the node. */
    mark_node_ptrs(succs[0]);
    (_, _) := list_search(l, k);
    return v;
}
```

The loop that marks a logically-deleted node is placed in a separate function so that it can be used by list_update. It simply loops on each forward pointer in turn, trying to add the mark until the mark bit is present.

```
void mark_node_ptrs(node_t *x) {
    for ( i := x→level-1; i ≥ 0; i-- )
        do {
            x_next := x→next[i];
            if (is_marked(x_next)) break;
        } while ( CAS(&x→next[i], x_next, mark(x_next)) ≠ x_next );
}
```

4.3.3.4 *Update*

An update operation searches for key k and, if it finds a matching node, it attempts to atomically update the value field to the new mapping (lines 6–15). If the matching node has a NULL value field then it is already logically deleted; after finishing pointer-marking (line 10), the update completes the physical deletion when it retries the search (line 5). If the key is not already present in the list then a new node is initialised and linked into the lowest level of the list (lines 16–18). The main loop (lines 19–32) introduces the node into higher levels of the list. Care is needed to ensure that this does not conflict with concurrent operations which may insert a new predecessor or successor at any level in the list, or delete the existing one, or even delete the node that is being inserted (lines 25–26). A new node becomes globally visible, and the insert operation linearises, when the node is inserted into the lowest level of the list. If an update modifies the value field of an existing node then that modification is the linearisation point.

```
mapval_t list_update(list_t *l, mapkey_t k, mapval_t v) {
    new := new node_t;
    (new→level, new→k, new→v) := (rand_level(), k, v);
  retry:
    (preds, succs) := list_search(l, k);
    /* Update the value field of an existing node. */
    if ( succs[0]→k = k ) {
        do {
            if ( (old_v := succs[0]→v) = NULL ) {
                mark_node_ptrs(succs[0]);
                goto retry;
            }
        } while ( CAS(&succs[0]→v, old_v, v) ≠ old_v );
        return old_v;
    }
    for (i := 0; i < new→level; i++) new→next[i] := succs[i];
    /* Node is visible once inserted at lowest level. */
    if ( CAS(&preds[0]→next[0], succs[0], new) ≠ succs[0] ) goto retry;
    for (i := 1; i < new→level; i++)
        while (TRUE) {
            pred := preds[i]; succ := succs[i];
            /* Update the forward pointer if it is stale. */
            new_next := new→next[i];
            if ( (new_next ≠ succ) ∧
                 (CAS(&new→next[i], unmark(new_next), succ) ≠ unmark(new_next)) )
                break; /* Give up if pointer is marked. */
            /* Check for old reference to a ‘k’-node. */
            if (succ→k = k) succ := unmark(succ→next);
            /* We retry the search if the CAS fails. */
            if ( CAS(&pred→next[i], succ, new) = succ ) break;
            (preds, succs) := list_search(l, k);
        }
    return NULL; /* No existing mapping was replaced. */
}
```

4.4 Binary search trees

Compared with skip lists, lock-free binary search trees (BSTs) are complicated by the problem of deleting a node with two non-empty subtrees. The classical algorithm replaces the deleted node with either the smallest node in its right

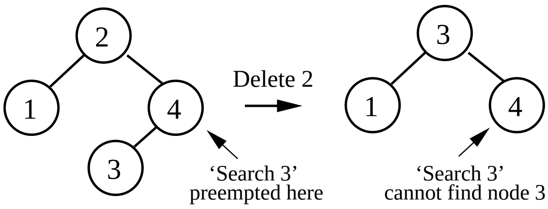

Figure 4.5: Consistency between search and delete operations.

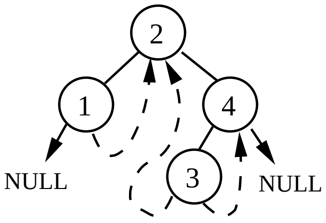

Figure 4.6: An example of a small threaded BST.

subtree or the largest node in its left subtree; these nodes can be easily removed from their current position as they have at most one subtree. Implementing this without adding extra synchronisation to search operations is difficult because it requires the replacement node to be atomically removed from its current location and inserted in place of the deleted node.

4.4.1 MCAS-based design

The problem of making an atomic update to multiple memory locations, to effect the simultaneous deletion and reinsertion, is solved by using MCAS. Although this ensures that all the updated memory locations change to their new value at the same instant in time, this is insufficient to ensure consistency of concurrent search operations. Consider the concurrent 'delete 2' and 'search 3' operations in Figure 4.5. The search is preempted when it reaches node 4, and continues only after node 2 has been deleted. However, since node 3 is relocated to replace node 2, the search will complete unsuccessfully.

This problem could be avoided entirely by using FSTM, but there are likely to be significant overheads compared with using the simpler MCAS operation. Instead I use a *threaded* tree representation [Perlis60] in which pointers to empty subtrees are instead linked to the immediate predecessor or successor node in the tree (see Figure 4.6). If this representation is applied to the example in Figure 4.5, the 'delete 2' operation will create a thread from node 4 to node 3 which acts as a tombstone. The 'search 3' operation can now be modified to follow this thread to check for the relocated node instead of immediately, and incorrectly, failing.

This neat solution permits the use of MCAS because the tree threads ensure that search operations remain synchronised.

4.4.1.1 *Thread links*

Threaded trees must provide a way to distinguish thread links from ordinary node pointers. I use a special mark value in the lowest-order bits of each link field, similar to the deletion mark added to skip list nodes in Section 4.3.3. For clarity I define some utility functions which act on node pointers: is_thread(p) returns TRUE if p is marked as a thread, thread(p) returns p with the thread mark set, and unthread(p) returns p with the thread mark cleared. These operations can be implemented efficiently in a handful of bit-level machine instructions.

Further care is needed because, as discussed in the next chapter, MCAS may also use mark bits to distinguish its descriptor pointers. MCAS reserves two bits in every updated machine word for this purpose, but needs only to distinguish between three classes of value: pointer to MCAS descriptor, pointer to CCAS descriptor, and all other values. This means that a fourth mark value is available for use in the BST representation. I provide an example of how to interpret mark bits without introducing conflicts in Section 5.1.

4.4.1.2 *Read operations*

The key field of a BST node can be read directly as the key is never modified after a node is initialized. Reads from pointer locations, including the subtree and value fields of a node, must use MCASRead in case an MCAS operation is currently in progress.

Another possibility is that a field is read after the node is deleted from the tree. I handle this by setting all the pointer fields of a deleted node to an otherwise unused value (NULL). This allows a read to detect when it has read from a defunct node and take appropriate action, such as retrying its access from the tree root.

4.4.1.3 *Search operations*

The search algorithm is encapsulated in a helper function, bst_search(t,k), which returns a tuple (p, n) consisting of the node n with key k, and its parent p. If key k is not in the tree then p is the final node on the search path, and n is the thread link which would be replaced if k were inserted into the tree.

```
(node_t *, node_t *) bst_search(tree_t *t, mapkey_t k) {
 retry:
   p := &t→root;
```

```
      n := MCASRead(&p→r);
 5    while ( ¬is_thread(n) ) {
         if ( k < n→k ) c := MCASRead(&n→l);
 7       else if ( k > n→k ) c := MCASRead(&n→r);
         else return (p, n);
 9       /* We retry if we read from a stale link. */
         if ( c = NULL ) goto retry;
11       p := n; n := c;
      }
13    /* If the thread matches, retry to find parent. */
      if ( k = unthread(n)→k ) goto retry;
15    return (p, n);
   }
```

The loop on lines 5–12 traverses the tree in the usual manner, checking for concurrent MCAS operations on the search path, and retrying from the root if the search traverses a deleted node. The test on line 14 is executed only if k was not found in the tree. In that case, the thread link found at the end of the search is followed to check if it leads to a node with key k. If so, the search must be retried because, although the required node has been found, it is not possible to find its parent without restarting from the root of the tree.

A lookup in a BST is implemented via a simple wrapper around bst_search:

```
mapval_t bst_lookup(tree_t *t, mapkey_t k) {
   (_, n) := bst_search(t, k);
   return is_thread(n) ? NULL : MCASRead(&n→v);
}
```

4.4.1.4 *Update and insertion*

There are two cases to consider when updating a *(key, value)* mapping. If bst_search finds an existing mapping for *key*, it attempts to directly modify that node's value field (line 10). If no current mapping is found then it inserts a newly-allocated node into the tree (line 21). CAS is used in both cases because only one memory location needs to be updated.

```
 1 mapval_t bst_update(tree_t *t, mapkey_t k, mapval_t v) {
      new := new node_t; new→k := k; new→v := v;
 3   retry:
      do {
 5       (p, n) := bst_search(t, k);
         if ( ¬is_thread(n) ) {
 7          do {
```

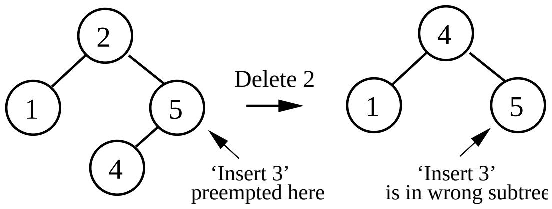

Figure 4.7: Consistency between insertion and deletion.

```
                old_v := MCASRead(&n→v);
 9              if ( old_v = NULL ) goto retry;
             } while ( CAS(&n→v, old_v, v) ≠ old_v );
11           return old_v;
          }
13        if ( p→k < k ) {
             if ( unthread(n)→k < k ) goto retry;
15           (new→l, new→r) := (thread(p), n);
          } else {
17           if ( unthread(n)→k > k ) goto retry;
             (new→l, new→r) := (n, thread(p));
19        }
       }
21     while ( CAS((p→k < k) ? &p→r : &p→l, n, new) ≠ n );
       return NULL;
23  }
```

Lines 14 and 17 deserve further comment. They are required because, if a node has been moved up the tree due to a deletion, the search may no longer have found the correct position to insert the new node. Figure 4.7 illustrates this problem more clearly: the original root node is replaced by key value 4, so a new node with key value 3 now belongs in the *left* subtree of the root.

It is instructive to note that lookups and deletions do *not* need to worry about this type of inconsistency. This may cause lookups and deletions to fail to find a node even though a matching one has been inserted in a different subtree. The failing operation is still linearisable because the inserted node must have appeared *after* the failing operation began executing. This is because the failing operation began executing before the deletion which caused the inconsistency ('delete 2' in Figure 4.7), but the insertion of the new node must be linearised after that deletion. The failing operation can therefore be linearised before the new node was inserted.

4.4.1.5 *Deletion*

Deletion is the most time-consuming operation to implement because of the number of different tree configurations which must be handled. Figure 4.8

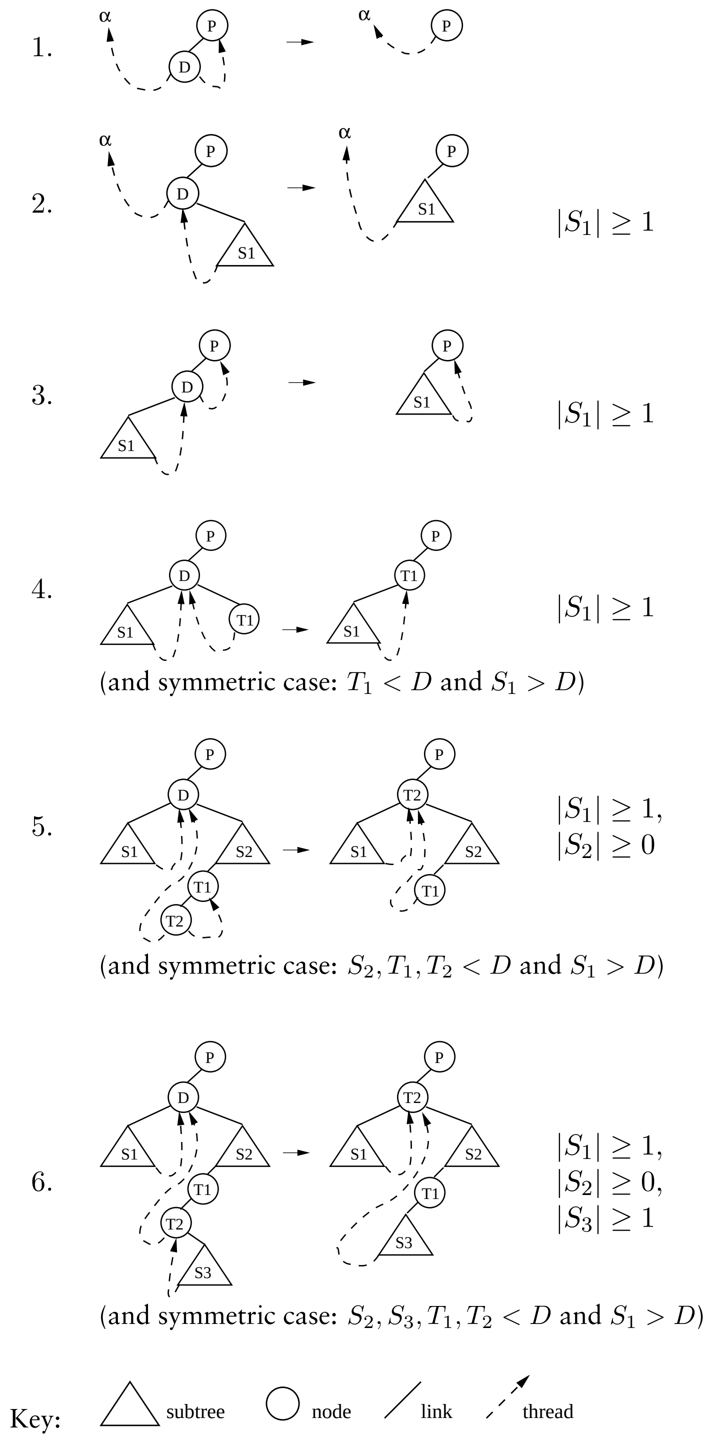

Figure 4.8: Deleting a node D from a threaded binary tree. The cases where D is the right-child of its parent P are omitted, but can be trivially inferred from the left-child transformations. Only threads that are introduced or removed by a transformation are included in its diagram. Where the target of a thread is outside the subtree in the diagram, the destination node is represented by α.

