Technical Report

UCAM-CL-TR-697
ISSN 1476-2986

Number 697

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Scaling Mount Concurrency: scalability and progress in concurrent algorithms

Chris J. Purcell

August 2007

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 July 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

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

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

ISSN 1476-2986

Abstract

As processor speeds plateau, chip manufacturers are turning to multi-processor and multi-core designs to increase performance. As the number of simultaneous threads grows, Amdahl's Law [6] means the performance of programs becomes limited by the cost that does not scale: communication, via the memory subsystem. Algorithm design is critical in minimizing these costs.

In this dissertation, I first show that existing instruction set architectures must be extended to allow general scalable algorithms to be built. Since it is impractical to entirely abandon existing hardware, I then present a reasonably scalable implementation of a map built on the widely-available compare-and-swap primitive, which outperforms existing algorithms for a range of usages.

Thirdly, I introduce a new primitive operation, and show that it provides efficient and scalable solutions to several problems before proving that it satisfies strong theoretical properties. Finally, I outline possible hardware implementations of the primitive with different properties and costs, and present results from a hardware evaluation, demonstrating that the new primitive can provide good practical performance.

Contents

List of figures **7**

1 Introduction **15**
1.1 Progress . . . 15
1.2 Scalability . . . 16
1.3 Contribution . . . 17
1.4 Outline . . . 17

2 Definitions **19**
2.1 Shared Objects . . . 19
2.2 Histories and Correctness . . . 20
2.3 Implementations and Synchronization . . . 22
2.4 Scalability . . . 23
2.5 Symbol Summary . . . 25

3 Related Work **27**
3.1 Primitives . . . 27
3.2 Progress . . . 29
3.3 Wait-Free Universality . . . 30
3.4 Lock-Free Universality . . . 31
3.5 Snapshot Objects . . . 31
3.6 Assistance . . . 34
3.7 DCAS . . . 35
3.8 Transactions . . . 36
3.8.1 Transactional Memory . . . 37
3.8.2 Software Transactional Memory and NCAS . . . 41
3.8.3 Hybrid Transactional Memory . . . 44

4 CAS is not Scalably Universal **47**
4.1 Definitions . . . 47
4.2 Scalability and Disjointness . . . 50
4.3 Scalability and Large Snapshots . . . 55
4.4 Load-Linked/Store-Conditional . . . 58

5 Reasonable Scalability: Open-Addressed Hashtables 61
5.1 Open-Addressing . . . 61
5.2 Bounding Searches . . . 63
5.3 Whack-a-Mole . . . 66
5.4 Inserting and Removing Keys . . . 70
5.5 Lock-Freedom and Multi-word Keys . . . 74
5.6 Value Replacement . . . 79
5.6.1 Migration . . . 79
5.6.2 In-Place . . . 82
5.6.3 Compacting Hybrid . . . 85
5.7 Storing Values on the Heap . . . 88
5.8 Dynamic Growth . . . 88
5.9 Evaluation . . . 90
5.9.1 Related Work . . . 90
5.9.2 Benchmark . . . 93
5.9.3 Discussion . . . 95

6 Diatomic Snapshot-Modify-Update 97
6.1 Snapshot Isolation . . . 97
6.2 Value Replacement . . . 101
6.3 Linked Lists . . . 106
6.4 Unbalanced Binary Trees . . . 109
6.5 Universality: Scalability and Progress . . . 117
6.5.1 Scalability . . . 117
6.5.2 Progress . . . 120

7 Implementing Diatomic Operations 125
7.1 Instruction Set Extension . . . 125
7.2 Hardware Designs . . . 127
7.2.1 Pragmatic Implementation . . . 127
7.2.2 Snapshot Set Implementation . . . 129
7.2.3 Timestamp Implementation . . . 132
7.3 Combining Operations . . . 133
7.4 Nestable Read-Like LL/SC Synergies . . . 135
7.5 Evaluation . . . 136
7.5.1 Results . . . 136
7.5.2 Avoidable Overhead . . . 139
7.5.3 Memory Footprint . . . 141
7.5.4 Discussion . . . 141

8 Conclusions **143**
8.1 Summary 143
8.2 Future Research 144
8.3 Acknowledgements 145

List of Figures

2.1 Possible linearizations for a sequence of operations. 21

3.1 Transactional memory on a machine with two processors. Memory accessed during a transaction is held in one of two 'transactional' states. Both caches may hold a copy of a cache line (here depicted as holding a single value) in shared mode, but only one can hold exclusive mode on a line at any one time. A transaction will abort rather than update a line held in the other cache, or read a line held in exclusive mode by the other cache. 37

4.1 Starting from logical state l, n disjoint update operations $o_1 \ldots o_n$ each update a different register in a shared memory. 51

4.2 History fragments $F_1 \ldots F_n$ allow the history HF to be extended to reach any of the sequentially-reachable states p_i without returning to logical state l. 52

4.3 History fragment G executes a single read operation, r, on logical state l (represented by sequentially-reachable state p). 53

4.4 History fragment G scheduled during a history chosen such that each r_i returns the same value, yet the history is never in logical state l during r's execution. 54

4.5 Implementing Compare-And-Swap from 8 to 15 in a simple, scalable, blocking implementation of a 4-bit register from a shared memory with only 2-bit registers. Offsets are counted from the left. 55

4.6 Each state j is connected to state 0 by fragments F_j and F_j^{-1}, following a path that can only go via states $[0, j]$, not $(j, s]$. . . . 56

4.7 History fragment G executes id on logical state l, represented by sequentially-reachable state $m(l)$. 57

4.8 If no r_i returns a unique value, history fragment G can be scheduled during a history chosen such that each r_i returns the same value, yet the history is never in logical state l during id's execution. . . 58

5.1 Bounds on collision indices for a hashtable holding keys 2, 7, 9, 12, 17. Hash function is $h(k) = k \bmod 8$, probe sequence is quadratic, p(k,i) = $(k + \frac{1}{2}(i^2 + i)) \bmod 8$. Key 17 is stored two steps along the probe sequence for bucket 1, so the probe bound is 2. 63

5.2 Problems maintaining a shared bound after a collision is removed from the end of the probe sequence. 64

5.3 Per-bucket probe bounds (code continued in Figure 5.8) 65

5.4 Moles and hammers: a uniqueness algorithm. Rosie reaches into Hammerspace and whacks Jim, preventing him from emerging simultaneously. 67

5.5 The whack-a-mole algorithm. Inserting value $v \in \mathbb{V}$, given primitive object m of type $\mathbb{F}$. 68

5.6 State machine used in hashtable. The mole represents a state transition which can only be taken after using the whack-a-mole algorithm to ensure uniqueness; only one bucket can be in the white-on-black `member` state at any one time for a given key. Note that the `busy` state intentionally appears twice. 71

5.7 Inserting key 12 with the whack-a-mole approach. 72

5.8 An obstruction-free set (continued from Figure 5.3) 73

5.9 State machine of a single bucket in the lock-free hashtable. Only one bucket may be in the white-on-black `member` state at any one time for a given key; the mole represents a state transition that can only be taken after ensuring this uniqueness with the whack-a-mole algorithm. Note that the `busy` state intentionally appears twice. 74

5.10 Problems assisting concurrent operations 75

5.11 Version-counted derivative of Figure 5.8 (continued in Figure 5.13) 76

5.12 Inserting key 12 (lock-free algorithm). As in the obstruction-free algorithm, duplicated attempts to insert the key are moved to `collided` state; however, the presence of version counters now allows the collided thread to assist the conflicting insertion to completion. The version count is incremented every time a bucket passes through `empty` state. 77

5.13 Lock-free insertion algorithm (continued from Figure 5.11) 78

5.14 *Migrating value replacement* hashtable state machine, simplified. The `collided` state is not shown. Only one bucket may be in a given white-on-black state at any one time for a given key, as guaranteed by the uniqueness algorithm introduced in Section 5.3. See Figure 5.24 for a more detailed diagram. 80

5.15 Migrating value replacement: A thread attempts to replace the value associated with key 17 from 891 to 112. The `changing` state represents a replacement 'mole' in the whack-a-mole consensus algorithm (a). Obstructing moles must be 'whacked' into `collided` state (b) before the replacement mole can move into `update` state (c). 81
5.16 Once a unique replacement has been chosen, the current `member` bucket is moved into `replaced` state (d), the `update` bucket is moved into `member` state in turn (e), and the `replaced` bucket emptied (f). 81
5.17 *In-place value replacement* hashtable state machine, simplified. Update buckets are no longer promoted to member state. Once again, the collided state is not shown. See Figure 5.24 for a more detailed diagram. 82
5.18 In-place value replacement: A thread attempts to replace the value associated with key 17 from 891 to 112. Once consensus on a unique replacement has been reached (a), the `update` bucket is moved into `copy` state (b), and the new value copied into the `replaced` bucket (c). 83
5.19 When the new value has been copied, the `copy` bucket is moved into `copied` state (d) before returning the `replaced` bucket to `member` state with a higher version count (e), and finally emptying the `copied` bucket (f). 83
5.20 Alternatively, a concurrent operation may delete the key–value pair by moving the `copy` bucket to `deleted` state (g) before moving the `replaced` bucket into `busy` state (h) and emptying the `deleted` bucket (i). 83
5.21 Alternatively, concurrent operations may reach consensus on a new replacement value (j), move the current `copy` bucket to `stale` state (k) and the `update` bucket into `copy` state (l), and finally empty the `stale` bucket (m). The thread copying the stale value in-place will then have to locate and copy the new value. 84
5.22 Key 17 migrates, allowing the probe sequence bound to be reduced. 85
5.23 If, during a scan, a key is always present in the table, it may be seen more than once (due to concurrent migration), but it will never be missed. 86
5.24 Conditions on state changes in the *compacting hybrid* value replacement model. Negative conditions must be observed on all buckets in the probe sequence, while positive conditions need only be observed on one. 87
5.25 Michael's algorithm: To insert a key, use CAS to swap in the new node. 90

5.26 Michael's algorithm: To erase a key, (a) mark the node as deleted, then (b) swap it out of the list. This latter step must be assisted by concurrent operations. 91
5.27 Lea's algorithm: To erase a key, the list is essentially duplicated node-for-node, though as an optimization the tail of the list after the erased node can be reused. 92
5.28 Performance of the competing map algorithms, without replacement, on a 16-way SPARC machine; lower is better. 94
5.29 Performance of the competing map algorithms on a 16-way SPARC machine; lower is better. 95
5.30 Performance of the replacement components of the competing map algorithms on a 16-way SPARC machine; lower is better. 96

6.1 Two concurrent diatomic operations both succeed, even though the snapshot of one overlaps the RMU of the other. As neither sees the other's update, neither operation can be linearized after the other, and the history as a whole is not linearizable; yet it is valid under snapshot isolation. 100
6.2 The simplest scalable solution combines reading the key–value pair (1) with the update of the value pointer (2) diatomically. 101
6.3 Code to replace the value associated with a key in a hashtable, using the `diatomically` construct. For simplicity, the function does not return the value replaced; this can be addressed. 102
6.4 An alternative solution allows the version counter to change when the value does, allowing safe concurrent assistance with a parity bit. An update finding a bucket with the relevant key (a) first updates the parity–value pair (b); any thread can then correct the resulting version–parity mismatch by incrementing the version counter (c). 102
6.5 Alternative code to replace the value associated with a key in a hashtable, using the `diatomically` construct only during updates. Once again, the function does not return the value replaced; this could easily be addressed. 103
6.6 Alternative code to lookup the value associated with a key in a hashtable, using the `diatomically` construct only during updates. 104
6.7 The third solution uses in-place copying. An update finding a bucket with the relevant key (a) writes a descriptor into the version–state field (b), updates the value in-place (c), then writes the new version–state pair (d). These last two steps can be concurrently assisted. 105
6.8 Interface for a linked list-based set built on diatomic operations. . 106
6.9 Public lookup function. Attempts to find the given key, using a diatomic construct to take a snapshot of the list. 107

6.10 Public insert function. Diatomically locates the correct location and swings a new node into the list. 107
6.11 Public erase function. Diatomically locates the target node and marks it as logically deleted, before running the find function repeatedly to ensure the node is removed. 108
6.12 Private find function for linked list. If a marked node is found, diatomically swings it out, deletes it, and instructs the caller to retry. Otherwise, finds the location for the given key in the absolutely-ordered list, returning whether or not the key is present. 108
6.13 Interface and data types for a lock-free unbalanced tree. 109
6.14 Steps in an example insertion of key 10. A thread encountering the tree in state (a) first descends the tree, searching for the correct place to insert the leaf, and ensuring no concurrent operations are in place that would obstruct it. In (b), the thread posts its new leaf into an existing node's control field. Any contending concurrent operations will now assist the insertion to completion, though searches will not yet find the new leaf. In (c), the thread swaps in a new interior node, making the new leaf visible to concurrent searches. Finally, in (d) the thread returns the control field to NULL. 112
6.15 Steps in an example deletion of key 8. A thread encountering the tree in state (e) first descends the tree, searching for the correct leaf, and ensuring no concurrent operations are in place that would obstruct it. In (f), the thread posts the leaf into its parent node's control field. Any contending concurrent operations will now assist the deletion, though searches will still see the leaf in place. The thread will now take steps to remove this parent. In (g), the thread now posts the parent node to the grandparent node's control field. To see why this is necessary, imagine that the uncle leaf (containing 14) is concurrently removed, and note that the grandparent would be removed by this operation. This conflict must be prevented before the parent node can safely be swapped out. In (h), the leaf and its parent can now be moved out of the tree by pointing the grandparent node at the deleted leaf's sibling. The leaf is no longer visible to concurrent searches. Finally, in (i) the thread returns the grandparent's control field to NULL and frees the deleted nodes. 113
6.16 Deleting a leaf is simplified if, as in (j), its parent is at the top of the tree: once the parent's control field has been updated, the parent and leaf can be swung immediately out of the tree and freed (k). 114
6.17 Insertion into the unbalanced tree, using the diatomically construction to ensure thread-safety (pseudocode continued in Figure 6.18) 115
6.18 Deleting from the unbalanced tree. 116

6.19 Implementing a blocking, scalable multi-object compare-and-swap primitive using diatomic operations. 119
6.20 A partial description of the `Transaction` class, containing a transaction encoded as a multi-object–compare-and-swap descriptor. . 120
6.21 A partial description of the `Object` class, showing the interface to its control field. 121
6.22 The transaction commit method. Building the descriptor and retrying on failure are left as exercises for the reader. 122
6.23 Helper functions for the transaction commit method. 123

7.1 If a sequence of reads hits in the cache, they must all have been present at the start of the sequence, assuming data is fetched only on demand. 127
7.2 Capacity misses due to a large working set, such as a large shared tree, will cause a pragmatic implementation of atomic snapshots to retry even in the absence of conflicting updates. 129
7.3 An update to location 0x1818 is detected and checked in parallel against the snapshot set. The location is not found in the fixed-size set, nor does it match the Bloom filter. 130
7.4 An update to location 0x2143 matches against the snapshot set, and is stored in the change set for later comparison. 132
7.5 A multiatomic operation created by combining two sequential diatomic operations. The second snapshot is combined with the first, saving the thread from having to read every word twice. However, the second update may fail after the first has succeeded; the algorithm must be robust against such partial updates. 134
7.6 Combining two diatomic operations on the fast path of Figure 6.11. 135
7.7 Performance of the competing tree algorithms, for smaller numbers of keys, on a 2-way PowerPC machine, with one and two threads; lower is better. 137
7.8 Performance of the competing tree algorithms, for larger numbers of keys, on a 2-way PowerPC machine, with one and two threads; lower is better. 138
7.9 Overhead of pragmatic implementation of diatomicity, showing the proportion of operations requiring at least one retry as occupancy and number of threads grows; lower is better. 139
7.10 Estimated overhead of snapshot set implementation of diatomicity, showing the proportion of operations requiring at least one retry as occupancy and number of threads grows; lower is better. 140
7.11 Memory use of the competing tree algorithms, with one to four threads; lower is better. 141

Chapter 1

Introduction

As processor speeds plateau, chip manufacturers are turning to multi-processor and multi-core designs to increase performance. As the number of simultaneous threads grows, Amdahl's Law [6] means the performance of programs becomes limited by the cost that does not scale: communication, via the memory subsystem. Algorithm design is critical in minimizing these costs.

I will show that the hardware primitives provided by existing architectures, and assumed by much previous research, are insufficient to avoid unnecessary communication overhead without memory costs growing with the number of threads. This result motivates my dissertation.

In this chapter, I outline some basic theoretical properties that have been explored in earlier work; the contributions made in this dissertation; and the structure of the remaining chapters.

1.1 Progress

The dominant paradigm in multithreaded algorithm design is *mutual exclusion*: threads executing critical sections of code exclude concurrent operations, preventing them from seeing inconsistent state or making erroneous and damaging updates. Mutual exclusion is usually negotiated with *locks*, which can only be held by one thread at a time.

Preemptive systems suspend the active thread, to handle interrupts, run priority code, or simply to give the illusion of parallelism. However, mutual exclusion does not interact well with preemption: if a suspended thread holds a lock, the active thread may be unable to make progress. Solving this problem while still using mutual exclusion for safety typically means making locking visible to preemption control, such as by suspending preemption during a critical section.

Non-blocking algorithms guarantee that suspension of a single thread will not affect the *progress* of other threads, allowing arbitrary preemption without knowledge of the state of the thread safety mechanism. 'Progress' here is defined

on a per-algorithm basis.

Three non-blocking progress guarantees have been identified in the literature; these will be introduced in Section 3.2. Note that the pivotal, negative result of my thesis is progress-guarantee–agnostic.

1.2 Scalability

In this dissertation, I am concerned with two kinds of scalability: *communication* (or synchronization) and *storage*.

An algorithm which scales perfectly in communication — no synchronization costs — is in general impossible, as threads cannot exchange information. Perfect storage scalability — no increase in memory use as the number of threads increases — is similarly impractical. However, four properties have emerged that are highly desirable in a generic algorithm. I give intuitive summaries here; more rigorous definitions can be found in the next chapter.

Disjoint-access parallelism. Operations that access or modify disjoint state run concurrently without communication. For instance, a disjoint-access parallel memory allows processors to read and modify different cachelines without requiring communication over the memory bus.

Read parallelism. Operations that do not update any shared state run concurrently without communication. For instance, a read parallel memory allows multiple processors to read from local copies of many shared cachelines without memory bus traffic.

Population obliviousness. Roughly speaking, an individual thread does not know the size of the population of threads that might run concurrent operations. Memory subsystems are not usually population oblivious; there is a fixed limit on the number of processors which can be added. In contrast, mutual exclusion algorithms are often population oblivious, as the memory requirements of a single lock in the absence of contention are fixed, regardless of how many threads may try to access it concurrently later on.

Garbage freedom. Roughly speaking, a system is garbage free if its memory requirements do not grow with time. Most algorithms are reasonably garbage free, as blossoming memory costs are highly visible. However, subtler problems are also excluded by garbage freedom. For instance, timestamps which cannot be reused theoretically require the storage used for timestamps to growing without bound. In practice, it may be implausible that a program generate enough garbage to create a problem; for instance, one might show that 64-bit timestamps would last longer than the lifetime of the Earth under reasonable assumptions.

I call an algorithm which satisfies disjoint-access and read parallelism, population obliviousness and garbage freedom, *scalable*.

1.3 Contribution

It is my thesis that existing instruction set architectures must be extended to allow general scalable algorithms to be built, and that this can be done without incurring detrimental hardware costs.

My first contribution is to provide formal definitions of the four scalability properties, introduced informally above, leading to a proof that existing single- and double-word primitives cannot implement arbitrary shared objects with all of the four scalability properties. This result is independent of requirements on progress, applying to both non-blocking and mutual exclusion–based algorithms.

Since it is impractical to entirely abandon existing hardware, my second contribution is a novel non-blocking implementation of a map using an open-addressed hashtable design, based on the widely-available single-word compare-and-swap (CAS) primitive. This algorithm is scalable under certain reasonable assumptions about its usage, occupying a new point in the progress–scalability design space, but it is not truly garbage-free, disjoint-access parallel or population oblivious, restricting the algorithm's range of applicability.

Another contribution is a new hardware primitive called a *diatomic operation*. I will show that this construction allows scalable, non-blocking implementations of several data structures, before proving that it is universal for building scalable, non-blocking algorithms. It is thus as strong as existing proposals for extending architectures on a theoretical footing, and stronger than existing primitives.

My final contribution is to outline possible hardware implementations of diatomic operations with different properties and costs, and quantitively compare the performance of a pragmatic implementation against existing solutions. I will thereby show that such extensions can indeed be made without a negative performance impact on the rest of the system.

Part of this work has been published previously ([69], [70]).

1.4 Outline

In Chapter 2, I give rigorous definitions of terms used in the dissertation.

In Chapter 3, I cover prior work related to the subject of my thesis.

In Chapter 4, I show that existing single- and double-word primitives cannot implement transactional memory with all four scalability properties.

In Chapter 5, I describe how to implement a lock-free, reasonably scalable map based on an open-addressed hashtable using the widely-available compare-and-swap instruction.

In Chapter 6, I introduce a new hardware primitive, the *diatomic operation*, and present several algorithms built from it, including a scalable, lock-free, tree-based set. I then show that it is universal for scalable, non-blocking algorithms.

In Chapter 7, I introduce an instruction set extension enabling the use of diatomic operations, and outline several possible hardware implementations with different properties and costs. The most pragmatic implementation can be emulated on existing hardware, allowing an empirical evaluation of the practicality of diatomic operations.

Finally, in Chapter 8, I conclude the dissertation and consider avenues of future research.

Chapter 2

Definitions

In this chapter, I give formal definitions of several terms used throughout this dissertation.

2.1 Shared Objects

A shared object has a type $\mathbb{T} = (\mathbb{S}, \mathbb{S}^0, \mathbb{O}, R)_\mathbb{T}$ defining a set of possible *states*, $\mathbb{S}_\mathbb{T}$, a set of distinguished *starting states*, ${\mathbb{S}_\mathbb{T}}^0$, a set of *operations*, $\mathbb{O}_\mathbb{T}$, that provide the only means to manipulate the object, and a set of return values, $R_\mathbb{T}$. Each operation u is a map from the states $s \in \mathbb{S}_\mathbb{T}$ to a finishing state $u \circ s \in \mathbb{S}_\mathbb{T}$ and a return value $u(s) \in R_\mathbb{T}$.

One canonical example I will be considering often is a *shared memory*: a large set of finite-sized registers, or *words*. (For the majority of this dissertation, I conform to the common practice amongst algorithm researchers of using "register" to refer to a shared memory location, not a processor-specific unit of temporary storage.) I denote a shared memory of n b-bit registers by $\mathcal{M}_b^n$. Operations must include a read for each location, $\mathsf{READ}[i]$, and a write for each location–value pair, $\mathsf{WRITE}[i, v]$: n read operations and $2^b n$ write operations.

$$\mathbb{S}_{\mathcal{M}_b^n} = [0, 2^b)^n$$
$$\mathbb{S}^0_{\mathcal{M}_b^n} = (0, \ldots, 0)$$
$$\mathbb{O}_{\mathcal{M}_b^n} \supseteq \{\mathsf{READ}[i] : i \in [0, n)\} \cup \{\mathsf{WRITE}[i, v] : i \in [0, n), v \in [0, 2^b)\}$$
$$R_{\mathcal{M}_b^n} \supseteq [0, 2^b) \cup \emptyset$$

$$\begin{aligned}
\mathsf{READ}[i] \circ \mathbf{s} &= \mathbf{s} \quad \forall i, \mathbf{s} \in \mathbb{S}_{\mathcal{M}_b^n} \\
\mathsf{READ}[i](\mathbf{s}) &= s_i \quad \forall i, \mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{S}_{\mathcal{M}_b^n} \\
\mathsf{WRITE}[i, v] \circ \mathbf{s} &= (s_0, \ldots, s_{i-1}, v, s_{i+1}, \ldots, s_{n-1}) \\
&\qquad \forall i, v, \mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{S}_{\mathcal{M}_b^n} \\
\mathsf{WRITE}[i, v](\mathbf{s}) &= \emptyset \quad \forall i, v, \mathbf{s} \in \mathbb{S}_{\mathcal{M}_b^n}
\end{aligned}$$

For any type, I define the set of *read operations*, $\mathbb{R}_\mathbb{T}$, as the set of operations that do not change the state of the object.

$$\mathbb{R}_\mathbb{T} = \{r \in \mathbb{O}_\mathbb{T} : r \circ s = s \; \forall s \in \mathbb{S}_\mathbb{T}\}$$

In a shared memory,

$$\mathbb{R}_{\mathcal{M}_b^n} \supseteq \{\mathsf{READ}[i] : i = 0 \ldots n-1\}$$

I call type $\mathbb{T}$ a *snapshot object* if $\exists \mathrm{ID} \in \mathbb{R}_\mathbb{T}$ with $\mathrm{ID}(s) = s \; \forall s \in \mathbb{S}_\mathbb{T}$: if there is a read operation which returns the entire state of the object. Shared memories are not typically snapshot objects; however, a fruitful area of research has been implementing (small) shared memories with these "atomic snapshot" operations — see Section 3.5.