shows all the different tree configurations that deletion may have to deal with, and the correct transformation for each case. Although somewhat cumbersome, the implementation of each transformation is straightforward: traverse the tree to find the nodes involved, and retry the operation if a garbage node is traversed or if the tree structure changes "under the operation's feet". For brevity the pseudocode handles only cases 4–6 in Figure 4.8 and does not consider any symmetric cases.

```
mapval_t bst_remove(tree_t *t, mapkey_t k) {
 retry:
  (p, d) := bst_search(t, k);
  if ( is_thread(d) ) return NULL;
  /* Read contents of node: retry if node is garbage. */
  (dl, dr, dv) := (MCASRead(&d→l), MCASRead(&d→r), MCASRead(&d→v));
  if ( (dl = NULL) ∨ (dr = NULL) ∨ (dv = NULL) ) goto retry;
  if ( (p→k > d→k) ∧ ¬is_thread(dl) ∧ ¬is_thread(dr) ) {
    /* Find predecessor, and its parent (pred,ppred). */
    (pred, cpred) := (d, dl);
    while ( ¬is_thread(cpred) ) {
      (ppred, pred, cpred) := (pred, cpred, MCASRead(&pred→r));
      if ( cpred = NULL ) goto retry;
    }
    /* Find successor, and its parent (succ,psucc). */
    (succ, csucc) := (d, dr);
    while ( ¬is_thread(csucc) ) {
      (psucc, succ, csucc) := (succ, csucc, MCASRead(&succ→l));
      if ( csucc = NULL ) goto retry;
    }
    (ptr[1], old[1], new[1]) := (&d→l, dl, NULL);
    (ptr[2], old[2], new[2]) := (&d→r, dr, NULL);
    (ptr[3], old[3], new[3]) := (&d→v, dv, NULL);
    (ptr[4], old[4], new[4]) := (&succ→l, thread(d), dl);
    (ptr[5], old[5], new[5]) := (&p→l, d, succ);
    (ptr[6], old[6], new[6]) := (&pred→r, thread(d), thread(succ));
    if ( succ = dr ) { /* Case 4, Fig. 4.8. */
      if ( ¬MCAS(6, ptr, old, new) ) goto retry;
    } else {          /* Cases 5 - 6, Fig. 4.8. */
      succ_r := MCASRead(&succ→r);
      (ptr[7], old[7], new[7]) := (&succ→r, succ_r, dr);
      (ptr[8], old[8], new[8]) := (&psucc→l, succ,
            is_thread(succ_r) ? thread(succ) : succ_r);
      if ( ¬MCAS(8, ptr, old, new) ) goto retry;
    }
  /* All symmetric and simpler cases omitted. */
  } else if ( ... ) ...
  return dv;
}
```

4.4.1.6 *Consistency of threaded binary search trees*

It is not obvious that a threaded representation ensures that concurrent search operations will see a consistent view of the tree. I therefore sketch a proof of correctness which demonstrates that the representation is sufficient, by reference to the tree transformations in Figure 4.8. This proof does not demonstrate that the operations discussed so far are a correct implementation of a BST: it is intended only to provide a convincing argument that nodes moved upwards in a BST remain continuously visible to concurrent search operations.

Lemma 1. *Threads to a node D can be safely removed when D is deleted.* A search treats a deleted node as no longer existing in the tree. Thus a search can safely complete without finding D, and any thread to D is redundant.

Lemma 2. *Threads from a node D can be moved to any node at or below that node's parent P when D is deleted.* If a concurrent search has yet to reach D, it will either: (i) follow the old link from P, find D deleted, and retry its operation; or (ii) it will follow the new link or thread from P and thus will ultimately follow the thread from its new location.

Lemma 3. *A thread from a node D to its parent P can be safely removed when D is deleted.* If a concurrent search has not reached D then it will find P first anyway. If the search has reached D, it will either: (i) follow the old thread $D \rightarrow P$; or (ii) it will detect that D is deleted and thus retry.

Theorem. *The threaded representation ensures that search operations complete correctly after any concurrent deletion.* The threaded representation is sufficient if, whenever a node is relocated, a thread is created at the end of the search path for that node. Furthermore, that thread must remain in existence until the node is deleted. I analyse each case in Figure 4.8 to prove that any search concurrent with the deletion of some node D will find the correct node:

Case 1. Thread $D \rightarrow P$ is safely removed (lemma 3). $D \rightarrow \alpha$ is safely relocated to P (lemma 2).

Case 2. Thread $D \rightarrow \alpha$ is safely moved into subtree $S1$ (lemma 2).

Case 3. Thread $D \rightarrow P$ is safely removed (lemma 3).

Case 4. Node $T1$ is relocated. However any concurrent thread can still find $T1$ directly, without following a thread. If a search reaches D it will either: (i) find $T1$ by following the old link from D; or (ii) detect that D is deleted, retry the search, and find $T1$ via the new link from P.

Case 5. Thread $T2 \rightarrow T1$ is removed. An argument similar to that in lemma 3

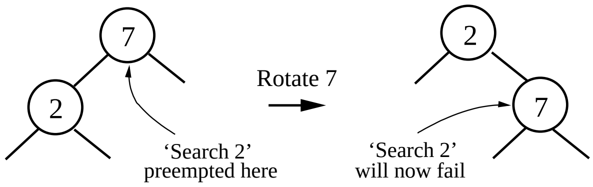

Figure 4.9: Searches must be correctly synchronised with concurrent tree rotations, in case the search path becomes invalid.

suffices to show that this is safe. Node $T2$ is relocated, but any search can still find it. If the search has not reached D it will either: (i) find $T2$ at its new location, or (ii) find D is deleted and retry. If the search has passed D it will follow the new thread from $T1$ to find $T2$.

Case 6. The thread $S3 \rightarrow T2$ remains valid after the deletion. Although node $T2$ is relocated, the argument in case 5 applies: a concurrent search will always find $T2$ by following the new link from P, finding D deleted, or following the existing thread.

4.5 Red-black trees

Unlike skip lists and unbalanced BSTs, red-black tree operations are guaranteed to execute in $O(\log n)$ time. As might be expected, this performance guarantee comes at the cost of increased algorithmic complexity which makes red-black trees an ideal case study for applying lock-free techniques in a practical, non-trivial application.

4.5.1 FSTM-based design

Since red-black trees make extensive use of rotation transformations to ensure that the structure remains balanced, search operations must be careful to remain synchronised with update processes. For this reason it would be very difficult to build red-black trees using CAS or MCAS: as shown in Figure 4.9, some additional technique would be required to synchronise search operations.

Despite this complexity, red-black trees can be implemented straightforwardly using FSTM. Each operation (lookup, update, and remove) begins by starting a new transaction. Each tree node is represented by a separate transactional object, so nodes must be opened for the appropriate type of access as the tree is traversed. Each operation finishes by attempting to commit its transaction: if this fails then the operation is retried.

An interesting design feature is the use of FSTM's *early-release* operation (Section 3.3.3). A common trick when implementing red-black trees is to replace NULL child pointers with references to a single 'sentinel node' [Cormen90]. By colouring this node black it is possible to avoid a considerable amount of special-case code that is otherwise required to correctly rebalance leaf nodes. In a transaction-based design, however, the sentinel node can easily become a performance bottleneck. Insertions and deletions are serialised because they all attempt to update the sentinel node's parent pointer. This problem is neatly avoided by explicitly releasing the sentinel node before attempting to commit such transactions.

Since full pseudocode for each of the red-black tree operations would run to many pages, I do not list the full design here. However, like the FSTM-based skip-list design, each operation is a straightforward adaptation of the textbook sequential design. For example, lookup proceeds as follows:

```
mapval_t list_lookup (list_t *l, mapkey_t k) {
  do {
    tx := new_transaction(l→memory);
    v := NULL;
    for ( nb := l→root; nb ≠ SENTINEL; nb := (k < n→k) ? n→l : n→r ) {
      n := open_for_reading(tx, nb);
      if ( k = n→k ) { v := n→v; break; }
    }
  } while ( ¬commit_transaction(tx) );
  return v;
}
```

Note that leaf nodes are distinguished by comparing against SENTINEL. This is the global 'sentinel node' that is used in place of NULL to avoid many special cases when rebalancing the tree. Updates and insertions perform an *early release* (Section 3.3.3.3) of SENTINEL before attempting to commit, to avoid unnecessary update conflicts.

4.5.2 Lock-based designs

Unlike skip lists and simple BSTs, there has been little practical work on parallelism in balanced trees. The complexity of even single-threaded implementations suggests that implementing a lock-based version which permits useful amounts of parallelism is likely to be very difficult. Reducing this complexity was one of the motivations for developing skip lists, which permit a simple yet highly-concurrent implementation [Pugh90a].

Due to the lack of previous work in this area, I discuss two possible lock-based red-black tree designs; in Chapter 6 I use these as a baseline against which to compare my STM-based design. The first design is simple but serialises all operations which update the tree. The second design relaxes this constraint to allow greater parallelism, but is significantly more complicated.

4.5.2.1 *Serialised writers*

Ellis presents two locking strategies for AVL trees, another search-tree design which uses rotation transformations to maintain balance [Ellis80]. Both locking strategies depend on a complicated protocol for mutual exclusion in which locks can be acquired in a number of different modes. The second strategy appears to achieve some useful parallelism from simultaneous update requests, but the implementation is extremely complicated. For example, the synchronisation protocol allows a tree node to be locked or marked in five different ways, yet the implementation of this intricate mechanism is not described.

Hanke describes how the simpler of Ellis's two locking protocols can be directly applied to red-black trees [Hanke99]. The protocol allows a node to be locked by a process in one of three ways: by acquiring an x-lock (exclusive lock), a w-lock (write-intention lock), or an r-lock (read lock). A node can be r-locked by multiple processes simultaneously, but only one process at a time may hold an x- or w-lock. Furthermore, a w-lock can be held simultaneously with r-locks, but an x-lock excludes all other processes.

Using these locks, a lookup operation proceeds by acquiring r-locks as it proceeds down the tree. By using *lock coupling* at most two nodes need to be locked at any time: each node's lock is held only until the child's lock is acquired.

Update and removal operations w-lock the whole search path from the root of the tree so that rebalancing can safely occur after a node is inserted or deleted. If rebalancing is required then any affected node is upgraded to an x-lock. Acquiring w-locks in the first instance ensures that other operations cannot invalidate the update's view of the tree.

I took this protocol as a starting point for my own initial red-black tree design. First I note that all update and removal operations are effectively serialised because they all acquire and hold a w-lock on the root of the tree for their duration. A simpler yet equivalent approach is to do away with w-locks entirely and instead have a single mutual-exclusion lock which is acquired by every update or removal operation. With w-locks no longer required, x- and r- locks map directly onto the operations supported by standard multi-reader locks.

In my scheme, lookup operations still proceed by read-coupling down the tree.

Update and removal operations do not need to acquire read locks because other writers are excluded by the global mutual-exclusion lock. If an operation needs to modify nodes in the tree, perhaps to implement a rebalancing rotation, then the subtree to be modified is write-locked. Write locks are acquired down the tree, in the same order as lookup operations acquire read locks: this avoids the possibility of deadlock with concurrent searches.

4.5.2.2 *Concurrent writers*

The scheme I outline above can be implemented as a simple modification of a non-concurrent red-black tree design. Unfortunately the single writer lock means that it will achieve very little parallelism on workloads which require a non-negligible number of updates and removals.

Consider again the scheme in which updates and removals w-lock their entire search path. This limits concurrency because these operations become serialised at the root of the tree. The only reason that the whole path is locked is because, after a node is inserted or deleted, rebalancing operations *might* be required all the way back up the tree to the root. Unfortunately, until the rebalancing rotations are executed we do not know how many rotations will be required; it is therefore impossible to know in advance how many nodes on the search path actually *need* to be write-locked.

One superficially attractive solution is to read-lock down the tree and then write-lock on the way back up, just as far as rebalancing operations are required. This scheme would acquire exclusive access to the minimal number of nodes (those that are actually modified), but can result in deadlock with search operations (which are locking *down* the tree).