2.2 Histories and Correctness

I assume an asynchronous execution model. An *event* consists of an invocation, a subsequent response, and modification and total footprints, defined later. Each thread executes a sequence of events, defining a *history* of invocations and responses with a total ordering, called *real-time*. (Note that 'incomplete' histories, containing unmatched invocations and responses, are ruled out by this definition; related work may call these *complete* histories.) An event A is said to *precede* B if the response to A occurs before the invocation of B, while the events are *concurrent* if neither A precedes B nor B precedes A. A *sequential* history is one in which each invocation is followed immediately by its corresponding response, i.e. with no concurrent events. I denote the set of all histories by $\mathbb{H}$, and Events is defined as the set of all events in all histories. $\mathbb{H}_t \subseteq \mathbb{H}$ is the set of all histories H valid with a thread pool of exactly t threads.

The basic correctness requirement for a shared object is *linearizability* [36], which requires that for every valid history, there exists some sequential history containing the same invocations and responses, such that any operation A preceding an operation B in the original history also precedes it in the sequential one. Linearizability means that operations appear to take effect atomically at some point between their invocation and response. Each event A in a linearizable history thus represents an operation u_A on a state s_A, and a linearizable history can also be represented by the sequence of states and operations of its sequential counterpart.

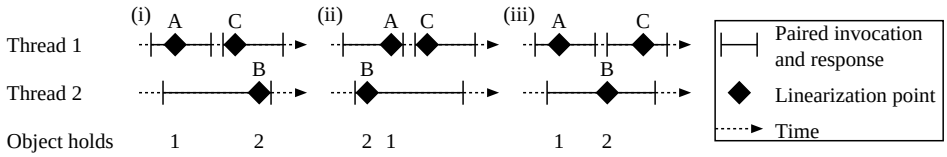


Figure 2.1: Possible linearizations for a sequence of operations.

For instance, a non-sequential history of a shared memory $\mathcal{M}_b^n$ might involve two threads T_1 and T_2, and three events, A, B and C. T_1 executes a single write, $\mathsf{WRITE}[0,1]$, and a subsequent read, $\mathsf{READ}[0]$; these are events A and C. T_2 concurrently executes a single write, $\mathsf{WRITE}[0,2]$; this is event B.

Suppose the history is as follows: T_1 invokes A; T_2 invokes B; A responds; T_1 invokes C; B responds; C responds. This history is not sequential; it is not obvious how the events that are scheduled should interact. What values could the read of event C legitimately return?

As shown in Figure 2.1, the possible linearization orders are: (i) ACB, (ii) BAC or (iii) ABC. In the former two, C should return the value written by A, 1; in the latter, C should return the value written by B, 2. Since A precedes C in the non-sequential history, it must also do so for any linearized ordering. This rules out other orderings, such as CAB, where C would return the value originally held by register 0, namely 0. If the shared memory is linearizable, therefore, the only values that can be returned in this non-sequential history are 1 or 2.

A *history fragment* is any part of a history whose invocations and responses are matched. I denote the set of all history fragments by $\mathbb{F}$. I write $threads(F)$ for the number of threads executing events in $F \in \mathbb{F}$, and $A \overset{\text{t}}{\sim} B$ iff events A and B are invoked by the same thread. $\langle A_1 \cdots A_n \rangle$ is the history fragment representing the sequential execution of events A_1 through A_n.

Two history fragments F and F' are *sequentially consistent* if each thread issues the same sequence of invocations, gets the same responses, and if the final state of the object is the same. I denote this by $F \sim F'$. In particular, any fragment is sequentially consistent with its linearization. (It is often convenient when considering sequentially consistent fragments to identify the events they contain.)

If the events in history H followed by those in history fragment F form a valid history H', I refer to F as *extending* H to form history HF, where $HF = H'$.

2.3 Implementations and Synchronization

An *implementation* $\mathbb{M}$ constructs a logical object, type $\mathbb{L}$, from a primitive object, type $\mathbb{P}$. Multiple primitive objects can be treated as a single object by considering the disjoint union of their states and operations. For any history H, $prim(H)$ is the set of primitive states in the history, and $logic(H)$ the set of logical states.

Until now, my definitions have been taken from previous work; for my thesis, however, I need rigorous definitions of a few more ideas. I therefore require that a type also provide a set of *synchronization points*, $\mathbb{Y}_{\mathbb{T}}$, and two functions $\mathsf{Events} \to \mathscr{P}(\mathbb{Y}_{\mathbb{T}})$: the *modification footprint* $f^m_{\mathbb{T}}(A)$ and the *total footprint* $f_{\mathbb{T}}(A)$. These must satisfy:

$$\left.\begin{array}{c} f^m_{\mathbb{T}}(A) \subseteq f_{\mathbb{T}}(A) \\ f^m_{\mathbb{T}}(A) = \emptyset \Rightarrow u_A \circ s_A = s_A \end{array}\right\} \forall A \in \mathsf{Events}$$

$$\left(\begin{array}{c} f^m_{\mathbb{T}}(A) \cap f_{\mathbb{T}}(B) = \emptyset \\ f^m_{\mathbb{T}}(B) \cap f_{\mathbb{T}}(A) = \emptyset \end{array}\right) \Rightarrow \langle A\ B\rangle \sim \langle B\ A\rangle \quad \forall \langle A\ B\rangle \in \mathbb{F}$$

These synchronization points summarize where operations must communicate, either by reading from or by updating portions of the object's state. Note that any operation with an empty modification footprint must be a read operation, but the converse is not true.

For the shared memory $\mathcal{M}^n_b$, the registers themselves are the synchronization points: $\mathbb{Y} = [0, n)$. The modification footprint of a write operation is the register it overwrites, while read operations have no modification footprint. The total footprint of both types of operation is the register involved.

u_A	$f(A)$	$f^m(A)$
$\mathsf{READ}[i]$	$\{i\}$	$\emptyset$
$\mathsf{WRITE}[i, v]$	$\{i\}$	$\{i\}$

If a shared memory provided a snapshot operation, ID, it would satisfy:

u_A	$f(A)$	$f^m(A)$
ID	$\mathbb{Y}$	$\emptyset$

In general, two operations must communicate if they do not commute; however, in real implementations, some commuting operations will still communicate. Two events run in different threads that do not communicate are said to execute in parallel: formally, a history fragment F *executes in parallel*, denoted by $F \rightrightarrows_{\mathbb{T}}$, if

$$F \rightrightarrows_{\mathbb{T}} \quad \stackrel{\text{def}}{\iff} \quad \forall A, B \in F, f^m_{\mathbb{T}}(A) \cap f_{\mathbb{T}}(B) \neq \emptyset \Longrightarrow A \overset{\text{t}}{\sim} B$$

In a shared memory, two operations will run in parallel if they are on different registers, and two read operations will always run in parallel. A snapshot operation will not run in parallel with any update operation.

I denote the combined modification (resp. total) footprint of the primitives used in the implementation of $A \in \mathsf{Events}$ by $f^m_{\mathbb{M}}(A)$ (resp. $f_{\mathbb{M}}(A)$).

2.4 Scalability

Amdahl's Law states that for highly-concurrent programs, performance will be limited by the cost that does not scale: communication. It is therefore important that implementations of shared objects preserve the potential parallelism available in the logical object being implemented. For instance, a user of an implementation of a shared memory with a snapshot operation would not be surprised that a snapshot would not run in parallel with an update operation. They would find it hard to use it scalably, however, if write operations to different registers had to communicate, or if two read operations on the same register did.

An implementation is *read parallel* if $\forall F \in \mathbb{F}\ (\forall A \in F\ (u_A \in \mathbb{R}_{\mathbb{L}}) \Rightarrow F \rightrightarrows_{\mathbb{M}})$: if all history fragments containing only read operations must execute in parallel.

An implementation is *disjoint-access parallel* if $\forall F \in \mathbb{F}\ (\forall \text{ distinct } A, B \in F\ (f_{\mathbb{L}}(A) \cap f_{\mathbb{L}}(B) = \emptyset) \Rightarrow F \rightrightarrows_{\mathbb{M}})$: if any history fragment where each thread executes operations whose footprints lie in disjoint sets of synchronization points must execute in parallel.

An implementation is *parallelism preserving* if $\forall F \in \mathbb{F}\ (F \rightrightarrows_{\mathbb{L}} \Rightarrow F \rightrightarrows_{\mathbb{M}})$: if any history fragment where each thread executes operations whose *modification* footprints are disjoint from all other events' *read* footprints must execute in parallel.

Any parallelism preserving implementation is also disjoint-access and read parallel; the converse is not true. For example, in an implementation of a binary tree, disjoint-access parallelism is not a useful property as all update operations must read the root of the tree, and so none are logically disjoint. Parallelism preservation is more relevant for such objects, as it implies updates run in parallel despite overlapping total footprints.

Synchronization scalability is only half of the picture, however. Equally important is that an implementation scale well in the amount of resources it consumes, both over time and as the number of threads grows. I wish to prevent an implementation from creating garbage (states that are unsafe to reuse) over time, as this prevents other algorithms, threads and processes from using those locations. I also wish to prevent an algorithm from requiring increasing investment of time and resources as the thread population grows, unless the activity of those threads demands it. The follow formalise these requirements.

An implementation is *garbage-free* if $\forall H \in \mathbb{H}$ ($|\text{logic}(H)| < \infty \Rightarrow |\text{prim}(H)| < \infty$): if a history visits an infinite set of primitive states, it must have visited an infinite set of logical states too.

$\mathbb{M}$ is *population oblivious* if $\forall t < t'$ ($\mathbb{H}_t \subseteq \mathbb{H}_{t'}$): the footprint of an operation does not depend on the size of the thread population.

I require that a *scalable* implementation of a shared object be at a minimum read parallel, disjoint-access parallel, population oblivious and garbage-free, allowing good preservation of the parallelism inherent in the workload without escalating memory costs.

2.5 Symbol Summary

Symbol	Description	Page
$\mathbb{T}$	A shared object type	19
$\mathbb{P}$	A primitive shared object type	22
$\mathbb{L}$	A logical shared object type	22
$\mathbb{M}$	An implementation of a logical object	22
$\mathbb{S}_{\mathbb{T}}$	States of type $\mathbb{T}$	19
$\mathbb{O}_{\mathbb{T}}$	Operations of type $\mathbb{T}$	19
$\mathbb{R}_{\mathbb{T}}$	Read operations of type $\mathbb{T}$	20
$\mathbb{Y}_{\mathbb{T}}$	Synchronization points of type $\mathbb{T}$	22
$u \circ s$	State after applying operation u to state s	19
$u(s)$	Return value after applying operation u to state s	19
u^{-1}	Inverse of operation u (dependent on starting state)	48
$\mathcal{M}_b^n$	Shared memory — n b-bit registers	19
$\mathbb{H}$	Execution histories	20
$\mathbb{H}_t$	Histories valid with a thread pool of t threads	20
$\mathbb{F}$	History fragments	21
HF	History H extended with fragment F	21
$F \sim F'$	Fragments F and F' are sequentially consistent	21
$prim(H)$	Primitive states in history H	22
$logic(H)$	Logical states in history H	22
$A \overset{\text{t}}{\sim} B$	Events A and B are executed by the same thread	21
$\langle A_1 \cdots A_n \rangle$	Sequential execution of events A_1 through A_n	21
$f_{\mathbb{T}}^m(A)$	Modification footprint of event A on type T	22
$f_{\mathbb{M}}^m(A)$	Modification footprint of A in implementation M	23
$f_{\mathbb{T}}^m(u)$	Modification footprint of an operation	48
$f_{\mathbb{T}}(A)$	Total footprint of event A on type T	22
$f_{\mathbb{M}}(A)$	Total footprint of A in implementation M	23
$f_{\mathbb{T}}(u)$	Total footprint of an operation	48
$F \rightrightarrows_{\mathbb{T}}$	History fragment F executes in parallel on type T	22
$S \rightrightarrows_{\mathbb{T}}$	Operations S executes in parallel on type T	48
$\mathcal{D}(\mathbb{T})$	Maximal disjointness of orthogonal type T	50

Chapter 3

Related Work

In this chapter, I cover previous work related to the subject of the thesis.