The problem is that an exclusive lock must be continuously held on any *imbalanced* node, until the imbalance is rectified. Otherwise other update operations can modify the node without realising it is imbalanced, and irreparably upset the red-black properties of the tree. Unfortunately the rotation transformations that rebalance the tree all require the imbalanced node's parent to be updated, and therefore locked — there is therefore no obvious way to avoid acquiring locks *up* the tree.

A neat solution to this quandary is to *mark* nodes that are imbalanced. There is then no need to continuously hold an exclusive lock on an imbalanced node, so long as the update algorithms are revised to take the imbalance mark into account. Fortunately the required revisions are already implemented for *relaxed* red-black trees [Hanke97]. Relaxed data structures decouple insertions and deletions from the transformations required to rebalance the structure. An update which creates an imbalance marks the appropriate node and queues work for a maintenance process which will perform the appropriate rebalance transform some time later. Insertions and deletions in a relaxed red-black tree are simple because it uses an *external* representation in which key-value pairs are stored only in leaf nodes; internal nodes are simply 'routers' for search operations. The rebalance transformations for a relaxed red-black tree include the usual node rotations and recolourings, but three more transformations are included which deal with conflict situations in which the transformations required by two imbalanced nodes overlap. Without these extra transformations the two rebalance operations would deadlock.

My fine-grained scheme borrows the external tree representation and extended set of tree transformations used by relaxed red-black trees. I apply a simple locking protocol in which all operations read-couple down the tree. When an update or removal reaches a matching node, or the closest-matching leaf node, this node is write-locked pending insertion, deletion, or value update. If an insertion or deletion causes an imbalance then the imbalanced node is marked, and all held locks are released. Rather than leaving the rebalancing work for a maintenance process, the update operation then applies the transformations itself.

Each local transformation is performed separately, and is responsible for acquiring and releasing all the locks that it requires. The first stage of a transformation is to traverse the local subtree to work out which transformation must be applied. This initial phase is executed with no locks held: if it detects an obvious inconsistency then it will abort and retry the subtree search. When the appropriate transformation has been selected, the subtree is write-locked, starting with the node nearest the root. Before each node is locked it is checked that the structure of the tree has not changed since the search phase: if it has then the transformation is aborted and retried. Once all required locks are held, the transformation is applied, the mark is propagated up the tree or removed entirely, and then all locks are released.

4.6 Summary

In this chapter I have presented highly-concurrent designs for three popular types of search structure. I introduced three lock-free skip-list designs based on CAS, MCAS, and FSTM: this allows a fair comparison between the three primitives in Chapter 6. I also presented an MCAS-based design for binary search trees: CAS is too difficult to apply directly in this case and, as I will show later, FSTM is an inferior choice when MCAS is a suitable alternative. Finally, I presented an FSTM-based design for red-black trees: as with BSTs, CAS is too hard to use directly; MCAS is also unsuitable because red-black trees require synchronisation between readers and writers.

In the next chapter I show how to turn these search-structure designs, and the underlying programming abstractions, into portable implementations for real hardware.

Chapter 5

Implementation issues

There is a considerable gap between the pseudocode I presented in the previous chapters and a useful implementation of those algorithms. I bridge this in the following sections by tackling three main implementation challenges: distinguishing 'operation descriptors' from other memory values (Section 5.1), reclamation of dynamically-allocated memory (Section 5.2) and memory-access ordering on architectures with relaxed memory-consistency models (Section 5.3).

I describe how these issues are resolved in my C implementation of the pseudocode algorithms, resulting in a portable library of lock-free abstractions and structures for Alpha, Intel IA-32, Intel IA-64, MIPS, PowerPC and SPARC processor families. Support for other architectures can easily be added by providing an interface to the required hardware-level primitives, such as memory barriers and the CAS instruction.

5.1 Descriptor identification

To allow implementation of the *is-a-descriptor* predicates from Chapter 3, there needs to be a way to distinguish MCAS, CCAS, and FSTM descriptors from other valid memory values. There are a number of techniques that might be applied.

If the programming language's run-time system retains type information then this may be sufficient to distinguish descriptor references from other types of value. This is likely to limit CCAS and MCAS to operate only on pointer-typed locations, as dynamically distinguishing a descriptor reference from an integer with the same representation is not generally possible. However, FSTM descriptors are installed only in place of data-block pointers, so FSTM trivially complies with this restriction.

In the absence of a typed run-time environment, an alternative approach is for the storage manager to maintain a list of allocated descriptors. The appropriate list can then be searched to implement each descriptor predicate. Note that this approach also restricts CCAS and MCAS to pointer-typed locations, to prevent confusion between descriptor references and identically-represented integers. The cost of searching the descriptor lists is likely to be impractical if more than a handful of descriptors are allocated. The search time can be reduced by allocating descriptors from a small set of contiguous *pools*. This shorter pool list is then sufficient to distinguish descriptor references, and can be searched more quickly.

The approach taken in my own implementations is to reserve the least-significant two bits of any location which may hold a reference to a descriptor. This reservation is easy if descriptor references are placed only in locations that otherwise contain word-aligned pointers. On a 32-bit system, for example, aligned references are always a multiple of four and the least-significant two bits are guaranteed to be zero. This approach also requires descriptors to be aligned in memory, so that the low-order bits of a descriptor reference can safely be reserved for identification; non-zero settings of these bits are used to distinguish the various types of descriptor from other heap values. The reserved bits are masked off before accessing a descriptor via a reference that was identified in this way.

Special care is needed to prevent clashing with the mark used to represent thread references in the MCAS implementation of binary search trees (Section 4.4). This can be achieved by assigning the following non-conflicting meanings to the two reserved bits:

Reserved bits	Interpretation
00	Ordinary heap reference
01	MCAS descriptor reference (Section 3.2)
10	CCAS descriptor reference (Section 3.2)
11	FSTM descriptor reference (Section 3.3)
11	CAS-based skip list: deleted node (Section 4.3.3)
11	MCAS-based BST: thread reference (Section 4.4)

By itself this technique is insufficient if further identifiers need to be allocated without reserving more bits. One possible extension is to reserve a tag field at a common offset in every type of descriptor. This would allow the different types of descriptor to share one reference identifier, freeing two identifiers for other purposes. The different descriptors can still be distinguished by reading the common tag field.

5.2 Storage management

So far it has been assumed that a run-time garbage collector will automatically reclaim dynamically-allocated memory that is no longer in use. However, there are a number of reasons for considering a customised scheme. Firstly, many run-time environments do not provide automatic garbage collection, so some alternative must be sought. Secondly, the garbage collectors found in general-purpose programming environments often do not scale well to highly-parallel workloads running on large multiprocessor systems. Thirdly, general-purpose collectors may not be designed to efficiently handle a very high rate of heap allocations and garbage creation, but this type of workload is likely to be created by FSTM (for example) which allocates a new version of an object every time it is updated. Finally, general-purpose collectors with a "stop the world" phase cannot provide the strong progress guarantees that lock-free applications may require: systems which absolutely require this guarantee must provide their own lock-free memory manager.

I use several schemes for managing the different types of dynamic object in my lock-free algorithms. In each case the selected scheme strikes a balance between the costs incurred by mutator processes, the rate and cost of garbage collection (where applicable), and the time taken to return garbage objects to an appropriate free list (slow reclamation increases the size of the heap and reduces locality of memory accesses). The chosen schemes are described and justified in the following subsections: object aggregation (5.2.1), reference counting (5.2.2), and epoch-based reclamation (5.2.3).

5.2.1 Object aggregation

Although the pseudocode design assumes so for simplicity, CCAS descriptors are not dynamically allocated. Instead, several are embedded within each MCAS descriptor, forming an *aggregate*. Embedding a small number of CCAS descriptors within each MCAS descriptor is sufficient because each one can be immediately reused as long as it is introduced to any particular memory location at most once. This restriction is satisfied by allocating a single CCAS descriptor to each process that participates in an MCAS operation; each process then reuses its descriptor for each of the CCAS sub-operations that it executes. Unless contention is very high it is unlikely that recursive helping will occur often, and so the average number of processes participating in a single MCAS operation will be very small.

If excessive helping does ever exhaust the embedded cache of CCAS descriptors then further allocation requests must be satisfied by dynamic allocation. These dynamically-allocated descriptors are managed by the same reference-counting mechanism as MCAS and FSTM descriptors.

The same storage method is used for the per-transaction object lists maintained by FSTM. Each transaction descriptor contains a pool of embedded object handles that are sequentially allocated as required. If a transaction opens a very large number of objects then further descriptors are allocated and chained together to extend the node pool.

Object aggregation is best suited to objects whose lifetimes are correlated since an aggregate cannot be reclaimed until all embedded objects are no longer in use. Although some space may be wasted by aggregating too many objects, this is generally not a problem if the embedded objects are small. In the cases of CCAS descriptors and FSTM object handles, the space overhead is far outweighed by the lack of need for dynamic storage management. Embedded objects are allocated sequentially within an aggregate, and are not reclaimed or reused except as part of the aggregate. Thus there is negligible cost associated with management of embedded objects.

5.2.2 Reference counting

Each MCAS and FSTM descriptor contains a reference count which indicates how many processes currently hold a reference to it. I use the method described by Michael and Scott to determine when it is safe to reuse a descriptor [Michael95]. This avoids the possibility of reclaiming a descriptor multiple times, by reserving a bit in each reference count which is set the first time that a descriptor is reclaimed and cleared when it is reused. The bit must not be cleared until the reference count is incremented by the operation that is reusing it. This prevents a delayed process from incrementing and then decrementing the reference count from and back to zero, which would result in the descriptor being reclaimed from under the feet of the new operation.

A descriptor's reference count does not need to be adjusted to include every shared reference. Instead, each process that acts on an operation descriptor increments the reference count just once. The process is then responsible for ensuring that all the shared references it introduces on behalf of the operation are removed before decrementing the descriptor's reference count.

Note that memory used to hold reference-counted descriptors cannot be reused for other types of dynamically-allocated object, nor can it be returned to the operating system. This is because at any time in the future a process with a stale reference to a defunct descriptor may attempt to modify its reference count. This will be disastrous if that memory location has been allocated a completely different purpose or is no longer accessible by the process. This problem is addressed by Greenwald and Cheriton's *type-stable memory*, which uses an out-of-band scheme, such as a stop-the-world tracing garbage collector, to determine when no such stale references exist [Greenwald96]. I do not consider retasking of descriptor memory in my implementation because a small number of descriptors proved sufficient to satisfy all dynamic allocation requests.

Reference counting was chosen for MCAS and FSTM descriptors for two reasons: (*i*) they are large, because they are aggregates containing embedded objects; and (*ii*) they are ephemeral, since they do not usually persist beyond the end of the operation that they describe. Since the descriptors are large it is important that they are reused as quickly as possible to prevent the heap exploding in size to accommodate defunct-but-unusable descriptors. Reference counting satisfies this requirement: a descriptor can be reused as soon as the last reference to it is relinquished. This is in contrast to techniques such as Michael's SMR [Michael02], in which garbage objects may be buffered for a considerable time to amortise the cost of a garbage-collection phase. The main criticism of reference counting is that reference-count manipulations can become a performance bottleneck. However, since operation descriptors are short-lived and only temporarily installed at a small number of memory locations, it is unlikely that many processes will access a particular descriptor and therefore need to manipulate its reference count.

5.2.3 Epoch-based reclamation

Apart from operation descriptors, all other dynamic objects (including search-structure nodes, FSTM object headers and FSTM data blocks) are reclaimed by an *epoch-based* garbage collector. The scheme builds on 'limbo lists' [Kung80, Manber84,Pugh90a,Arcangeli03] which hold a garbage object until no stale references can possibly exist. However, I deviate from previous designs to improve cache locality and efficiently determine when stale references cannot exist.

This style of garbage collection requires that, when an object is no longer referenced from the shared heap, it is explicitly added to the current garbage list. It is generally very simple to augment pseudocode with the required garbage-list operations: for example, operations that successfully delete a node from a search structure are then solely responsible for placing that node on the list. Only the CAS-based skip-list design raises a significant complication. In this case, a node may be deleted while it is still being inserted at higher levels in the list. If this occurs then the delete operation cannot place the node on the garbage list, since new shared references may still be created. This problem is solved by *deferring* responsibility to the operation that completes last. Insertions and deletions both attempt to set a per-node deferral flag: whichever operation observes that the flag is already set is responsible for placing the node on the garbage list. A single boolean flag is sufficient because only two operations may place a node on the garbage list: the operation that inserted the node, and the delete operation that logically deleted the node by setting its value field to NULL. Other processes that help the deletion do not attempt to free the node and so need not be considered.

Note that an object can be added to the current limbo list only when there are no more references to it in shared memory, and no new shared references will be created. If this restriction is correctly applied then the only references that can exist for a given limbo object are: (*i*) private, and (*ii*) held by processes which started their current operation *before* the object was 'put in limbo'.

This property allows me to use a global *epoch count* to determine when no stale references exist to any object in a limbo list. Each time a process starts an operation in which it will access shared memory objects, it *observes* the current epoch. When all processes have observed the current epoch, the limbo list that was populated two epochs ago can safely be reclaimed. This now-empty list can be immediately recycled and populated with garbage nodes in the new epoch; thus only three limbo lists are ever needed.

It is not immediately obvious that once all processes have observed the current epoch, the list populated during the previous epoch cannot be immediately reclaimed. Note, however, that not all processes observe a new epoch at the same time. Thus *two* limbo lists are being populated with garbage objects at any point in time: the list associated with the current epoch (which processes are moving to) and the list associated with the previous epoch (which processes are moving from). Processes that have observed epoch e may therefore still hold private references to objects in the limbo list associated with epoch $e - 1$, so it is not safe to reuse those objects until epoch $e + 1$.

Whenever a process starts a shared-memory operation it probabilistically scans a process list to determine whether all processes that are currently executing within a critical region have seen the current epoch[1]. If so, the process prepends the contents of the oldest limbo list to the free list and then increments the epoch count. This scheme avoids the need for a maintenance process to perform reclamation, and attempts to distribute the workload of garbage collection among all processes.

[1] Excluding processes not executing within a critical region ensures that quiescent processes do not obstruct garbage collection.

Although limbo lists are accessed using lock-free operations, and garbage collection does not interfere with other mutator processes, this reclamation scheme is *not* strictly lock-free. For example, a process which stalls for any reason during a shared-memory operation will not observe updates to the epoch count. In this situation the limbo lists will never be reclaimed and memory cannot be reused. Other processes can make progress only until the application reaches its memory limit. This drawback may also affect preemptively-scheduled systems, in which a process may be descheduled in the middle of a shared-memory operation with no guarantee when it will be rescheduled.

In situations where this limitation is unreasonable, an alternative and truly lock-free scheme should be used. Unfortunately both SMR [Michael02] and pass-the-buck [Herlihy02] incur extra overheads for mutator processes. Each time a new object reference is traversed it must be *announced* before it is dereferenced. On all modern architectures this requires a costly memory barrier to ensure that the announcement is immediately made visible to other processors. To give some indication of the overhead that these schemes incur, adding the memory barriers that would be required in my lock-free BST algorithm increased execution time by over 20%. Furthermore, freeing an object using pass-the-buck is expensive because a global array of per-process pointers must be checked to ensure no private references remain. SMR amortises the array scan over a suitably large number of defunct objects, but this delays reuse and may harm cache locality and increase the heap size. A more efficient scheme which provides a weaker progress guarantee is likely to be preferable where that is sufficient.

5.3 Relaxed memory-consistency models

In common with most published implementations of lock-free data structures, my pseudocode designs assume that the underlying memory operations are *sequentially consistent* [Lamport79]. As defined by Lamport, a multiprocessor system is sequentially consistent if and only if "the result of any execution is the same as if the operation of all the processes were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program". If this property holds then each process appears to execute instructions in program order, memory operations appear to occur atomically and instantaneously, and operations from different processors may be arbitrarily interleaved.

Unfortunately no contemporary multiprocessor architecture provides sequential consistency. As described by Adve and Gharachorloo, consistency guarantees are instead *relaxed* to some degree to permit high-performance microarchitectural features such as out-of-order execution, write buffers, and write-back caches [Adve96].

Relaxed consistency models are much easier to describe using a formalised terminology. I therefore define an *execution order* to be a total ordering, represented by $<$, of memory accesses by all processors that are participating in some execution of a parallel application, as deduced by an external observer (note that not all the memory accesses may directly result in processor-external interactions due to architectural features such as caches and write buffers). I also define two useful partial orders over the memory accesses of a parallel program: $A <_m B$ if and only if $A < B$ in all valid execution orders; and $A <_p B$ if and only if A and B are executed on the same processor, and A occurs before B in the instruction sequence. Intuitively, $<_m$ expresses the guarantees provided by the memory consistency model while $<_p$ represents 'program order'.

5.3.1 Minimal consistency guarantees

Although processor architectures relax their memory-consistency models to varying extents, nearly all architectures provide the following minimal set of consistency guarantees:

Coherency Writes to individual memory words are globally serialised, there is only ever one up-to-date version of each memory word, and this latest version is eventually visible to all processors in the system (in the absence of further writes).

Self consistency If A and B access the same memory word, and $A <_p B$, then $A <_m B$. Informally, accesses from the same processor to the same location are seen by all processors to occur in program order. Some processor architectures, including SPARC v9 and Intel IA-64, may violate self consistency when both A and B are reads [Weaver94, Intel03].

Dependency consistency $A <_m B$ if A and B are executed on the same processor, and (i) B depends on a control decision influenced by A (such as a conditional jump), and B is a write; or (ii) B depends on state written by A, such as a machine register. The Alpha architecture violates dependency consistency [DEC92].

Further to these basic guarantees, many architectures provide additional and stronger guarantees, although these generally fall short of the requirements for sequential consistency. Examples of further guarantees provided by well-known multiprocessor architectures include:

Intel IA-32 (P6, P4) If $A <_p B$, and B is a write, then $A <_m B$. This prevents memory operations from being delayed beyond any later write.

SPARC (Total Store Order) If $A <_p B$, and A is a read or B is a write, then $A <_m B$. This is slightly stronger than IA-32 consistency because it also prevents reordering of reads.

SPARC (Partial Store Order) If $A <_p B$, and A is a read, then $A <_m B$. This prevents reads from being delayed beyond any later memory operation.

5.3.2 Memory barriers

When a required ordering is not implicitly guaranteed by the memory model, it can be established using *barrier* instructions. All memory operations before a barrier must commit (become globally visible) before any later operation may be executed.

Barrier instructions are often provided which affect only certain classes of memory access, such as read operations. These weaker forms can be used to improve performance when they are a safe replacement for a full barrier. The most common forms of weak barrier are *read barriers* and *write barriers*, which respectively affect only the ordering of reads and writes.

The most common use of barrier instructions is in the implementation of locking primitives. A mutual-exclusion lock must ensure that memory accesses within the critical region occur only while the lock is held; this usually requires inclusion of a barrier instruction in both *acquire* and *release*. Since the lock implementation is responsible for coping with possible memory-access reordering, applications do not need to pay special attention to relaxed consistency models provided that all shared data is protected by a lock.

5.3.3 Inducing required orderings

Since the pseudocode in this dissertation assumes a sequentially-consistent memory model, the algorithms cannot be directly implemented for most modern processor architectures. As described above, *memory barriers* must first be added to enforce ordering between shared-memory operations where that is required for correctness.

Unfortunately there is no automatic method for determining the optimal placement of memory barriers. At the very least this would require a formal definition of correctness for each shared-memory operation that is being implemented; a demonstration of *why* the pseudocode implementation satisfies the correctness condition on sequentially-consistent hardware is also likely to be required. Given this information, it may be possible to devise an algorithm that accurately determines which pairs of memory accesses *need* barriers between them when the memory model is relaxed. In the absence of an automated method, the accepted technique is to determine manually, and in an *ad hoc* fashion, where barriers need to be placed. I base these decisions on analysis of the pseudocode; essentially applying an informal version of the analysis method described above. This is backed up with extensive testing on real hardware, the results of which are checked off-line using a formal model of correctness. I describe this method of testing in greater detail in the next section.

A simple illustration of this analysis is provided by the implementation of MCAS, which consists of two phases with a decision point between them. Successful MCAS operations linearise (must appear to atomically execute) when their status field is updated to *successful*, during execution of the decision point. This update must occur after all writes in the first phase of the algorithm; other processes might otherwise see the MCAS descriptor in some locations, and therefore see the new value, but still see the old value in other locations. A similar argument can be applied to writes in the second phase: updates which replace the descriptor reference with the new value must be applied *after* the decision point, otherwise locations which still contain the descriptor reference will appear to still contain their old value. On some processor architectures, such as Alpha and SPARC, *write barriers* are required immediately before and after the decision point to guarantee these two ordering constraints. These barriers guarantee that all previous memory writes are visible to other processors before any later writes are executed.

A further complication for an implementation which must run on multiple processor architectures is that each requires different barrier placements. The approach I have taken is to determine where barriers would be required by an architecture which provides the minimal consistency guarantees described in Section 5.3.1, and insert *barrier functions* into the implementation as necessary. For each supported architecture I then map the barrier functions to the required machine instruction, or to a *no-operation* if the architecture implicitly guarantees the order. For example, Intel IA-32 guarantees that memory writes will commit in-order: explicit write barriers are therefore not required.

Even when the processor does not require an explicit barrier instruction, a hint must sometimes be provided to the compiler so that it does not reorder critical memory accesses, or cache shared-memory values in local registers, across memory barriers. In my C-based implementations, barriers are implemented as assembly fragments that are not analysed by the compiler. Most C compilers will therefore conservatively assume that shared memory is accessed and updated by the barrier[1], even when it is empty or a *no-op* instruction. One exception is the GNU compiler, *gcc*, which needs to be told explicitly what is accessed or updated by an assembly fragment. This requires `memory` to be added to the list of 'clobbered' values for each of the barrier fragments, so that memory accesses are not reordered across the barrier, and all cached shared-memory values are invalidated.

5.3.4 Very relaxed consistency models

Finally, it is worth mentioning the complications that arise if a processor architecture does not guarantee the assumed minimal consistency guarantees. A real-world example is provided by the Alpha 21264, which does *not* guarantee dependency consistency [DEC92]. It is not immediately obvious how an implementation of the 21264 might violate this guarantee: how can a memory access be executed before the instruction which computes the access address, for example? One possibility is that *value speculation* is used to guess the access address before it is computed: this allows the access to be executed early, and validated later when the address is known with certainty. Although no current processor implements value speculation, in practise the 21264 still violates dependency consistency because of its relaxed cache-coherency protocol [Alp00]. When a cache line is requested for exclusive access, an *invalidate request* is broadcast to other processors so that they will throw away stale versions of the line. Most processors will act on the invalidation request before sending an acknowledgement back to the originating processor; however, the 21264 relaxes this requirement by allowing a processor to acknowledge the broadcast as soon as it enters its request queue. The originating processor can therefore update the cache line, and execute past write barriers, while stale versions of the line still exist!

This relaxation of consistency means that certain lock-free programming idioms cannot be straightforwardly implemented for the 21264. Consider allocating, initialising and inserting a new node into a linked list. On most processors, the only explicit synchronisation that is required is a write barrier immediately before the new node is inserted. Since other processes must read a reference to the new node before they can access its contents, such accesses are *data-dependent* on the read which finds the reference. The single write barrier executed by the inserting process is therefore sufficient to ensure that any process that observes the new node will see correctly-initialised data. This is not the case for the 21264, however: unless a read barrier is executed by the reading process it may see stale

[1] The same assumption is made for function calls to other object files, which also cannot be analysed by the compiler.

data when it accesses the contents of the node. One solution which I considered for my search-structure implementations is to execute a read barrier each time a node is traversed. Unfortunately this significantly reduces performance — by around 25% in my experiments. Instead, I pre-initialise heap memory with an otherwise-unused *garbage* value, and ensure that this is visible to all processors before I allow the memory to be allocated. Each time an operation reads from a node it checks the value read against the garbage value and, if it matches, a read barrier is executed before repeating the access. This vastly reduces the number of executed barrier instructions, but at the cost of extra programming complexity and specialisation for a particular type of processor.

Note that lock-based applications do not need special attention to execute correctly on the 21264, provided that shared data is only ever accessed after the appropriate lock is acquired. This even applies to highly-concurrent designs in which the locks themselves are dynamically allocated. Consider a version of the linked-list example which uses per-node locks: if a lock is acquired before any other field is accessed then that is sufficient to ensure that no stale data relating to the node exists in the cache. This is because the lock field is accessed using a read-modify-write instruction which gains exclusive access to the cache line and handles any pending invalidate request for that line. Furthermore, as described in Section 5.3.2, the *acquire* operation will contain a barrier instruction which ensures that memory accesses within the critical region will not act on stale data. Locking also removes the need for the write barrier before insertion: an adequate barrier will be executed by the lock's *release* operation.

5.4 Summary

I used the techniques described in this section to implement a portable library of lock-free abstractions and structures for Alpha, Intel IA-32, Intel IA-64, MIPS, PowerPC and SPARC processor families. Most of the library is implemented in portable C code. The epoch-based garbage collector is implemented as an independent module that is linked with each data-structure implementation. Reference counting, object aggregation and descriptor identification are implemented directly within the MCAS and FSTM modules: this allows some degree of code specialisation to tailor the implementation to its particular use.

Architecture-specific issues are abstracted by a set of macro definitions in a per-architecture header file. Each header file encapsulates all the non-portable aspects of the implementation, and exports them via a set of uniformly-named macros:

Macro declaration	Comments
`MB()`	Full memory barrier
`RMB()`	Read memory barrier
`WMB()`	Write memory barrier
`CAS(a, e, n)`	CAS, returning previous contents of `a`

These macros are implemented in the form of small pieces of inline assembly code. Each memory-barrier macro is implemented by a single machine instruction; the sole exception is `WMB` on Intel IA-32, which requires no implementation because the architecture commits all memory writes in program order. The `CAS` macro is implemented using a hardware CAS instruction where that is available; on Alpha, MIPS and PowerPC I instead use a loop based on *load-linked* and *store-conditional*.