All multi-processor systems with shared memory must provide primitives with a well-defined set of behaviours when multiple processors access the same register concurrently. A question that naturally arises is: what primitives is it necessary to provide to allow all algorithms to be implemented (a property known as *universality*)? And what restrictions (e.g. guaranteed progress, bounded memory consumption) can be imposed on the implementations?

Section 3.1 covers basic primitives that have been proposed in earlier work, and Section 3.2 introduces several progress guarantees that have been considered. Sections 3.3 and 3.4 describe work done on *universal constructions* — code transformations, typically from sequential code, yielding concurrent algorithms — for various primitives and progress guarantees.

Section 3.5 covers a special case in concurrent algorithms: shared memories with a snapshot operation. Section 3.6 discusses the general topic of assisting obstructing threads to completion in lock-free algorithms. Section 3.7 covers algorithms built from DCAS, a powerful primitive making many simpler concurrent algorithms, such as reference counting, trivial, but a primitive with no well-performing implementation on any platform. Finally, Section 3.8 covers a growing movement in concurrency research: providing a convenient abstraction, *transactions*, for writing concurrent algorithms.

3.1 Primitives

Many primitive atomic operations have been suggested in the literature, though not all have been implemented in production hardware. These are generally guaranteed to be *atomic*, also known as *linearizable* (see Section 2.2).

Read and write registers only support concurrent atomic *reading* and *writing*. Reads are guaranteed to return the last value written. (Compare with "safe" registers, where reads may return any arbitrary value if run during a concurrent

write; and unsafe registers, which additionally may contain any arbitrary value after two writes occur concurrently. Neither of these are atomic.)

Most research assumes a stronger, combined read-and-update primitive, usually assumed to coexist with atomic reads and writes of the same register:

Test-and-set: Sets one bit of a register and returns the value the bit held immediately before. Test-and-set is sufficient to implement a simple spin-lock, repeatedly attempting to set a lock bit, and entering the critical section only if the bit is found to have been clear.

Swap: Writes a value to a register and returns the value it previously held.

Fetch and add: Atomically increments a register, returning the old value.

Sticky bits: Tri-valued objects taking one of 0, 1 or `undecided`. They provide an atomic read, and an atomic transition *out* of the undecided state, but only a "safe" transition *back* to undecided state, which produces unpredictable results if it overlaps any other operation.

CAS (Compare-And-Swap): Takes a register, an *expected* and a *new* value; returns the value held by a register, and replaces it with `new` only if it matches `expected`.

CAS allows a trivial lock-free (see Section 3.2) implementation of the preceding primitives, and indeed any atomic single-location read-and-update primitive, by reading the register, calculating the desired new value, and attempting to update the location, retrying if it no longer contains the same value.

A traditional problem with writing concurrent algorithms using CAS is that a read-CAS pair is not guaranteed to be undivided: a register containing A when first read, and still containing A when a subsequent CAS succeeds, may nevertheless have held intermediate value B. This is commonly called the *ABA problem* [1].

LL/SC (Load-Linked, Store-Conditional): A pair of operations, together forming a read-and-modify primitive. A load-linked operation simply returns the value stored in a register; a subsequent store-conditional to that register will only succeed if the LL/SC pair executed atomically (that is, if the register has not been modified since the previous load-linked operation on that register by that thread).

Strong LL/SC further guarantees that a store-conditional will only fail if the location *has* been modified, and allows LL/SC pairs to be nested. *Weak* LL/SC allows spurious failures, prevents nesting of LL/SC instructions, and typically limits the memory operations that can be nested between the pair, with certain operations guaranteed to cause the store-conditional to fail.

LL/SC allows a trivial lock-free implementation of CAS. More importantly, it avoids the ABA problem, simplifying concurrent algorithm design.

Memory-to-memory swap: Atomically swaps the values held in two registers.

DCAS (Double Compare-And-Swap): Returns the values held in two registers, replacing them with new values only if they both match expected values atomically. Once again, DCAS allows a trivial lock-free implementation of any two-location read-and-update primitive.

DWCAS (Double-Width CAS): A DCAS operation, but restricted to operating on a limited set of pairs of registers, namely those pairs which form an aligned double-word in memory. DWCAS is not uncommon on 32-bit architectures with support for 64-bit updates.

Atomic snapshot: Reads multiple locations atomically.

N-register assignment: Writes to multiple locations atomically.

NCAS (N-location Compare-And-Swap): Extends DCAS to cover N locations atomically. NCAS implements an atomic snapshot of N locations if all expected values match the new values. Also abbreviated to CASN, CASn or MCAS in other work.

kCSS (k-Compare, Single-Swap): A restricted form of NCAS which can only update a single location. (I use a small k instead of a capital N to highlight the difference, as NCSS and NCAS are easily confused.)

3.2 Progress

An implementation is *wait-free* if all logical operations complete after a bounded number of (primitive operation) steps. Wait-free algorithms guarantee progress and fairness in the face of an antagonistic scheduler. Wait-freedom dates back as far as 1983 [67].

An implementation is *lock-free* if *global* progress is guaranteed after a thread takes a bounded number of (primitive operation) steps. Individual threads may be indefinitely starved of progress under a lock-free guarantee, provided some thread is making progress. The first appearance of lock-freedom is commonly attributed to a paper by Lamport in 1977 ([50], attribution in e.g. [10]); however, this algorithm was not actually lock-free, as suspension of a writer could prevent progress of concurrent readers. A lock-free set implementation was initially presented in 1988 [52], while the term itself was coined in 1991 by Massalin and Pu [58].

An implementation is *obstruction-free* if a thread executed in isolation (all other threads suspended) will make progress after a bounded number of its

own primitive operations. While obstruction-free algorithms are not new, the term itself was coined in 2003 [42]. An obstruction-free algorithm needs a *contention manager* to achieve reliable progress in the face of contention, as otherwise threads tend to *livelock*, continually blocking each other's progress. More about contention managers can be found in Section 3.8.2

Many older papers have used the term *non-blocking* synonymously with lock-freedom, but non-blocking has since been weakened to include obstruction-free algorithms. In modern usage, therefore, an algorithm is non-blocking if suspension of an arbitrary number of threads cannot prevent progress. This means non-blocking algorithms can be used on preemptive systems, where threads may be suspended at any time for long periods, without negative interactions with the scheduler preventing progress.

Note that, by definition, all wait-free algorithms are lock-free, all lock-free algorithms are obstruction-free, and all obstruction-free algorithms non-blocking.

3.3 Wait-Free Universality

In 1988, Herlihy demonstrated that atomic primitives exhibit a "wait-free hierarchy" [37] The *consensus number* (CN) of a concurrent object is defined as the maximum number of processes for which the object can solve a simple consensus problem. Read-write registers have CN 1; test-and-set, swap and fetch-and-add have CN 2; n-register assignment has CN $2n-2$; and compare-and-swap, LL/SC, and all stronger primitives have a CN of ∞.

He showed that it is impossible to construct a wait-free implementation of an object from objects with a lower consensus number. Thus, read and write registers cannot be used to build any wait-free concurrent object with a consensus number greater than 1, such as a queue or stack (both have CN 2).

Later, Herlihy gave a constructive proof [39] that any object of consensus number n can be used to create a wait-free implementation of any other such object for use by no more than n processes. Thus compare-and-swap, which has consensus number ∞, is *universal*, in the sense that wait-free implementations of any concurrent object can be constructed from it. (Indeed, sticky bits, despite being only tri-valued with weak read-modify-write semantics, are universal as they are just strong enough to implement wait-free consensus [68].)

A universal construction is a technique for converting a sequential (or, more rarely, a lock-based) algorithm into a non-blocking one. Originally intended to prove universality, as with Herlihy's wait-free construction, subsequent research tackled efficiency issues with the intent of creating practical alternatives to traditional mutual exclusion techniques.

3.4 Lock-Free Universality

Herlihy demonstrated a universal lock-free construction based on CAS [38]. Updates atomically swapped a single root pointer from the old version of the object to a new one, preventing disjoint-access parallelism. Memory could be shared between versions to reduce copying overheads. The approach was compared favourably with coarse-grained mutual exclusion, but clearly cannot compete with good fine-grained locking as it must serialize all operations. Reference counting was used to manage memory.

Herlihy subsequently showed how to build a universal construction, in a similar fashion, from any weak LL/SC that can wrap read and write operations [41]. This avoided the need for reference counting, as any update would cause the final SC of all concurrent operations to fail. Once again, this approach is garbage-free and population oblivious, but neither disjoint-access nor read parallel.

Turek *et al.* showed how to use DWCAS to transform any deadlock-free blocking algorithm into a lock-free one [85]. Obstructed threads assist other operations to completion; unfortunately, that means all possible execution paths of a thread must be encoded into a *continuation*, to allow it to be assisted sensibly. The overhead of making and decoding these continuations is not analysed in the paper. The main advantage of this approach is that any disjoint-access parallelism available in the blocking algorithm is preserved in the lock-free transformation.

Alemany and Felten extended Herlihy's methodology [4], avoiding excessive wasted work by maintaining an 'active thread' count per object; a thread attempting to update an object with too many concurrent active threads would yield CPU time to other tasks. To be lock-free, rather than blocking, the method relies on kernel support; when an active thread is suspended by the kernel, all objects it is updating must have their active thread count reduced, allowing other threads to begin operating on them. This approach assumes the asynchrony of the system is bounded, postulating that long delays are solely caused by the scheduler.

Barnes showed how to avoid the copying overheads of Herlihy's algorithm by breaking the shared object into disjoint parts, relying on obstructed threads assisting conflicting operations to achieve lock-freedom [12]. (Herlihy's approach linearizes at a single operation, the update of the root pointer, so threads cannot be obstructed by partially completed operations.) This approach is disjoint-access parallel, garbage-free and population oblivious but not read parallel; it requires strong LL/SC.

3.5 Snapshot Objects

One important problem in concurrent algorithms is designing a large object, typically a shared memory, supporting a *snapshot operation*: an atomic operation

which simply returns the current state of the object.

While I do not build a snapshot object from single-word atomic primitives in this dissertation, the subject is strongly tied to the results of Chapter 4, and so have been presented for completeness.

Lock-based algorithms typically support a trivial snapshot operation: grab every lock, respecting the locking order to avoid deadlock; snapshot the object, while concurrent updates are blocked; release the locks. The problem becomes more difficult — and interesting — when updates cannot be blocked.

Lamport first solved this problem in 1977 [50]. The object is protected by two version counters; the first is incremented before the object is updated, the second after. Readers read the second counter before reading the object, and the first after; if they do not match, an update was in progress at some point during the snapshot, and the reader must retry.

In terms of the scalability properties of Section 2.4, Lamport's algorithm is read parallel and population oblivious. It is not garbage-free, because counter values cannot be reused. If multiple objects are protected by version counters, a combined snapshot can be taken atomically; this extension is parallelism-preserving.

An equivalent algorithm uses just a single version counter, incremented both before and after updating the object. Readers check this counter twice, before and after reading the object; if the counter is odd, or changes during the snapshot, an update was in progress and the reader must retry.