These macro definitions are sufficient to reconcile the non-portable aspects of almost all the various supported processor architectures. Alpha is the one exception which, as discussed in the previous section, needs special care to handle the possible reordering of dependent memory accesses. For example, I ensure that allocated objects are initialised to a known value that is visible to all processes, the garbage collector has an extra epoch delay before reusing defunct objects, during which time they are initialised to zero. By the time these objects are reused the epoch will have changed and all processes will have started at least one new critical section, and will thus have executed the necessary memory barrier. Note that the Alpha architecture was retired several years ago by Compaq, and is now almost at the end of its life. No other processor architecture relaxes the ordering between dependent memory accesses.

In the next chapter I present an evaluation of my lock-free implementations on a modern SPARC multiprocessor system. However, the performance on other architectures is very similar since the implementation differences are small. Performance variations are expected across different execution platforms, even within the same processor family, due to differences in CPU speed and memory subsystem.

Chapter 6

Evaluation

In the previous chapter I discussed how the abstract pseudocode presented in Chapters 3 and 4 could be turned into practical implementations for a range of modern processor architectures. In this chapter I discuss how I validated the correctness of the resulting library of lock-free algorithms by processing operation-invocation logs using an off-line model checker. I then present a performance evaluation of the previously-discussed search structure designs on a modern large-scale SPARC multiprocessor system. Since many of these structures are built using the MCAS and FSTM designs from Chapter 3, my evaluation also demonstrates the effectiveness of these primitives, and allows a comparison between them when they are used to implement 'real world' data structures.

6.1 Correctness evaluation

The C implementations of my lock-free algorithms are sufficiently complex that some form of testing is required to ensure that they are free of errors. Testing the final implementations, rather than relying on analysis of the underlying algorithms, has the advantage that it encompasses aspects of the system that are not considered in abstract pseudocode. This is particularly important for lock-free algorithms for which issues such as placement of memory barriers can allow subtle synchronisation bugs to creep in.

Run-time testing is, of course, limited to validating only the executions that happen to occur during a finite set of test runs. However, other methods of validating algorithms have their own weaknesses. Manual proofs of correctness are very popular in the literature but tend to be complex and difficult to check; there is also a considerable danger of making subtle invalid assumptions in a complex proof. For example, while implementing a search tree design by Kung and Lehman, which is accompanied by a 'proof of correctness' [Kung80], I was hindered by a bug in their deletion algorithm — this highlights the danger of substituting manual proof for implementation and testing. Automatic model checkers are usually based on a search algorithm that can check only limited test cases, or that requires significant simplification of the algorithm being checked to produce a finite-state model.

The approach I take here is to log information about the operation invocations executed by a number of parallel processes running a pseudo-random workload. The log contains an entry for each operation invocation executed by each process. Each entry specifies the operation that was invoked, its parameters, its final result, and a pair of system-wide timestamps taken when the invocation began and when it completed.

This log is processed by an off-line model checker which searches for a linearised execution of the invocations that: (*i*) follows the requirement that an operation appears to atomically occur at some point during the time that it executes, and (*ii*) obeys the semantics of the abstract data type on which the operations act. Condition (*i*) requires that any invocation B which begins executing after some invocation A completes must be placed after A in the linearised execution; otherwise B might appear to execute before it is invoked, or A might appear to execute after it completes. Condition (*ii*) simply means that the result of each invocation in the serialised execution must match the result of simulating the execution on the abstract data type.

Wing and Gong prove that finding such a schedule for an unconstrained parallel execution is NP-complete [Wing93]. In the absence of a faster solution I use a greedy algorithm which executes a depth-first search to determine a satisfactory ordering for the invocations. Each step of the search selects an operation invocation from the set S of those not already on the search path. A valid selection must be consistent with the current state of the abstract data type and must have a start timestamp smaller than the earliest completion timestamp in S. These conditions ensure that abstract state remains consistent and that time ordering is conserved. When an operation invocation is added to the search path, the abstract state is modified as appropriate.

Since I need to check only the search structure designs described in Chapter 4, I make two simplifications to the checking algorithm. Firstly, checking abstract state is simple: the state of the set datatype, as described in Section 4.2, is represented by an array of *values*, indexed by *key*. Emulating or validating an operation invocation requires a single array access. Furthermore, since the only valid operations are *lookup*, *update* and *remove*, each of which depends only on the current state of a single key, operations which act on different keys can be linearised independently. This allows much larger logs to be processed within a reasonable time: for a log of size N describing operations on a set with maximum key K, the search algorithm is applied to logs of expected size N/K. Since the expected execution time of the depth-first search is super-linear, this is much faster than directly validating a log of size N. Furthermore, the search algorithm can be applied to each key in parallel, making it practical to validate longer test runs on a multiprocessor system.

The more executions of an implementation that are tested, the more confident we can be that it contains no errors. A non-linearisable execution is firm evidence that the implementation, and perhaps the original algorithm, is incorrect; furthermore, it often gives some indication of where the bug lies. Since the off-line scheduler performs a worst-case exponential-time search, the most effective way to test many operation invocations is to execute lots of very small test runs. I tested each implementation for an hour on an UltraSPARC-based four-processor Sun Fire V480 server. The log file from each run was copied to a separate machine which ran the off-line checker. It is my experience that an hour's testing by this method is sufficient to find even very subtle bugs. Incorrect implementations always created an invalid log within a few minutes; conversely, I never found a bug in any implementation that was successfully tested for more than half an hour.

6.2 Performance evaluation

All experiments were run on a Sun Fire 15K server populated with 106 UltraSPARC III processors, each running at 1.2GHz. The server comprises 18 CPU/memory boards, each of which contains four processors and several gigabytes of memory. The boards are plugged into a backplane that permits communication via a high-speed crossbar interconnect. A further 34 processors reside on 17 smaller CPU-only boards.

I submitted benchmark runs to a 96-processor dispatch queue. I limited experiments to a maximum of 90 processes to ensure that each process could be bound to a unique physical processor with minimal risk of migration between processors or preemption in favour of system tasks.

Each experiment is specified by three adjustable parameters:

S	—	The search structure, or *set*, that is being tested
P	—	The number of parallel processes accessing the set
K	—	The average number of unique key values in the set

The benchmark program begins by creating P processes and an initial set, implemented by S, containing the keys 0, 2, 4, $\ldots$, $2K$. All processes then enter a tight loop which they execute for 10 wall-clock seconds. On each iteration they randomly select whether to execute a lookup ($p = 75\%$), update ($p = 12.5\%$), or remove ($p = 12.5\%$). This distribution is chosen because reads dominate writes in many observed real workloads; it is also very similar to the distributions used in previous evaluations of parallel algorithms [Mellor-Crummey91b, Shalev03]. When 10 seconds have elapsed, each process records its total number of completed operations. These totals are summed and used to calculate the *result* of the experiment: the mean number of CPU-microseconds required to execute a random operation.

I chose a wall-clock execution time of 10 seconds because this is sufficient to amortise the overheads associated with warming each processor's data caches, and starting and stopping the benchmark loop. Running the benchmark loop for longer than 10 seconds does not measurably affect the final result.

6.2.1 Alternative lock-based implementations

To provide a meaningful baseline for evaluation of the various lock-free data structures, I have implemented a range of alternative lock-based designs. In this section I briefly describe each of the designs that I implemented, including several that improve on the previous best-known algorithm. I indicate beside each design, in bold face, the name by which I refer to it in the results section. I conclude this section by describing the scalable lock implementations on which I build the lock-based designs.

6.2.1.1 *Skip lists*

Per-pointer locks
Pugh describes a highly-concurrent skip list implementation which uses per-pointer mutual-exclusion locks [Pugh90a]. Any update to a pointer must be protected by its lock. As discussed in Chapter 4, deleted nodes have their pointers updated to link *backwards* thus ensuring that a search correctly backtracks if it traverses into a defunct node.

Per-node locks
Although per-pointer locking successfully limits the possibility of conflicting processes, the overhead of acquiring and releasing so many locks is an important consideration. I therefore also implemented Pugh's design using per-node locks. The operations are identical to those for per-pointer locks, except that a node's lock is acquired before it is first updated and continuously held until after the final update to the node. Although this slightly increases the possibility of con-

flict between processes, in many cases this is more than repaid by the reduced locking overheads.

6.2.1.2 *Binary search trees*

There are at least two existing concurrent search-tree designs which use per-node mutual-exclusion locks, both of which are motivated by the need for efficient querying in database systems.

Per-node locks (Kung)
Kung and Lehman [Kung80] note that deletion is the hardest operation to implement. They deal with this by deleting only nodes with at most one subtree. A node which has two subtrees is moved down the tree using standard rotation transformations until it satisfies the required property. However, each rotation requires two nodes to be replaced, to ensure consistency with other operations. Furthermore the rotations can cause the tree to become very unbalanced, so the algorithm attempts to apply the appropriate number of reverse rotations after the node is deleted; this may not be possible if concurrent updates have occurred.

Per-node locks (Manber)
Manber and Ladner [Manber84] describe a rather different approach that deals directly with nodes having two subtrees. Their solution is to replace the deleted node with a *copy* of its predecessor. Removal of the old version of the predecessor is postponed until all concurrent tree operations have completed. This is feasible only if a maintenance process is able to determine the complete set of currently-live operations. Each node may be tagged to indicate whether it is a redundant copy to be eliminated by the maintenance process, whether it is a copy of a previous node, and whether it is garbage. Interpreting and maintaining these tags correctly significantly complicates the algorithm.

Per-node locks (Fraser)
The principles applied in my threaded lock-free design can be transferred to a design that uses locks. As with the existing lock-based designs, this avoids the need for search operations to acquire locks. Operations which modify the tree must lock nodes which have any of their fields updated. To prevent deadlock, locks are acquired down the tree; that is, nodes nearest to the root are acquired first. This simple locking strategy, applied to a threaded representation, allows an efficient BST implementation with no need for a maintenance process or costly rotations.

6.2.1.3 *Red-black trees*

As I noted in Chapter 4, there are no existing designs for highly-parallel red-black trees. I therefore implemented two designs of my own: the first serialises all write operations using a single mutual-exclusion lock; and the second relaxes this constraint to allow greater parallelism. The designs, which I call **serialised writers** and **concurrent writers** in my evaluation, are described in greater detail in Section 4.5.2.

6.2.1.4 *Mutual-exclusion locks and multi-reader locks*

To achieve good performance on a highly-parallel system such as the Sun Fire server, these lock-based designs require carefully-implemented lock operations.

I implement mutual-exclusion locks using Mellor-Crummey and Scott's scalable queue-based spinlocks [Mellor-Crummey91a]. *MCS locks* avoid unnecessary cache-line transfers between processors that are spinning on the same lock by requiring each invocation of the *acquire* operation to enqueue a 'lock node' containing a private busy-wait flag. Each spinning process is signalled when it reaches the head of the queue, by the preceding process when it calls the *release* operation. Although seemingly complex, the MCS operations are highly competitive even when the lock is not contended; an uncontended lock is acquired or released with a single read-modify-write access. Furthermore, contended MCS locks create far less memory traffic than standard *test-and-set* or *test-and-test-and-set* locks.

Where multi-reader locks are required I use another queue-based design by the same authors [Mellor-Crummey91b]. In this case each element in the queue is tagged as a reader or a writer. Writers are removed from the queue one-by-one and enjoy exclusive access to the protected data. When a reader at the head of the queue is signalled it also signals the reader immediately behind it, if one exists. Thus a sequence of adjacently-queued readers may enter their critical regions simultaneously when the first of the sequence reaches the head of the queue.

6.2.2 Alternative non-blocking implementations

The lock-free MCAS algorithm presented in Chapter 3 cannot be fairly compared with any of the existing non-blocking designs from the literature. Each of the previous MCAS designs places at least one significant constraint on the locations that they update. For example, Israeli and Rappaport reserve a bit per process in each updated word [Israeli94]; this limits synthetic tests to at most 64 processors, and practical tests to far fewer so that there is space left in each word for application data. These limitations put the alternative designs beyond practical use for the dynamic MCAS-based designs in Chapter 4.

My STM-based search structures are evaluated using both FSTM and Herlihy *et al.*'s obstruction-free STM. Since the FSTM programming interface borrows heavily from their design, switching between the two designs is generally straightforward. One exception is that the restricted form of *early release* (Section 3.3.3) they provide cannot be applied to the red-black tree's 'sentinel node' (Section 4.5). This is because the sentinel node is opened for writing by all insertions and deletions, but the restricted early release operates only on read-only objects. I avoid this potential performance bottleneck by extending their transactional interface to allow particular objects to be registered with *non-transactional semantics*. Registered objects are accessed via the transactional interface in the usual way but are not acquired nor validated during a transaction's commit phase; instead, remote updates are ignored and updates by the committing transaction are thrown away. This interface extension is used by the red-black tree benchmark to register the sentinel node before measurements begin.