This latter formulation illustrates one problem with this solution: readers must spin indefinitely if an update is in progress. The algorithm is not lock-free or even obstruction-free. Another problem is that the algorithm permits only a single concurrent writer; multiple writers must use a separate mutual exclusion mechanism.

Peterson addressed the first problem in 1983 [67]. By maintaining two main copies of the object, a reader can be sure one will be valid if it takes a snapshot overlapping a single update; by communicating that a snapshot is in progress to the (single) writer, and providing each reader with a buffer for the writer to place a copy of the object's state, the reader can be sure of obtaining a valid snapshot even if it overlaps a sequence of updates.

Peterson's algorithm is wait-free, parallelism-preserving and garbage-free, but not population oblivious. If there are n readers of an object of size k, each update requires $\Omega(k+n)$ and $O(kn)$ operations; the memory requirements are $\Theta(kn)$. It only allows a single writer at a time.

During the late '80s and the '90s, other snapshot algorithms were presented. Often, algorithms were refined in a series of publications, or distributed in unpublished form among researchers before being accepted much later; as such, it is unedifying to examine publication dates. In complexity formulae, k represents the size of the object (number of registers), n the number of readers, and w the number of writers if readers and writers are distinct; all algorithms use only read and write operations unless otherwise stated:

- Anderson presented a multi-reader, multi-writer, wait-free shared memory with snapshot operation; unfortunately, the time complexity of a read is $O(2^{kw})$, and of a write, $O(n + 2^{kw})$, with w the number of writers. The construction is read parallel and garbage-free, but neither disjoint-access parallel nor population oblivious. [8]

- Kirousis *et al.* showed how to construct a single-reader, multi-writer wait-free shared memory with snapshot. The time complexity of a read is $\Theta(kw)$, and of a write, $\Theta(1)$, with w again being the number of writers. The construction is disjoint-access parallel and garbage-free, but neither population oblivious nor, since only one reader is permitted, read parallel. [46]

- Afek *et al.* designed a series of algorithms culminating in a multi-reader, multi-writer, wait-free shared memory with snapshot; all operations are $O(n^2k)$ time complexity. The algorithm is read parallel and garbage-free, but neither population oblivious nor disjoint-access parallel. [2]

- Attiya and Rachman proposed a multi-reader and -writer, wait-free shared memory with snapshot, with all operations of $O(n \log n)$ time complexity. The algorithm is population oblivious, but not garbage-free, read or disjoint-access parallel. [11]

- Anderson presented an improved shared memory with snapshot, also multi-reader and -writer, where the time complexity is $O(n^2k)$. The construct is read parallel and garbage-free, but neither disjoint-access parallel nor population oblivious. [9]

- Riany *et al.* showed that, for the multi-reader, single-writer case, a wait-free algorithm exists with $O(1)$ and $O(k + n)$ running times for write and snapshot, respectively. Their algorithm is disjoint-access parallel, but not garbage-free, population oblivious or read parallel. It also requires LL/SC, or an emulation of it with Compare-and-Swap and timestamps, and Fetch-and-Increment. [76]

Research in this area has also continued into the new millennium:

- Afek *et al.* demonstrated a multi-reader, multi-writer, wait-free shared memory with snapshot, where the time complexity of operations depends on the contention k, the number of threads performing concurrent operations, rather than the total number of threads. Specifically, the time complexity is $O(k^4)$. This algorithm is population oblivious, but not garbage-free (requires unbounded registers), read or disjoint-access parallel. [3]

- Fatourou *et al.* proved that, for $n > k$, implementing a multi-reader, multi-writer wait-free shared memory with snapshot using only k primitive registers (a provably optimal space requirement) imposes a $\Omega(n)$ lower bound on

the scan time [21]. In a subsequent paper, they improved this lower bound to $\Omega(kn)$, matching the best known algorithm [22].

- Jayanti improved the results of Riany *et al.*, showing that a wait-free algorithm with $O(1)$ and $O(k)$ running times for writes and snapshot, respectively, exists in the multi-reader, multi-writer case. Their algorithm requires Compare-and-Swap, and is disjoint-access parallel and population-oblivious. It is not read parallel; neither is it garbage-free, as it must store a unique ID for each reading process. [44]

- Do Ba improved the space complexity of Jayanti's result from $O(kn^2)$ to $O(kn)$, relying on an LL/SC primitive. He also presented an algorithm with $O(k)$ space complexity, $O(1)$ and $O(k)$ running times for writes and scans, respectively, in the absence of contention, using only reads and writes, but only providing an obstruction-free progress guarantee. [20]

3.6 Assistance

To achieve a lock-free or wait-free progress guarantee, threads performing one operation may be required to assist other operations to completion. A simple example of this is found in Peterson's wait-free single-writer multi-reader snapshot object [67]. The writer thread, on detecting a conflict with a concurrent read operation, will assist that read operation by copying a valid snapshot of the object into a per-thread buffer.

This assistance-by-copying is common to many of the snapshot object implementations introduced above, but is insufficient for more complex logical objects, which have a greater range of potentially conflicting, non-idempotent operations that need to be assisted.

Another approach, taken by Barnes' universal transformation [12], is to encode each operation in a *continuation* or *descriptor*. This must contain enough information to allow another thread to complete the operation, such as (in the case of an NCAS operation) a list of memory locations, each with corresponding old and new values. It may also contain information about the current status of the operation, as in Greenwald's Two-Handed Emulation [29].

Key to any assistance-based approach is ensuring the system is deadlock and livelock free. For the snapshot object, this is trivial: reader threads do not assist, so cannot deadlock or livelock; and whenever the writer thread is blocked, there is always an obstructing read operation that can be assisted. General systems are more complex, as an obstructing operation may in turn be obstructed by other operations. A naive approach may result in a ring of operations each obstructing the last, resulting in deadlock.

Barnes solves this by having each operation, in the initial stage of the algorithm, claim each disjoint resource being modified by the operation, following a

pre-defined order. A set of operations cannot mutually obstruct each other during this stage, since by construction one of them must be about to claim an object which none of the others have claimed, so this one can be assisted to completion by the others. Once this stage is over, an operation cannot be obstructed further, so again can be assisted to completion by any obstructed thread.

An alternative is to define a *priority* ordering on the operations themselves, for instance based on the memory location of their descriptors. To allow this, threads must be able to *abort* obstructing operations; whether one operation aborts or assists another is decided by their relative priorities.

Shavit and Touitou argue that *recursive* assistance, where an obstructed thread may have to help a concurrent operation that is not directly obstructing it, is a source of inefficiency [80]. In their alternative, *non-redundant* helping, threads only assisting an operation that directly obstructs them. If that operation in turn is obstructed, the thread aborts it instead of assisting it. Lock-freedom of the system is still guaranteed.

The chief obstacle to high throughput is assistance in general: if one thread attempts to assist another, live thread, the cost of synchronizing the two will dominate the performance. Better average-case throughput can be achieved with a *contention management* scheme, which controls whether a thread attempts a potentially costly interaction with an obstructing operation, or waits for the operation to complete. Such schemes have been investigated in the context of obstruction-free algorithms (see Section 3.8.2). It would be enlightening to see whether these ideas transfer directly to the lock-free domain.

3.7 DCAS

DCAS has often been suggested as a good primitive to implement to allow faster, more scalable implementations of concurrent objects than can be achieved with CAS alone. The first collection of DCAS-based algorithms were presented by Massalin and Pu [58] in 1991: both their LIFO stack and general linked lists required DCAS for thread-safety.

In his doctoral dissertation [28], Greenwald presented several new lock-free algorithms based on DCAS: two stacks, one array-based and consequently fixed-size, one list-based; a FIFO queue; a priority queue; and two fixed-size deques, one which allowed no disjoint-access-parallelism as it stored both head and tail pointer in a single word, and one which has elsewhere been asserted as incorrect [5].

Greenwald also showed how to emulate a lock-free NCAS with DCAS, storing the progress of each NCAS operation in a descriptor, and using DCAS to update the progress counter and the main memory locations atomically. The first half of the NCAS stores a pointer to the log in each of the N memory locations; thus $2N$ DCAS operations are required per successful NCAS. This scheme is disjoint-access-parallel, but not read-parallel even if many of the N locations are

unmodified by the operation.

This method of atomically updating memory with one hand and a shared progress counter with the other, was later presented separately by Greenwald as "two-handed emulation" [29], a universal method of creating lock-free implementations of concurrent objects. The resulting algorithms require modification to achieve good scalability, and as was pointed out in a subsequent paper [19], the techniques for doing so are subtle and complicated. Naive two-handed emulation can be seen as a universality proof for DCAS rather than a practical universal transformation.

Agesen *et al.* have shown two DCAS-based deques [5], one fixed-sized and one dynamically-sized; the latter used two DCAS operations per pop, and reserved a bit in each pointer. Detlefs *et al.* improved the dynamically-sized deque algorithm [17], using one DCAS per uncontended operation and removing the need for the reserved bit, but a later paper [19] demonstrated the algorithm incorrect, and presented a corrected version. An alternative approach allowed memory allocation and reclamation to be aggregated [57]

All of the dynamically-sized DCAS-based algorithms, including the DCAS-based MCAS and two-handed emulation, require garbage collection to reclaim memory. Detlefs *et al.* [18] demonstrated how to use DCAS to implement concurrent reference counting for this purpose; however, the need to update reference counts on every node accessed in an operation denies both disjoint-access- and read-parallelism, and greatly increases the number of atomic operations required.

As has been observed [19], "DCAS is not a magic bullet". Designing efficient and scalable concurrent objects with DCAS, and proving them correct, is non-trivial. Further, as subsequent research has shown, it is often not necessary to demand DCAS to achieve comparable properties for the objects described above.

3.8 Transactions

In 1992 (republished in 1993 [40]), Herlihy and Moss proposed extending processor architectures to support *transactions* on arbitrary memory locations. Threads would compose an atomic transaction using reads and writes, then issue an instruction to hardware to commit the changes made. If the transaction could not be executed atomically, the commit would fail, the changes would be rolled back, and the thread could retry. Failed transactions would have no externally-visible effects.

This approach, called *transactional memory* (TM), is positioned as simplifying concurrent programming — no need to worry about deadlocking or data races — whilst keeping or bettering the best performance of existing concurrent algorithms.

Subsequent research has presented alternative hardware transactional memory designs, software emulation of transactional memory (STM) on existing hardware,

and hybrid approaches. The hardware approaches all support scalable software, while STM proposals sacrifice one or more of the scalable properties I have outlined in Chapter 2.

3.8.1 Transactional Memory

A limited form of transactional memory was proposed in 1986 by Knight for use in "mostly functional programming languages" [47]. Knight's design implemented kCSS rather than NCAS, and relied on a pre-defined commit ordering between transactions. Due to these restrictions, I shall not discuss the details further, except to note that it demanded a fully-associative cache to avoid conflict misses.

The first proposal for composing arbitrary transactions in hardware was by Herlihy and Moss in 1992, as mentioned above. By extending the coherency protocol of the memory subsystem (Figure 3.1), Herlihy and Moss could guarantee lock-freedom given certain restrictions on the set of valid transactions: namely, that the entire transaction fits into a cache, designed for the purpose, occurs within a single scheduling quantum, and attempts to gain ownership of each memory location in a predefined order. Given a reasonable quanta and cache, this would allow the construction of NCAS for some architecture-specific N.

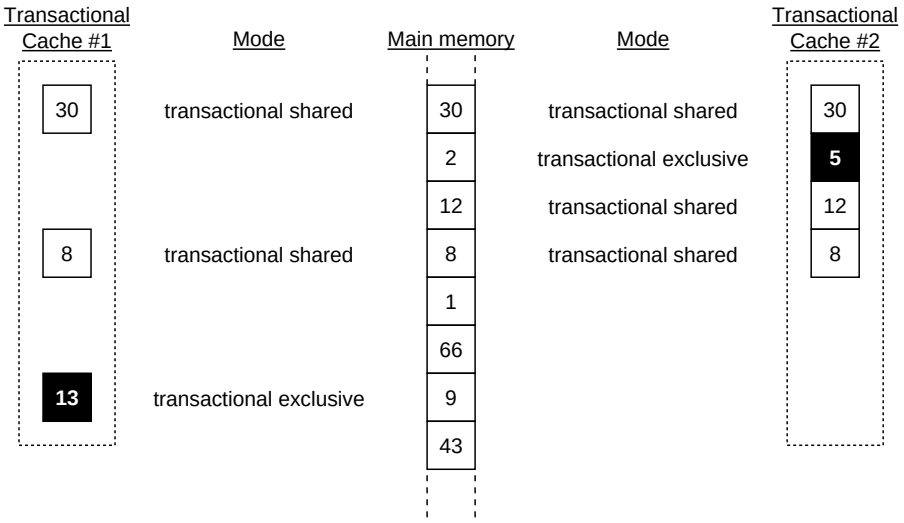


Figure 3.1: Transactional memory on a machine with two processors. Memory accessed during a transaction is held in one of two 'transactional' states. Both caches may hold a copy of a cache line (here depicted as holding a single value) in shared mode, but only one can hold exclusive mode on a line at any one time. A transaction will abort rather than update a line held in the other cache, or read a line held in exclusive mode by the other cache.

This decomposition of transactions into memory reads and writes allows sen-

sible pipelining on modern processors, and does not complicate the register file. This is a significant benefit, especially on RISC processors, where implementation is a major factor in instruction set choice. TM also preserves disjoint-access and read-parallelism, key factors in allowing scalable algorithms to be built from it.

There are obstacles to the adoption of this *transactional memory* as originally proposed. A new inter-chip coherence protocol prevents the adoption of proven memory subsystem hardware, and the hard limit on transaction sizes prevents TM being blindly used to protect critical sections in the stead of traditional mutual exclusion. Further, TM, despite being intended for implementing lock-free data structures, is not lock-free in the general case. The policy of aborting a transaction that tries to revoke ownership of another active transaction unfortunately admits livelock, as the aborted transaction may restart and cause the abortion of the other transaction if memory locations are not modified in some global order.

Rajwar and Goodman proposed Transactional Lock Removal (TLR, [72]), combining earlier work, Speculative Lock Elision (SLE, [71]), with timestamp-based transactional execution. This involves radical changes throughout the hardware, but no changes to the instruction set, instead relying on heuristics to determine when locks are held and released. Like TM, transactions must fit in the cache and complete within a quantum; otherwise the locks will not be elided and the execution becomes blocking. Unlike TM, the use of timestamps prevents starvation when TLR is successful.

TLR, as with traditional mutual-exclusion approaches to thread-safety, may force on the programmer an awkward choice between coarse-grained and fine-grained locking. If the critical section can be executed in a single transaction, coarse-grained locking achieves the best performance, as it minimises overhead. If the critical section is frequently executed by holding the lock, fine-grained locking will produce better scalability.

In his Master's thesis, Lie proposed an *unbounded* transactional memory (UTM04, [54]). Unlike TM, transactions could access an arbitrary data set and run for an arbitrary length of time. Transactions which overflow their cache or quanta spill into uncached main memory, where a hash table effectively extends the transactional cache at the cost of performance. This frees the programmer from worries about transaction sizes.

UTM04 also assumes a standard coherency protocol, simplifying the task of the hardware architect, but resulting in an obstruction-free design that cannot be made lock-free even with careful ordering of memory accesses.

Hammond *et al.* took an alternative approach, called Transactional memory Coherence and Consistency (TCC, [33]). Their design stores transactional updates locally on the processor cache, as with TM, but transmits the updates atomically over the memory bus on commit, rather than negotiating for exclusive access to each cacheline individually. This avoids problems of livelock, yielding a lock-free progress guarantee, but limits scalability, as supporting one-to-all

broadcast on large numbers of processors has not historically been feasible.

The main objection that could be made to transactional memory at the time it was proposed was the hardware cost: silicon that a transactional cache would require was in great demand for larger regular caches. Modern chips, however, have a much greater silicon budget, and with multiprocessing becoming the norm even on cheap commodity hardware, transactional memory is now a much more compelling idea. In the last two years (2005–06), therefore, there has been a significant body of material published on transactional memory; I will cover the major hardware proposals in chronological order.

- Ananian, Lie *et al.* presented another unbounded transactional memory (UTM05, [7]). This emulates a more complex coherency protocol in main memory, using timestamps to resolve conflicts, giving priority to older transactions. In the common case of small, uncontended transactions, a transactional cache avoids the need to write to main memory, avoiding severe performance penalties. However, cache misses always require a read of main memory, even for non-transactional reads and writes.

  UTM05 is a blocking implementation, as a switched-out thread's transaction will block all subsequent transactions that contend with it. It works with standard memory buses and RAM modules, but demands substantial changes to the caching system and main processor design.

- Moore *et al.* describe an unbounded abstraction, Thread-Level Transactional Memory (TTM, [64]), which uses a per-thread log to allow rollback in the event of aborts of overflowed transactions. Their abstraction presents a well-defined interface to the user, but admits a wide variety of implementation strategies. They present two such implementations for broadcast and directory coherence protocols; the former detects conflict pessimistically for overflowed transactions, reducing performance but maintaining correctness, on the assumption that transactions only rarely overflow; the latter demands an extension of the directory protocol to support overflowed transactions. It is unclear whether TTM allows transactions to overflow scheduling quanta: the implementations do not appear to distinguish a thread from a processor, suggesting not.

- Rajwar *et al.* proposed Virtual Transactional Memory (VTM05, [73]), another combination software/hardware solution. They assumed an existing bounded hardware implementation of transactional memory, and described an extension built on top that allows transactions to overflow in time and space. As with UTM05, they implement a more complex coherency protocol, but use cacheable memory, and optimize the common case of no contention using Bloom filters [14]. Standard memory buses and RAM can be used.

- In his Master's thesis [83], Sukha suggested combining transactional memory with memory-mapped I/O: once the file is loaded into memory, concurrent threads and even concurrent processes could use transactional memory to update the file. This could greatly simplify programs that require concurrent I/O, e.g. databases, without sacrificing their scalability.

- Vallejo *et al.* described how to execute critical sections in a transactional manner on specific hardware, the 'Kilo-Instruction Multiprocessor' [86]; as with TLE, this silently executes lock-based code transactionally.

- McDonald *et al.* produced a detailed comparison of TCC versus traditional snoopy coherency protocols [59], concluding that the overhead of TCC was acceptably small even for optimized parallel programs. They also claimed that certain hardware decisions, such as adding a victim cache, could ensure TCC provided acceptable performance for most applications.

- Moss and Hosking considered how to model nested transactions [66], concluding that there may be performance gains in allowing sub-transactions to commit before their parent completes, as fewer transactions will have to rollback due to (logically) false conflicts.

- Chou *et al.* demonstrated that TLE can improve the performance of a single thread by allowing the latency of a write missing in the cache to be hidden by the execution of subsequent instructions [15].

- Moore *et al.* presented LogTM [65], which stores new values while a transaction is running, writing back the old values from a cached log in the event of a conflict. LogTM requires some changes to the memory subsystem, such as allowing a processor to evict a cacheline involved in a transaction. By allowing a software trap-handler to manage rollbacks in the event of contention, LogTM progress can be either obstruction-free or blocking.

- Grinberg and Weiss showed that transactional memory implementations can be investigated using field-programmable gate arrays, allowing much faster analysis than software emulations [30].

- Chung *et al.* analysed the transactional behaviour of thirty five multi-threaded programs from a range of application domains [16]. They observed that most transactions are short, and very few overflow the second level of cache, strongly suggesting that short transactions should be supported directly by hardware, while longer ones could be managed by software. I/O operations within transactions are rare, and the observed patterns are easy to handle through buffering techniques, without demanding hardware support. Nested transactions occur mostly in system code, and limited hardware support is thus likely to be sufficient.

- McDonald *et al.* proposed complex additions to existing transactional interfaces to allow transactions to include such features as library calls, conditional synchronization, system calls, I/O and even runtime exceptions [60].

- Ramadan *et al.* analysed how to use transactions in the Linux kernel [74]. They suggested changes to existing transactional memory models that could ease this process, such as supporting nested transactions for interrupts, and allowing the kernel to provide hints about conflict management priorities.

3.8.2 Software Transactional Memory and NCAS

Software Transactional Memory (STM) was first proposed in 1995 by Shavit and Touitou [80]. Unlike universal constructions, which take serial (or lock-based) code and apply a programmatic transformation, an STM provides an abstraction for writing concurrent non-blocking algorithms directly: namely, as with Herlihy and Moss' transactional memory, wrapping memory accesses into a transaction, and retrying the operation if the transaction fails.

I will now cover subsequent work in this area, but first a few general points. I cover NCAS implementations here as well, as they are in fact STM implementations. All the algorithms in this section rely on *descriptors*: sections of shared memory that describe an operation in progress, allowing other threads to assist (or retard) its progress. Unlike hardware transactional memory, STMs to date do not provide all four scalability guarantees; typically, they rely on an out-of-line garbage collection scheme, and so are not garbage-free.

Shavit and Touitou's STM emulates the ownership protocol of memory subsystems. Each transaction attempts to gain exclusive ownership of each word it will use, and backs off if it encounters contention. To prevent deadlock, a transaction will then assist the obstructing transaction until it completes or backs off in turn, before trying again. To prevent livelock, each transaction gains ownership of its words in a globally-used order, ensuring that two transactions cannot obstruct each other and both abort.

The implementation relies on an LL/SC primitive that can wrap reads, and reserves an ownership location for each word that may be involved in transactions. Transactions acting on disjoint locations do not interfere, but two operations cannot own the same location concurrently; thus the algorithm is disjoint-access parallel but not read parallel

In his thesis, Greenwald described an NCAS-on-DCAS emulation which can be seen as an improvement on this algorithm: by combining ownership and storage into the same word, it reduces the overhead to just one bit. The main costs are the need for DCAS and garbage collection.

In 2002, Harris, Fraser and Pratt published a new NCAS implementation [35] with similar properties to Greenwald's. However, theirs incorporated a restricted emulation of DCAS from CAS, and hence removed the need for DCAS, or even

LL/SC. The restricted DCAS is achieved by first publishing a DCAS descriptor in one location, validating another, then writing a new value over the descriptor. As such, it emulates a read-wrapping LL/SC rather than a full DCAS, and systems that provide a native read-wrapping LL/SC can perform this operation directly for a modest performance improvement.

The NCAS is then implemented by publishing an NCAS descriptor at each location involved, allowing concurrent operations to assist the NCAS to completion. As with Shavit and Touitou's STM, deadlock is prevented by assigning a global ordering in which memory locations are obtained. This algorithm is also disjoint-access parallel but not read parallel, and demands garbage collection to reclaim the descriptors used.

This showed that neither LL/SC nor DCAS were necessary for low-overhead disjoint-access parallel lock-free algorithms. However, read-parallelism was still missing: cases like binary trees, where the lack of read-parallelism would result in all operations being serialized, still required a complex algorithm to achieve scalability, even when relying on an emulated NCAS.

In 2003, Herlihy *et al.* presented an *object-based* software transactional memory (DSTM, [43]). This provides an abstraction at the granularity of objects rather than machine words, demanding that shared objects be accessed via the STM interface but subsequently allowing direct access to the elements of the object. They discarded lock-freedom, choosing to implement an obstruction-free STM; progress in general is then the responsibility of a contention manager. Crucially, their approach admitted read parallelism; using it, a programmer could achieve similar scalability to a hand-coded algorithm without, as Herlihy once put it, "ending up with a publishable result" [41].

In his thesis, Fraser demonstrated a lock-free object-based STM that also preserved read parallelism, building upon the earlier NCAS design. Livelock could not be prevented with any global ordering of locations, as two operations with disjoint update sets but overlapping footprint could still deadlock at the final, read-only stage of commit. Instead, the operations themselves were ordered, using the address of the main descriptor; lower-priority operations could be rolled-back by higher-priority ones to achieve progress.

Both read-parallel STM implementations rely on garbage collection of objects as well as descriptors. Read parallelism is achieved by locating all non-updated objects immediately prior to updating a status field in the transaction descriptor. Object updates involve copying the contents to a new chunk of memory, and hence objects will always change location when they are updated, allowing the transaction to commit if it sees none of the objects have moved. This, of course, relies on limited memory reuse, and hence garbage collection. Every transaction must allocate memory, and thus has an aggregate cost dependent on the collector used. It has been noted elsewhere [55] that this can have detrimental effects on performance unless the collector is (and can be) chosen appropriately.