6.2.3 Results and discussion

In this section I present performance results for each of the parallel search structure designs that I described in Chapter 4. The results are split across a number of figures, each of which shows experimental results for one class of search structure: either skip lists, binary search trees, or red-black trees.

6.2.3.1 *Scalability under low contention*

The first set of results measure performance when contention between concurrent operations is very low. Each experiment runs with a mean of 2^{19} keys in the set, which is sufficient to ensure that parallel writers are extremely unlikely to update overlapping sections of the data structure. A well-designed algorithm which avoids unnecessary contention between logically non-conflicting operations should scale extremely well under these conditions.

Note that all the graphs in this section show a significant drop in performance when parallelism increases beyond 5 to 10 processors. This is due to the architecture of the underlying hardware: small benchmark runs execute within one or two processor 'quads', each of which has its own on-board memory. Most or all memory reads in small runs are therefore serviced from local memory which is considerably faster than transferring cache lines across the switched inter-quad backplane.

Figure 6.1 shows the performance of each of the skip-list implementations. As expected, the STM-based implementations perform poorly compared with the other lock-free schemes; this demonstrates that there are significant overheads associated with maintaining the lists of opened objects, constructing shadow

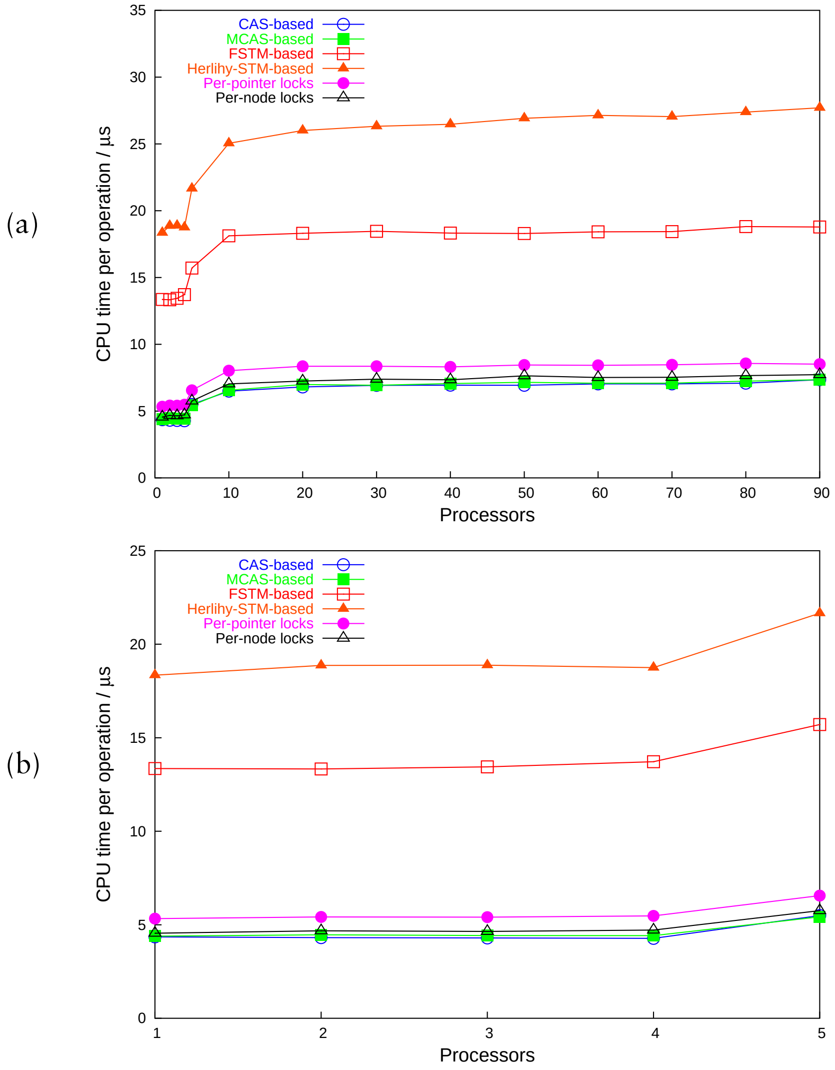

Figure 6.1: Graph (a) shows the performance of large skip lists ($K = 2^{19}$) as parallelism is increased to 90 processors. Graph (b) is a 'zoom' of (a), showing the performance of up to 5 processors.

copies of updated objects, and validating opened objects. Interestingly, under low contention the MCAS-based design has almost identical performance to the much more complicated CAS-based design — the extra complexity of using hardware primitives directly is not always worthwhile. Both schemes surpass the two lock-based designs, of which the finer-grained scheme is slower because of the costs associated with traversing and manipulating the larger number of locks.

Figure 6.2 shows results for the binary search tree implementations. Here the MCAS scheme performs significantly better than the lock-based alternatives, particularly as parallelism increases. This can be attributed to better cache lo-

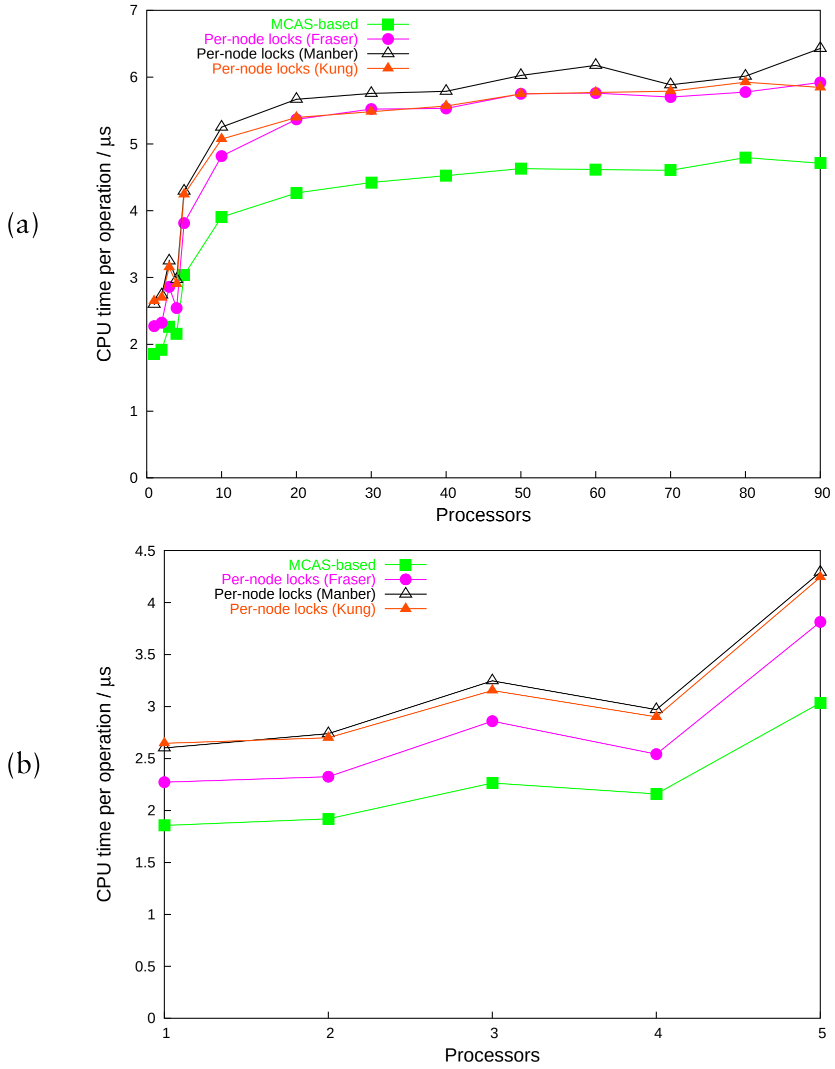

Figure 6.2: Graph (a) shows the performance of large binary search trees ($K = 2^{19}$) as parallelism is increased to 90 processors. Graph (b) is a 'zoom' of (a), showing the performance of up to 5 processors.

cality: the lock field adds a 33% space overhead to each node. Despite being simpler, my own lock-based design performs at least as well as the alternatives.

Figure 6.3, presenting results for red-black trees, gives the clearest indication of the benefits of lock-free programming. Neither of the lock-based schemes scales effectively with increasing parallelism. Surprisingly, the scheme that permits parallel updates performs hardly any better than the much simpler and more conservative design. This is because the main performance bottleneck in both schemes is contention when accessing the multi-reader lock at the root of the tree. Although multiple readers can enter their critical region simultaneously, there is significant contention for updating the shared synchronisation fields within the

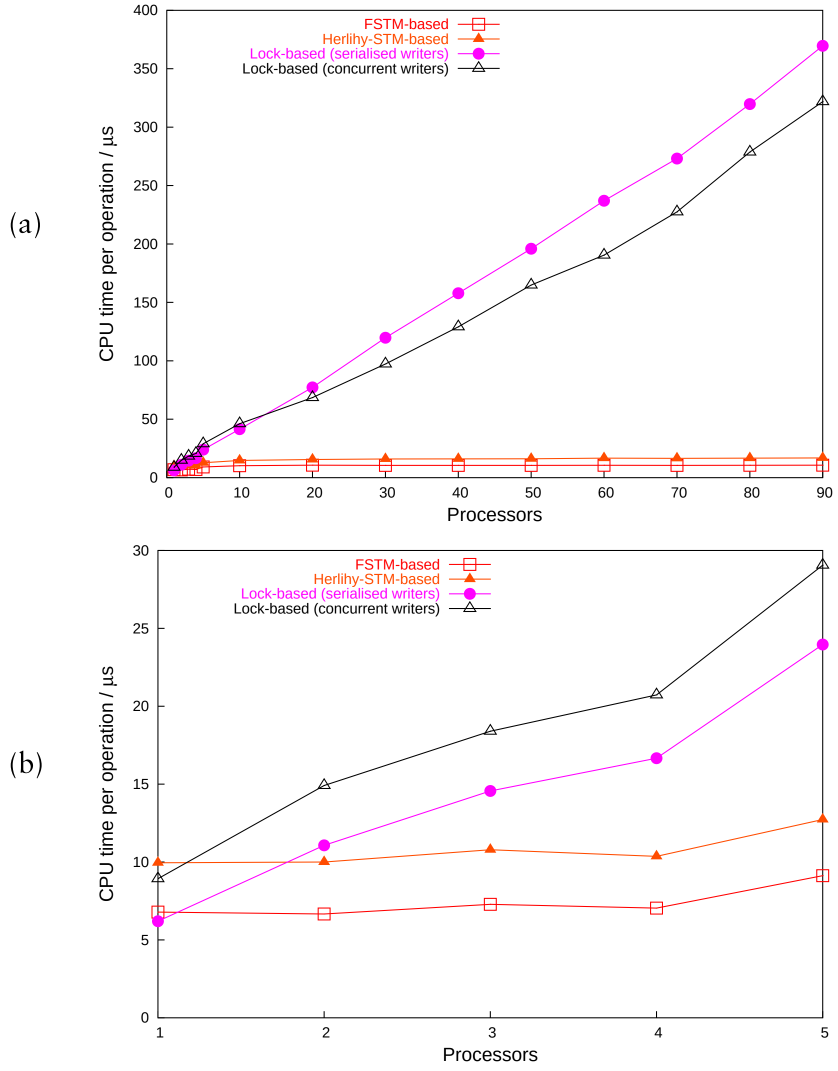

Figure 6.3: Graph (a) shows the performance of large red-black trees ($K = 2^{19}$) as parallelism is increased to 90 processors. Graph (b) is a 'zoom' of (a), showing the performance of up to 5 processors.

lock itself. Put simply, using a more permissive type of lock (i.e., multi-reader) does not improve performance because the bottleneck is caused by cache-line contention rather than lock contention.

In contrast, the STM schemes scale very well because transactional reads do not cause potentially-conflicting memory writes in the underlying synchronisation primitives. FSTM is considerably faster then Herlihy's design, due to better cache locality. Herlihy's STM requires a triple-indirection when opening a transactional object: thus three cache lines are accessed when reading a field within a previously-unopened object. In contrast my scheme accesses two cache lines; more levels of the tree fit inside each processor's caches and, when traversing

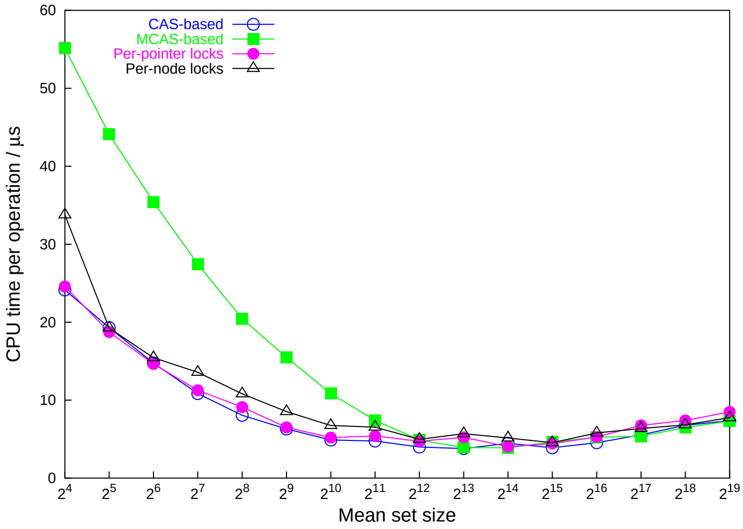

Figure 6.4: Effect of contention on concurrent skip lists ($P = 90$).

levels that do not fit in the cache, 50% fewer lines must be fetched from main memory.

6.2.3.2 *Performance under varying contention*

The second set of results shows how performance is affected by increasing contention — a particular concern for non-blocking algorithms, which usually assume that conflicts are rare. This assumption allows the use of *optimistic* techniques for concurrency control; when conflicts do occur they are handled using a fairly heavyweight mechanism such as recursive helping. Contrast this with using locks, where an operation assumes the worst and 'announces' its intent before accessing shared data. As I showed in the previous section, this approach introduces unnecessary overheads when contention is low: fine-grained locking requires expensive juggling of acquire and release invocations. The results here allow us to investigate whether these overheads pay off as contention increases. All experiments are executed with 90 parallel processes ($P = 90$).

Figure 6.4 shows the effect of contention on each of the skip-list implementations. It indicates that there is sometimes a price for using high-level abstractions such as MCAS. The poor performance of MCAS when contention is high is because many operations must retry several times before they succeed: it is likely that the data structure will have been modified before an update operation attempts to make its modifications globally visible. In contrast, the carefully-implemented CAS-based scheme attempts to do the minimal work necessary to update its 'view' when it observes a change to the data structure. This effort pays off under very high contention; in these conditions the CAS-based design

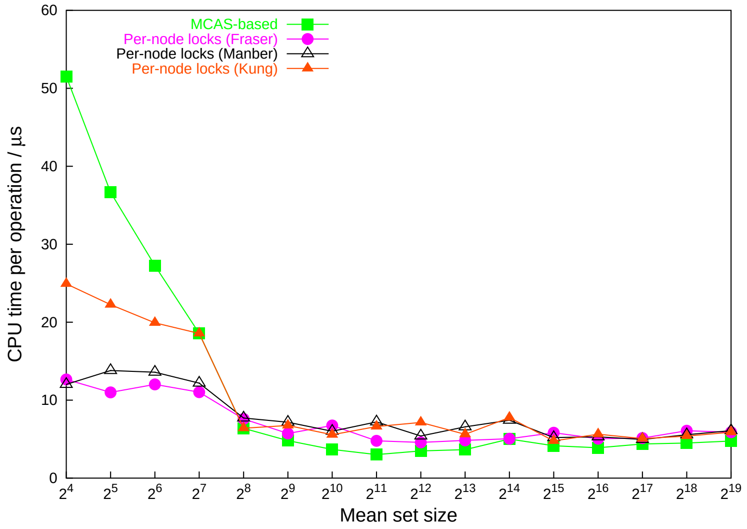

Figure 6.5: Effect of contention on concurrent binary search trees ($P = 90$).

performs as well as per-pointer locks. These results also demonstrate a particular weakness of locks: the optimal granularity of locking depends on the level of contention. Here, per-pointer locks are the best choice under very high contention, but they introduce unnecessary overheads compared with per-node locks under moderate to low contention. Lock-free techniques avoid the need to make this particular tradeoff. Finally, note that the performance of each implementation drops as the mean set size becomes very large. This is because the time taken to search the skip list begins to dominate the execution time.

Figure 6.5 shows performance results for binary search trees. As with skip lists, it demonstrates that MCAS-based synchronisation is not the best choice when contention is high. However, its performance improves quickly as contention drops: the MCAS scheme performs as well as the lock-based alternatives on a set containing just 256 keys. Further analysis is required to determine why Kung's algorithm performs relatively poorly under high contention. It is likely due, however, to conflicts introduced by the rotations required when deleting internal tree nodes.

Finally, Figure 6.6 presents results for red-black trees, and shows that locks are not always the best choice when contention is high. Both lock-based schemes suffer contention for cache lines at the root of the tree where most operations must acquire the multi-reader lock. The FSTM-based scheme performs best in all cases, although conflicts still significantly affect its performance. Herlihy's STM performs comparatively poorly under high contention, despite a contention-handling mechanism which introduces exponential backoff to 'politely' deal with conflicts. Furthermore, the execution times of individual opera-

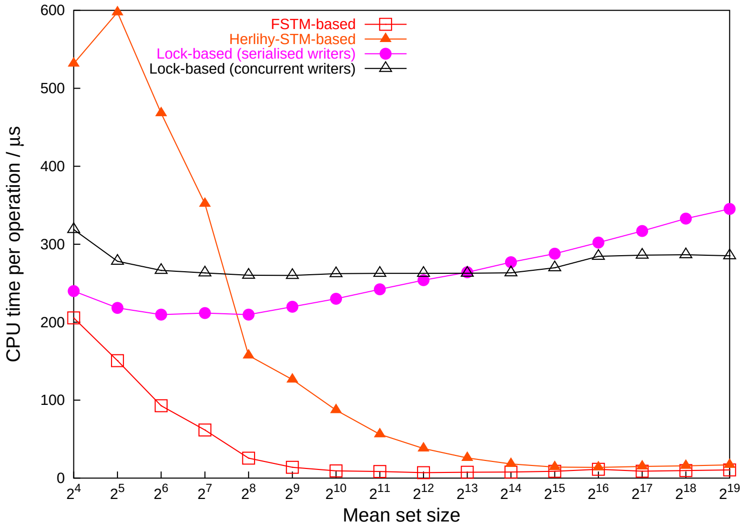

Figure 6.6: Effect of contention on concurrent red-black trees ($P = 90$).

tions are very variable, which explains the performance 'spike' at the left-hand side of the graph. This low and variable performance is caused by sensitivity to the choice of back-off rate: I use the same values as the original authors, but these were chosen for a Java-based implementation of red-black trees and they do not discuss how to choose a more appropriate set of values for different circumstances. A dynamic scheme which adjusts backoff according to current contention might perform better; however, this is a topic for future research.

6.3 Summary

The results I have presented in this chapter demonstrate that well-implemented lock-free algorithms can match or surpass the performance of state-of-the-art lock-based designs in many situations. Thus, not only do lock-free synchronisation methods have many *functional advantages* compared with locks (such as freedom from deadlock and unfortunate scheduler interactions), but they can also be implemented on modern multiprocessor systems with *better performance* than traditional lock-based schemes.

Figure 6.7 presents a comparison of each of the synchronisation techniques that I have discussed in this dissertation. The comparative rankings are based on observation of how easy it was to design practical search structures using each technique, and the relative performance results under varying levels of contention between concurrent update operations. *CAS*, *MCAS* and *FSTM* represent the three lock-free techniques that I have evaluated in this chapter. *RW-locks* repre-

		Run-time performance	
Rank	**Ease of use**	**Low contention**	**High contention**
1	FSTM	CAS, MCAS	CAS, W-locks
2	RW-locks	—	—
3	MCAS	W-locks	MCAS
4	W-locks	FSTM	FSTM
5	CAS	RW-locks	RW-locks

Figure 6.7: Effectiveness of various methods for managing concurrency in parallel applications, according to three criteria: ease of use for programmers, performance when operating within a lightly-contended data structure, and performance within a highly-contended data structure. The methods are ranked under each criterion, from best- to worst-performing.

sents data structures that require both read and write operations to take locks: these will usually be implemented using multi-reader locks. *W-locks* represents data structures that use locks to synchronise only write operations — some other method may be required to ensure that readers are correctly synchronised with respect to concurrent updates.

In situations where ease of use is most important, FSTM and RW-locks are the best choices because they both ensure that readers are synchronised with concurrent updates. FSTM is ranked above RW-locks because it avoids the need to consider issues such as granularity of locking and the order in which locks should be acquired to avoid deadlock. MCAS and W-locks have similar complexity: they both handle synchronisation between concurrent updates but an out-of-band method may be required to synchronise readers. Like FSTM, MCAS is ranked higher than W-locks because it avoids implementation issues that pertain only to locks. CAS is by far the trickiest abstraction to work with because some method must be devised to efficiently 'tie together' related updates to multiple memory locations.

When access to a data structure is not commonly contended, CAS and MCAS both perform very well. W-locks tend to perform slightly worse because of reduced cache locality compared with lock-free techniques, and the overhead of juggling locks when executing write operations. FSTM performs worse than CAS, MCAS and W-locks because of transactional overheads and the need to *double read* object headers to ensure that transactional reads are consistent during commit. RW-locks generally perform worst of all, particularly for a data structure which has only one point of entry: this *root* can easily become a performance bottleneck due to concurrent updates to fields within its multi-reader lock.

Under high contention, CAS-based designs perform well if they have been carefully designed to do the least possible work when an inconsistency or conflict is observed — however, this may require a very complicated algorithm. The extra space and time overheads of W-locks pay off under very high contention: MCAS performs considerably worse because memory locations are very likely to have been updated before MCAS is even invoked. FSTM also suffers because it, like MCAS, is an optimistic technique which detects conflicts *after* time has been spent executing a potentially expensive operation. However, it will still perform better than RW-locks in many cases because contention at the root of the data structure is still the most significant performance bottleneck for this technique.

If a data structure needs to perform well under greatly varying contention, then it may appear that direct use of CAS is the best option. Unfortunately the complexity of using CAS directly puts this option beyond reasonable use in most cases. Another possibility is to implement a hybrid scheme; for example, based on both MCAS and W-locks. Run-time performance feedback might then be used to dynamically select which technique to use according to the current level of contention. However, further research is required to determine whether the overheads of run-time profiling and switching between synchronisation primitives are outweighed by the benefits of using the most appropriate technique at any given time.

Chapter 7

Conclusion

In this dissertation I have introduced a number of techniques for managing the complexity of practical lock-free algorithms, and used these to implement real-world data structures. Experimental results demonstrate that my designs perform at least as well as, and often surpass, lock-based alternatives under reasonable levels of contention, yet provide all the usual benefits of lock-free design, including freedom from deadlock, no risk of lock convoying, and no need to choose between different locking granularities. In this chapter I summarise my contributions and describe some potential avenues for future research.

7.1 Summary

In Chapter 1 I began by motivating the need for alternatives to mutual exclusion when implementing highly-concurrent data structures. I then presented my thesis, that the practical lock-free programming abstractions I introduce in this dissertation allow a range of real-world data structures to be implemented for modern multiprocessor systems. Furthermore, these implementations can offer reduced complexity and improved performance compared with alternative lock-based and lock-free designs.

In Chapter 2 I discussed terminology and related work in the field of lock-free programming. The existing lock-free algorithms and techniques described in this chapter are impractical for general use in real applications due to excessive run-time overheads, or unrealistic constraints placed on the layout of shared memory or the structure of overlying applications.

In Chapter 3 I presented the main contribution of this dissertation: the first practical lock-free MCAS and STM designs. These make it easier to implement lock-free data structures, while incurring a very reasonable overhead compared with direct use of hardware primitives.

In Chapter 4 I presented a range of lock-free designs for real-world data structures, including skip lists, binary search trees, and red-black trees. Together these represent a further major contribution of this dissertation. Some of these designs are based on the abstractions presented in Chapter 3, providing insight into how they can simplify the programmer's task compared with direct use of atomic primitives such as CAS.

In Chapter 5 I discussed the issues raised when implementing the pseudocode designs of Chapters 3 and 4 on real hardware. These issues have frequently been ignored in previous work, but must be resolved to produce useful implementations for modern processor architectures.

Finally, in Chapter 6 I described how I tested my lock-free implementations for correctness. I then presented performance results for each of the lock-free and lock-based search structures running a parallel workload. Since many of the search structures are implemented using MCAS or FSTM, these results also demonstrate the practicality of these abstractions.

In conclusion, my thesis — that practical lock-free programming abstractions can be deployed on modern multiprocessor systems, and that this greatly simplifies the implementation of competitive lock-free data structures — is justified as follows. Firstly, I presented efficient designs for two suitable abstractions in Chapter 3. Secondly, the simple MCAS- and FSTM-based search structures that I presented in Chapter 4 demonstrate that real-world algorithms can be implemented over these high-level abstractions. Finally, the performance results in Chapter 6 show that structures implemented over these abstractions can match or surpass intricate lock-based designs. Using the lock-free programming abstractions that I have presented in this dissertation, it is now practical to deploy lock-free techniques, with all their attendant advantages, in many real-world situations where lock-based synchronisation would traditionally be the only viable option.

7.2 Future research

Obstruction-freedom is a recently-proposed progress guarantee for non-blocking algorithms which may eventually result in more efficient parallel applications. However, it is not yet clear whether this weaker guarantee will allow efficient implementation of a richer set of data structures. Further investigation of the 'programming tricks' that are permitted by obstruction-freedom, and deeper analysis of suitable contention-avoidance schemes, is required. For example, obstruction-freedom allows operations to *abort* each other, rather than introducing a mechanism for recursive helping. Obstruction-freedom may permit other performance-enhancing techniques that are disallowed in lock-free programs.