Subsequent work can be divided into related groups. The first is the devel-

opment of sophisticated contention management strategies for Herlihy *et al.*'s DSTM:

- Scherer and Scott described a range of contention managers in April 2004: 'Aggressive' always aborts obstructing transactions; 'Polite' retries up to eight times, with backoff periods growing exponentially, before aborting an obstruction; 'Randomized' decides randomly whether to abort or backoff; 'Karma' stores how much memory a transaction has touched as a priority scheme, waiting longer for long-running transactions; 'Eruption' extends Karma, adding the priority of blocked transactions to the transaction blocking them; 'Kill-blocked' always aborts transactions if they are blocked, otherwise it waits and aborts them after a maximum waiting time; 'Kindergarten' allows each thread to block a transaction for a short time, but then repeatedly aborts that thread if it ever blocks the transaction afterwards; 'Timestamp' allocates timestamps to each transaction and prioritizes old transactions, as well as using a 'defunct' flag to allow a slow transaction to tell faster ones it is still running; finally, 'Queue-on-block' maintains a notification queue for each transaction of threads blocked by it, allowing them to be notified when the transaction terminates, but rendering the manager susceptible to dependency cycles. [77]

  Despite the large number of policies tried, every single policy was found to perform abysmally in some benchmark, though Karma and Polite were frequently among the best performers.

- In July 2005, Scherer and Scott presented two new contention managers [78]. 'Published-timestamp' improves the original timestamp manager by having active transactions periodically publish a 'recency' timestamp; this allows preempted transactions to be rapidly aborted without the overhead of the 'defunct' flag.

  'Polka' combines Polite's randomized exponential backoff with Karma's priority accumulation. It backs off for a number of intervals equal to the difference in priorities between the transaction and its obstruction, and the length of these backoff intervals increase exponentially. The former minimizes wasted work due to conflict, while the latter minimizes coherency traffic costs.

  Managers were tested on several benchmarks, including write-dominated workloads where all transactions conflict, and a red-black tree where much parallelism could potentially be exploited. The Polka policy was found to achieve top or near-top performance in all benchmarks, and was recommended as a default setting for a software transactional memory.

- Scherer and Scott have also experimented with randomizing various aspects of the Karma contention manager: randomizing the exponentially-growing

backoff periods; randomizing the number of backoffs before aborting an obstructing transaction; and randomizing the gain in priority of each step of the transaction. They found that in every benchmark there was some combination of randomization that improved performance. [79]

- Guerraoui *et al.* presented a timestamp-based 'greedy' manager, with provable worst-case throughput in a model with finite transaction delays. In a model with unbounded delays, or thread failures, their manager is blocking. [31]

- Fich *et al.* proved that, under a *semi-synchronous* model, where there is a bound on the number of concurrent operations that can execute between any two consecutive instructions issued by a single thread, but where that bound is not known, with an appropriate choice of contention manager, an obstruction-free algorithm can in fact be wait-free even in the face of thread failures. Further, their approach allows the use of a standard contention manager except in exceptionally unfair schedules, when a thread that is not making progress can raise a 'panic' flag to ensure it ultimately completes. This means a contention manager with good throughput can be chosen without sacrificing wait-freedom. [25]

- Guerraoui *et al.* improved their greedy contention manager, with provable worst-case throughput even with thread failures. Unlike Fich *et al.*, they again provided a quantitative bound on throughput. [32]

3.8.3 Hybrid Transactional Memory

Some proposals have recommended hybrid approaches to transactional memory, where the hardware provides some of the machinery necessary to implement transactions, and the rest is done by a software library. This reduces the required hardware complexity without passing on artificial constraints to the programmer, and even allows successive generations of an architecture to vary the complexity of their transactional hardware with only a single library rewrite.

Kumar *et al.* presented Hybrid Transactional Memory (HybTM, [49]), combining a bounded transactional memory with a modified version of Herlihy *et al.*'s DSTM, exploiting the separation of correctness from progress in the latter to allow hardware transactions to coexist with software ones. Unfortunately, there is no way of 'spilling' a transaction from hardware to software if the transaction overflows the hardware limits. In all such events, the transaction must be explicitly retried in 'software mode', which does not benefit from the hardware support. The proposal is not scalable, as the DSTM it is built on is not garbage-free.

Shriraman *et al.* went further (RTM, [81]), making the transactional hardware entirely dependent upon their STM, and even allowing the STM to control what memory should and should not be managed by the hardware. By allowing two

transactional caches to speculatively update the same memory concurrently, and placing conflict management entirely in the hands of the STM, their design can dynamically choose between various management strategies. As with other proposals, however, their design cannot achieve lock-freedom, since there is no means of assisting concurrent operations. It cannot be used without a special software layer, even for small transactions, as conflict management is not provided by the hardware.

In conclusion, transactional memory as initially proposed by Herlihy and Moss has led to a wide range of designs. Hardware designs are inherently scalable, as defined in Section 2.4, but cannot provide strong progress guarantees such as lock-freedom without radical hardware changes. Further, research on contention management strongly suggests that obstruction-free algorithms do not provide good throughput in a wide range of benchmarks if obstructed threads cannot obtain information about what operation is blocking them, information that cannot reliably be exchanged using only obstruction-free primitives.

Software implementations of transactional memory greatly simplify the creation of practical non-blocking algorithms on current architectures. Recent research has provided compelling evidence that such designs can provide strong performance, and handle contention without severe slowdowns. However, no proposal provides all four scalability properties. In Chapter 4, I show that this is a consequence of building upon existing primitives.

Perhaps the most compelling approach is a hybrid one: a library implementing transactional memory in software, building on the primitives provided by the hardware. HybTM and RTM are examples of such an approach; however, they do not overcome the lack of guaranteed progress in pure-hardware designs. This raises a question, which I address in Chapters 6 and 7: is there a primitive which allows a scalable STM to be built, yet also has a practical, lock-free hardware implementation?

Chapter 4

CAS is not Scalably Universal

In this chapter, I prove some constraints on what can be scalably implemented from a given primitive object. The goal is to determine whether or not a shared memory with CAS operations can scalably implement one with DCAS, and whether either can implement transactional memory; the conclusion is that they cannot.

4.1 Definitions

Before starting on the arguments proper, I need to introduce some terms. The model introduced in Chapter 2 is intentionally general, to allow any shared object and any implementation to be described. I now define some properties found in many shared objects such as shared memories.

Orthogonality. An *orthogonal type* $\mathbb{T}$ must have the following properties: the footprint of an event depends only on the operation it performs; if the value returned by a read operation r is changed by an update p, it cannot be changed back by any number of subsequent disjoint modifications; finally, all operations must have finite footprints. In particular, shared memories supporting read, write and CAS operations are orthogonal.

$$u_E = u_{E'} \Rightarrow \left.\begin{pmatrix} f_{\mathbb{T}}(E) = f_{\mathbb{T}}(E') \\ f^m_{\mathbb{T}}(E) = f^m_{\mathbb{T}}(E') \end{pmatrix}\right\} \forall E, E' \in \mathsf{Events}$$

$$\left.\begin{array}{c} p \circ s \neq s \\ f^m_{\mathbb{T}}(p) \not\subseteq \cup_i f^m_{\mathbb{T}}(p_i) \\ f_{\mathbb{T}}(r) \cap f^m_{\mathbb{T}}(p) \neq \emptyset \end{array}\right) \Rightarrow r(p_n \cdots p_1 \circ p \circ s) \neq r(s) \quad \left.\begin{array}{l} \forall p, p_1 \ldots p_n \in \mathbb{O}_{\mathbb{T}} \\ \forall r \in \mathbb{R}_{\mathbb{T}} \\ \forall s \in \mathbb{S}_{\mathbb{T}} \end{array}\right.$$

$$|f_{\mathbb{T}}(E)| < \infty \quad \forall E \in \mathsf{Events}$$

As footprints in an orthogonal type depend only on the operation being performed, I can define the footprint of an operation:

$$\left.\begin{array}{l} f_{\mathbb{T}}(u_E) \stackrel{\text{def}}{=} f_{\mathbb{T}}(E) \\ f^m_{\mathbb{T}}(u_E) \stackrel{\text{def}}{=} f^m_{\mathbb{T}}(E) \end{array}\right\} \forall E \in \mathsf{Events}$$

A set of operations S executes in parallel if none of them communicate:

$$\forall S \subseteq \mathbb{O}_{\mathbb{T}}, S \rightrightarrows_{\mathbb{T}} \stackrel{\text{def}}{\iff} \forall p, p' \in S \; (p \neq p' \Rightarrow f^m_{\mathbb{T}}(p) \cap f_{\mathbb{T}}(p') = \emptyset)$$

> For an example of a non-orthogonal type, consider a single register with read and add operations. This can be modelled by allocating a single synchronization point for each thread; add operations update the synchronization point of the thread doing the add, while read operations have the entire set of synchronization points as a footprint. Thus, add operations can execute in parallel, and read operations can execute in parallel, but any add/read pair must communicate.
>
> By insisting upon orthogonality, I prevent add operations from executing in parallel: two add operations executed by different threads can cancel each other out, so, by the definition of orthogonality, would have overlapping modification footprints.
>
> See Section 4.4 for another example of a non-orthogonal type.

Inverses. Type $\mathbb{T}$ is said to *have inverses* if any operation can be undone by a single subsequent operation with the same footprint:

$$\forall s \in \mathbb{S}_{\mathbb{T}}, u \in \mathbb{O}_{\mathbb{T}}, \exists u^{-1} \in \mathbb{O}_{\mathbb{T}} \text{ s.t. } \left\{\begin{array}{c} u^{-1} \circ u \circ s = s \\ f_{\mathbb{T}}(u) = f_{\mathbb{T}}(u^{-1}) \\ f^m_{\mathbb{T}}(u) = f^m_{\mathbb{T}}(u^{-1}) \end{array}\right.$$

> This again includes shared memories with any of the primitives discussed in Chapter 3. For instance, a write can be undone by writing the old value back into the register.
>
> Note that this does not demand that all operations have a *single* inverse operation regardless of the starting state. Since write operations are not injections, this would be impossible for any shared memory.

Completeness. Type $\mathbb{T}$ is *complete* if any state can be reached from any other state with a single operation:

$$\forall l, l' \in \mathbb{S}_{\mathbb{T}}, l \neq l', \exists u \in \mathbb{O}_{\mathbb{T}} \text{ s.t. } u \circ l = l'$$

While shared memories are not complete, individual registers are: any state can be reached in a single operation by simply writing in the new value.

Determinism. I wish to be able to construct new histories by reordering events in existing ones. In general, however, the theory of shared objects makes no guarantees that any reordered history is valid — that is, could be observed by some interaction with the shared object in question. I therefore define determinism, allowing reordering arguments to be made.

A type is *deterministic* if the outcome of an operation in a sequential history depends only on the state of the object. This holds for all primitives introduced in Chapter 3. This property may have been implied by the language used in Chapter 2, but the theory can in fact be built up without it. I now eliminate such pathology from consideration.

In particular, assuming a deterministic primitive allows the behaviour of one history to be determined by considering that of a sequentially consistent one.

Simple examples of non-determinism are timeouts and infinite clocks. A timeout allows a heavily-delayed thread to abort and retry, limiting the effects of an antagonistic scheduler. In particular, this requires knowledge about how rapidly concurrent events can be scheduled; without this, a timeout can be reduced to a deterministic primitive by simply scheduling all events before any timeout can elapse. Infinite clocks do not exist in practice, and all finite clocks can be reduced to a deterministic primitive by scheduling all events to occur with the same frequency with which the clock overflows.