Efficient implementations of high-level abstractions are an important step towards deployment of lock-free techniques in real-world computer systems. However, there remains the problem of how these abstractions can best be presented to application programmers. Together with Harris I have made some progress in this area: they discuss how transactions can be presented in the Java programming language by introducing the *atomic* {...} construction [Harris03]. Shared-memory accesses within an atomic block are executed with transactional semantics.

Although lexical scoping of atomic regions looks attractive, it is not clear whether it is suitable in all situations. Different styles of concurrent programming could be investigated by introducing lock-free techniques into existing complex parallel systems, such as operating systems. The locking protocols in such systems are often complex: locks may be dynamically acquired and released, and the protocol must often interact with other concurrency mechanisms such as reference counts. These systems would provide an excellent testbed for experimenting with methods for applying lock-free techniques, and measuring the possible improvements in terms of reduced complexity and enhanced performance.

Finally, existing techniques for determining placement of memory barriers in parallel programs are not ideal. Manual placement does not scale well to large programs, and is prone to error. Furthermore, conservative placement is often required to produce a portable implementation that can be compiled for a wide range of processor architectures. Investigating semi-automated placement strategies would relieve some of this burden from the programmer and allow optimised placements to be calculated separately for each supported processor architecture. However, note that this is not an issue that affects *users* of lock-free data structures since, unlike the underlying memory-access primitives, these implementations are usually *linearisable*. Rather, automatic placement is intended to assist experts in the development of high-performance lock-free programming abstractions and data-structure libraries for use by mainstream application programmers.

Bibliography

[Adve96] Sarita V. Adve and Kourosh Gharachorloo. *Shared Memory Consistency Models: A Tutorial.* IEEE Computer, 29(12):66–76, 1996. (p 82)

[Alemany92] Juan Alemany and Edward W Felten. *Performance issues in non-blocking synchronization on shared-memory multiprocessors.* In Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC '92), pages 125–134, August 1992. (p 18)

[Alp00] *Alpha 21264/EV67 Microprocessor Hardware Reference Manual.* Hewlett Packard, 2000. (p 85)

[Anderson92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, 10(1):53–79, February 1992. (p 19)

[Anderson95] James H. Anderson and Mark Moir. *Universal Constructions for Multi-Object Operations.* In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95), pages 184–193, August 1995. (p 21)

[Anderson97] James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. *Implementing Wait-Free Objects on Priority-Based Systems.* In Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC '97), pages 229–238, August 1997. (p 21)

[Arcangeli03] Andrea Arcangeli, Mingming Cao, Paul McKenney, and Dipankar Sarma. *Using Read-Copy Update Techniques* *for System V IPC in the Linux 2.5 Kernel.* In Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track, pages 297–310, June 2003. (pp 26, 79)

[Barnes93] Greg Barnes. *A Method for Implementing Lock-Free Data Structures.* In Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 261–270, June 1993. (p 19)

[Bershad93] Brian N. Bershad. *Practical Considerations for Non-Blocking Concurrent Objects.* In Proceedings of the 13th International Conference on Distributed Computing Systems, pages 264–274. IEEE, May 1993. (p 18)

[Brinch Hansen78] Per Brinch Hansen. *Distributed Processes: A Concurrent Programming Concept.* Communications of the ACM, 21(11):934–941, November 1978. (p 20)

[Cormen90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* MIT Press, 1990. (p 69)

[DEC92] *Alpha Architecture Handbook.* Digital Press, 1992. (pp 17, 82, 85)

[Detlefs00] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. *Even Better DCAS-Based Concurrent Deques.* In Proceedings of the 14th International Symposium on Distributed Computing (DISC '00), pages 59–73. Springer-Verlag, 2000. (p 17)

[Detlefs01] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. *Lock-Free Reference Counting.* In Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC '01), pages 190–199, August 2001. (p 25)

[Ellis80] Carla Ellis. *Concurrent Search and Insertion in AVL Trees.* IEEE Transactions on Computers, C-29(9):811–817, 1980. (p 70)

[Fischer85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. *Impossibility of Distributed Consensus with One Faulty Processor.* Journal of the ACM, 32(2):374–382, 1985. (p 17)

[Greenwald02] Michael Greenwald. *Two-Handed Emulation: How to Build Non-Blocking Implementations of Complex Data Structures Using DCAS*. In Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC '02), pages 260–269, July 2002. (p 19)

[Greenwald96] Michael Greenwald and David Cheriton. *The Synergy Between Non-blocking Synchronization and Operating System Structure*. In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96), pages 123–136. USENIX Association, October 1996. (pp 24, 79)

[Greenwald99] Michael Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999. Also available as Technical Report STAN-CS-TR-99-1624, Stanford University, Computer Science Department. (pp 17, 21, 25)

[Hanke97] Sabine Hanke, Thomas Ottmann, and Eljas Soisalon-Soininen. *Relaxed Balanced Red-Black Trees*. In Proceedings of the 3rd Italian Conference on Algorithms and Complexity, volume 1203 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1997. (p 71)

[Hanke99] Sabine Hanke. *The Performance of Concurrent Red-Black Tree Algorithms*. In Proceedings of the 3rd Workshop on Algorithm Engineering, volume 1668 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 1999. (p 70)

[Harris01] Tim Harris. *A Pragmatic Implementation of Non-Blocking Linked Lists*. In Proceedings of the 15th International Symposium on Distributed Computing (DISC '01), pages 300–314. Springer-Verlag, October 2001. (pp 24, 25, 57)

[Harris03] Tim Harris and Keir Fraser. *Language Support for Lightweight Transactions*. In Proceedings of the 18th Annual ACM-SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '03), October 2003. (pp 23, 107)

[Herlihy02] Maurice Herlihy, Victor Luchangco, and Mark Moir. *The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures.* In Proceedings of the 16th International Symposium on Distributed Computing (DISC '02). Springer-Verlag, October 2002. (pp 26, 81)

[Herlihy03a] Maurice Herlihy, Victor Luchangco, and Mark Moir. *Obstruction-Free Synchronization: Double-Ended Queues as an Example.* In Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS). IEEE, May 2003. (pp 9, 15)

[Herlihy03b] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer. *Software Transactional Memory for Dynamic-Sized Data Structures.* In Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03), pages 92–101, 2003. (pp 22, 35, 48)

[Herlihy88] Maurice Herlihy. *Impossibility and Universality Results for Walt-Free Synchronization.* In Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC '88), pages 276–290, New York, August 1988. (p 17)

[Herlihy90a] Maurice Herlihy. *Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types.* ACM Transactions on Database Systems, 15(1):96–124, March 1990. (p 23)

[Herlihy90b] Maurice Herlihy. *A Methodology for Implementing Highly Concurrent Data Objects.* In Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 197–206, March 1990. (p 18)

[Herlihy90c] Maurice Herlihy and Jeannette M. Wing. *Linearizability: A Correctness Condition for Concurrent Objects.* ACM Transactions on Programming Languages and Systems, 12(3):463–492, July 1990. (p 16)

[Herlihy92] Maurice Herlihy and J. Eliot B. Moss. *Lock-Free*

Garbage Collection on Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 3(3):304–311, May 1992. (p 25)

[Herlihy93a] Maurice Herlihy. *A Methodology for Implementing Highly Concurrent Data Objects*. ACM Transactions on Programming Languages and Systems, 15(5):745–770, November 1993. (pp 17, 18)

[Herlihy93b] Maurice Herlihy and J. Eliot B. Moss. *Transactional Memory: Architectural Support for Lock-Free Data Structures*. In Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93), pages 289–301. ACM Press, May 1993. (p 21)

[Hoare74] C. A. R. Hoare. *Monitors: An Operating System Structuring Concept*. Communications of the ACM, 17(10):549–557, October 1974. Erratum in *Communications of the ACM* 18, 2 (Feb. 1975). (p 20)

[Hoare85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. (p 20)

[IBM70] IBM. *System/370 Principles of Operation*. Order Number GA22-7000, 1970. (pp 17, 33)

[Intel03] Intel. *Intel Itanium Architecture Software Developer's Manual, Volume 1: Application Architecture, Revision 2.1*. 2003. (p 82)

[Israeli94] Amos Israeli and Lihu Rappoport. *Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives*. In Proceedings of the 13nd Annual ACM Symposium on Principles of Distributed Computing (PODC '94), pages 151–160, August 1994. (pp 16, 20, 94)

[Kung80] H. T. Kung and Philip L. Lehman. *Concurrent Manipulation of Binary Search Trees*. ACM Transactions on Database Systems, 5(3):354–382, September 1980. (pp 26, 79, 89, 93)

[Kung81] H. T. Kung and John T. Robinson. *On Optimistic Methods for Concurrency Control*. ACM Transactions on Database Systems, 6(2):213–226, June 1981. (p 23)

[Lamport77] Leslie Lamport. *Concurrent Reading and Writing*. Communications of the ACM, 20(11):806–811, November 1977. (p 18)

[Lamport79] Leslie Lamport. *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*. IEEE Transactions on Computers, 28(9):690–691, September 1979. (p 81)

[Larson98] Per-Ake Larson and Murali Krishnan. *Memory allocation for long-running server applications*. In Proceedings of the ACM-SIGPLAN International Symposium on Memory Management (ISMM), pages 176–185, October 1998. (p 8)

[Liskov83] Barbara Liskov and Robert Scheifler. *Guardians and actions: linguistic support for robust, distributed programs*. ACM Transactions on Programming Languages and Systems, 5(3):381–404, July 1983. (p 20)

[Lomet77] D. B. Lomet. *Process Structuring, Synchronization and Recovery Using Atomic Actions*. In David B. Wortman, editor, *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 128–137. ACM, ACM, March 1977. (p 20)

[Manber84] Udi Manber and Richard E. Ladner. *Concurrency Control in a Dynamic Search Structure*. ACM Transactions on Database Systems, 9(3):439–455, September 1984. (pp 26, 79, 93)

[Massalin91] Henry Massalin and Calton Pu. *A Lock-Free Multiprocessor OS Kernel*. Technical Report CUCS-005-91, Columbia University, 1991. (p 24)

[Mellor-Crummey91a] John Mellor-Crummey and Michael Scott. *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*. ACM Transactions on Computer Systems, 9(1):21–65, 1991. (p 94)

[Mellor-Crummey91b] John Mellor-Crummey and Michael Scott. *Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors*. In Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 106–113, 1991. (pp 92, 94)

[Michael02] Maged M. Michael. *Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes.* In Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC '02), July 2002. (pp 26, 79, 81)

[Michael95] Maged M. Michael and Michael Scott. *Correction of a Memory Management Method for Lock-Free Data Structures.* Technical Report TR599, University of Rochester, Computer Science Department, December 1995. (pp 24, 78)

[Moir97] Mark Moir. *Transparent Support for Wait-Free Transactions.* In Distributed Algorithms, 11th International Workshop, volume 1320 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, September 1997. (pp 21, 22, 35)

[Motorola, Inc.] Motorola, Inc. *M68000 Family Programmer's Reference Manual.* Order Number M68000PM. (p 17)

[Perlis60] Alan J. Perlis and Charles Thornton. *Symbol Manipulation by Threaded Lists.* Communications of the ACM, 3(4):195–204, 1960. (p 61)

[Pugh90a] William Pugh. *Concurrent Maintenance of Skip Lists.* Technical Report CS-TR-2222, Department of Computer Science, University of Maryland, June 1990. (pp 26, 53, 55, 69, 79, 92)

[Pugh90b] William Pugh. *Skip Lists: A Probabilistic Alternative to Balanced Trees.* Communications of the ACM, 33(6):668–676, June 1990. (p 53)

[Shalev03] Ori Shalev and Nir Shavit. *Split-Ordered Lists: Lock-Free Extensible Hash Tables.* In Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03), pages 102–111, 2003. (p 92)

[Shavit95] Nir Shavit and Dan Touitou. *Software Transactional Memory.* In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95), pages 204–213, August 1995. (p 22)

[Tullsen95] D. Tullsen, S. Eggers, and H. Levy. *Simultaneous Mul-*

tithreading: Maximizing On-Chip Parallelism. In Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95), pages 392–403. ACM Press, June 1995. (p 8)

[Turek92] John Turek, Dennis Shasha, and Sundeep Prakash. *Locking without Blocking: Making Lock-Based Concurrent Data Structure Algorithms Nonblocking*. In Proceedings of the 11th ACM Symposium on Principles of Database Systems, pages 212–222, June 1992. (p 19)

[Valois95] John D. Valois. *Lock-Free Linked Lists Using Compare-and-Swap*. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95), pages 214–222, August 1995. (pp 24, 25)

[Weaver94] David Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9*. Prentice-Hall, 1994. (p 82)

[Wing93] Jeanette M. Wing and Chun Gong. *Testing and Verifying Concurrent Objects*. Journal of Parallel and Distributed Computing, 17(1):164–182, January 1993. (p 90)

[Wu93] Zhixue Wu. *A New Approach to Implementing Atomic Data Types*. PhD thesis, University of Cambridge, October 1993. Also available as Technical Report 338, University of Cambridge Computer Laboratory. (p 23)