Invisible reads. An implementation $\mathbb{M}$ of $\mathbb{L}$ from $\mathbb{P}$ *has invisible reads* if, for all read operations $r \in \mathbb{R}_{\mathbb{L}}$, any (finite) history H can be extended with a single event E executing r in a new thread such that the new event does not update any synchronization point found in any footprint of any event of H, i.e.

$$\left.\begin{array}{c} f^{m}_{\mathbb{P}}(E) \cap f_{\mathbb{P}}(A) = \emptyset \\ E \overset{\mathrm{t}}{\not\sim} A \end{array}\right\} \forall A \in H$$

Strategies for implementing mutual exclusion do not typically have invisible reads, as update operations must communicate with conflicting read operations. If they are invisible, reads will typically rely on a non-repeating value being stored in some synchronization point to verify they did not conflict with an update — an approach that, while scalable in other respects, is not garbage-free.

Sequentially-reachable states. Primitive state $p \in \mathbb{S}_{\mathbb{P}}$ is *sequentially-reachable* if there exists a sequential history of implementation $\mathbb{M}$ ending in state p. Sequentially-reachable states always encode a unique logical state, there being no choice how to linearize the history. The following proofs are simplified by the fact that only sequentially-reachable states need to be considered.

Maximal disjointness. To prove results about disjoint-access parallelism, I introduce maximal disjointness, which characterizes the range of granularities of operations provided by a shared object. The *maximal disjointness* of orthogonal type $\mathbb{T}$ is the number of disjoint updates which can execute in parallel that nevertheless all conflict with a single read:

$$\mathcal{D}(\mathbb{T}) = max \left\{ |S| : \begin{array}{l} S \subseteq \mathbb{O}_{\mathbb{T}} \setminus \mathbb{R}_{\mathbb{T}} \\ S \rightrightarrows_{\mathbb{T}} \\ \exists r \in \mathbb{R}_{\mathbb{T}}, l \in \mathbb{S}_{\mathbb{T}} \text{ s.t. } r(p \circ l) \neq r(l)\ \forall p \in S \end{array} \right\}$$

> Maximal disjointness characterises the size of read operations provided by a primitive. In a shared memory with single-register operations, two writes execute in parallel only if they update disjoint registers, and thus cannot conflict with the same read operations; the maximal disjointness is 1. A DCAS operation would increase this to 2, as a DCAS operation can be a read operation if the swapped values equal the compared values, allowing a snapshot of two locations to be taken. Transactional memory [40] has theoretically unbounded maximal disjointness, as each register can be disjointly updated, yet one operation can take a snapshot of an arbitrary number of register.

4.2 Scalability and Disjointness

Lemma 4.2.1 *A population-oblivious, read parallel implementation of a shared object built from an orthogonal, deterministic type must have invisible reads.*

Proof Consider an arbitrary finite history $H \in \mathbb{H}_t$, any t, and let $x = |\cup_{E \in H} f_{\mathbb{M}}(E)|$. Since the primitive is orthogonal, $x < \infty$. By population obliviousness, $H \in \mathbb{H}_{t+x+1}$, i.e. we can add $x+1$ threads to the pool without affecting the total footprint of any event in H. We now schedule each of these $x+1$ threads to execute read operation $r \in \mathbb{R}_{\mathbb{L}}$. By read parallelism, each thread must have a disjoint modification footprint, as no updates are in progress (since H is a history); thus one of these read events, E' say, must satisfy $f^m_{\mathbb{M}}(E') \cap (\cup_{E \in H} f_{\mathbb{M}}(E)) = \emptyset$, i.e. $f^m_{\mathbb{M}}(E') \cap f_{\mathbb{M}}(E) = \emptyset\ \forall E \in H, E \neq E'$. By determinism, $H\langle E' \rangle$ is a valid history, as desired.

Lemma 4.2.2 *The maximal disjointness of a garbage-free, population-oblivious, read parallel implementation of an orthogonal object with inverses is at most the maximal disjointness of the primitive it is built from, if that primitive is orthogonal and deterministic.*

Proof I prove the lemma by first constructing a sequence of history fragments which can be run serially in any combination; by executing these fragments during a concurrent read operation, I then show that assuming the primitive has lower maximal disjointness than the implemented object leads to a contradiction.

Suppose there exists a logical state l, a read operation r, and n update operations $o_1 \dots o_n$ s.t. $\{o_1 \dots o_n\} \rightrightarrows_{\mathbb{M}}$ and $r(o_i \circ l) \neq r(l)\ \forall i$ (Figure 4.1). Let $l_i = o_i \circ l\ \forall i$.

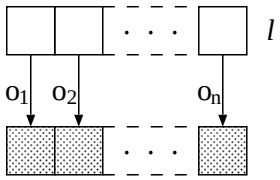


Figure 4.1: Starting from logical state l, n disjoint update operations $o_1 \dots o_n$ each update a different register in a shared memory.

Consider a sequential history H ending in logical state l and some sequentially-reachable state p. I wish to extend this history with a particular series of fragments of the n disjoint update operations and their inverses. First, let E_i be an event executing operation o_i from starting state p, ending in logical state l_i and some sequentially-reachable state p_i, and let G_i be a history fragment extending HE_i by applying o_i^{-1} then repeatedly applying $o_i^{-1} \circ o_i$ some finite number of times to return to state p.

Such H, p, (E_i) and (G_i) must exist by garbage-freedom, else one could extend any sequential history H ending in logical state l to an infinite sequential history H^∞ by applying some sequence of o_is and o_i^{-1}s such that each sequentially-reachable state representing l_i was unique, yet the history passes through only finitely many logical states.

I define fragments built from these events and fragments as follows:

$$\begin{aligned} F &\stackrel{\text{def}}{=} \langle E_1 \rangle \\ F^{-1} &\stackrel{\text{def}}{=} G_1 \\ F_1 &\stackrel{\text{def}}{=} \langle\rangle \\ F_1^{-1} &\stackrel{\text{def}}{=} \langle\rangle \\ F_i &\stackrel{\text{def}}{=} \langle E_i \rangle G_1\ \forall i > 1 \\ F_i^{-1} &\stackrel{\text{def}}{=} \langle E_1 \rangle G_i\ \forall i > 1 \end{aligned}$$

I can now move between the sequentially-reachable states $p_1 \ldots p_n$ by executing a sequence of these history fragments (Figure 4.2). By determinism, given any sequence $\overline{v} \in [1, n]^k$, $HFF_{v_1}F_{v_1}^{-1} \cdots F_{v_n}F_{v_n}^{-1}$ will be a valid history, and by orthogonality, any such extension to H will never pass through state l.

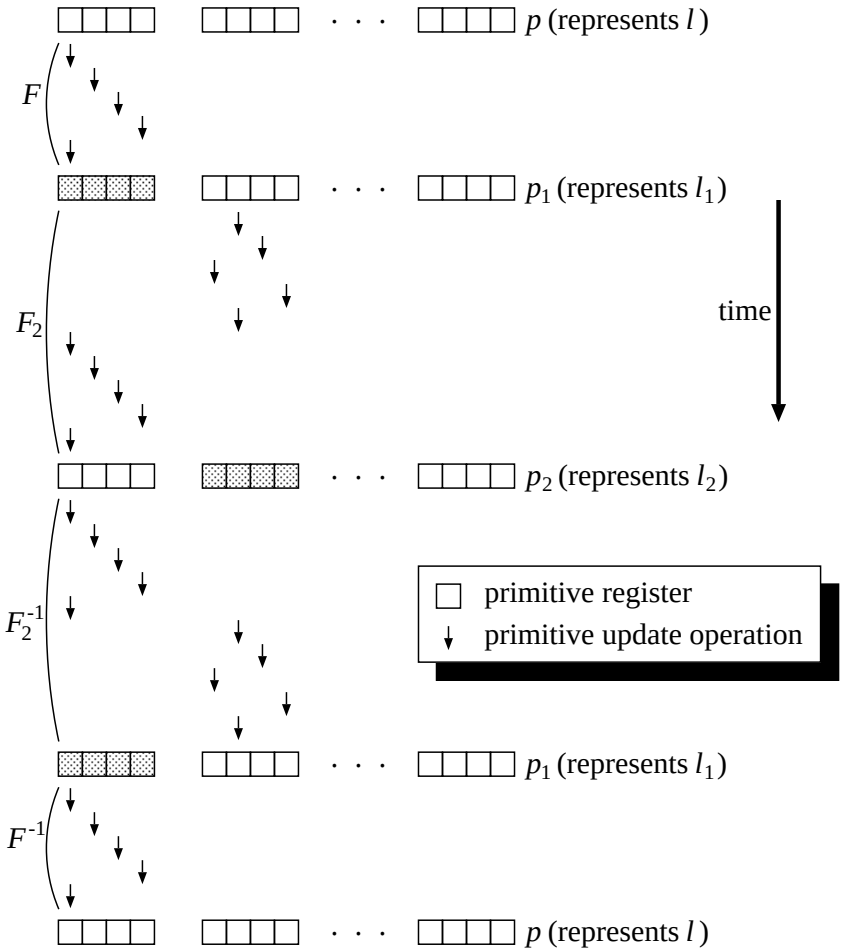


Figure 4.2: History fragments $F_1 \ldots F_n$ allow the history HF to be extended to reach any of the sequentially-reachable states p_i without returning to logical state l.

By Lemma 4.2.1, since the implementation is read parallel and population oblivious, it must have invisible reads. The history $HFF_1F_1^{-1} \cdots F_nF_n^{-1}F^{-1}$ can therefore be extended by a history fragment G, consisting of a single logical event E s.t. $u_E = r$, and $\forall A \in HF_1 \cdots F_n$, $f_{\mathbb{M}}^m(E) \cap f_{\mathbb{M}}(A) = \emptyset$, and $E \overset{t}{\not\sim} A$. This fragment consists of the execution of a series of operations $r_1 \ldots r_k \in \mathbb{O}_{\mathbb{P}}$. (Figure 4.3.)

Since $f_{\mathbb{M}}^m(E) \cap f_{\mathbb{M}}(A) = \emptyset \ \forall A \in HF_1 \cdots F_n$, determinism implies that if G

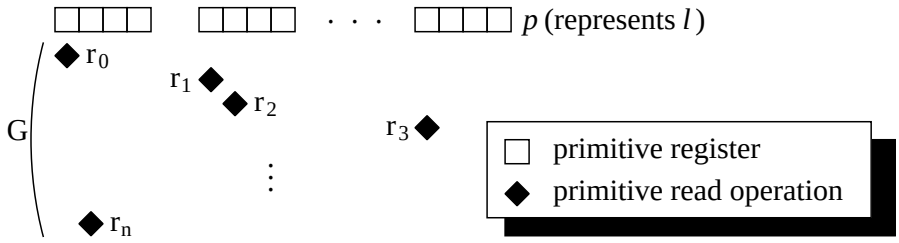


Figure 4.3: History fragment G executes a single read operation, r, on logical state l (represented by sequentially-reachable state p).

can be scheduled during some composition of the history fragments defined above such that all operations $r_1 \dots r_k$ return the same values as in the history G was originally scheduled in, then a history following this schedule *is* a valid execution of the implementation.

I now assume the lemma is false, and derive a contradiction. Suppose $\forall i \in [1,k]$, $\exists j_i$ s.t. $r_i(p_{j_i}) = r_i(p)$, and consider scheduling E during the history $HFF_{j_1}F_{j_1}^{-1} \cdots F_{j_k}F_{j_k}^{-1}$ such that each step i is executed immediately after the corresponding fragment F_{j_i} (Figure 4.4). By determinism and by construction, in such a schedule, E would comprise the same primitives, r_i, and would return the same value as in the history H. Hence, this schedule is a valid history of the implementation, and E must return the same result as in the history H; yet in the latter E runs on state l, which by construction and by orthogonality means it cannot return the same value as in the former schedule.

Hence the supposition must be invalid, and $\exists i \in [1,m]$ s.t. $r_i(p_j) \neq r_i(p)$ $\forall j = 1 \dots n$, and r_i does not execute in parallel with any $F_j, j = 1 \dots n$; hence the maximal disjointness of $\mathbb{P}$ must be at least n. The lemma follows.

Theorem 4.2.3 *No scalable implementation of DCAS exists from CAS; nor of (N+1)CAS from NCAS; nor of transactional memory from CAS, DCAS or NCAS.*

Proof CAS has a maximal disjointness of 1, DCAS of 2 and NCAS of N; transactional memory has theoretically unbounded disjointness. All variants of CAS are orthogonal, deterministic and have inverses. The theorem thus follows directly from Lemma 4.2.2.

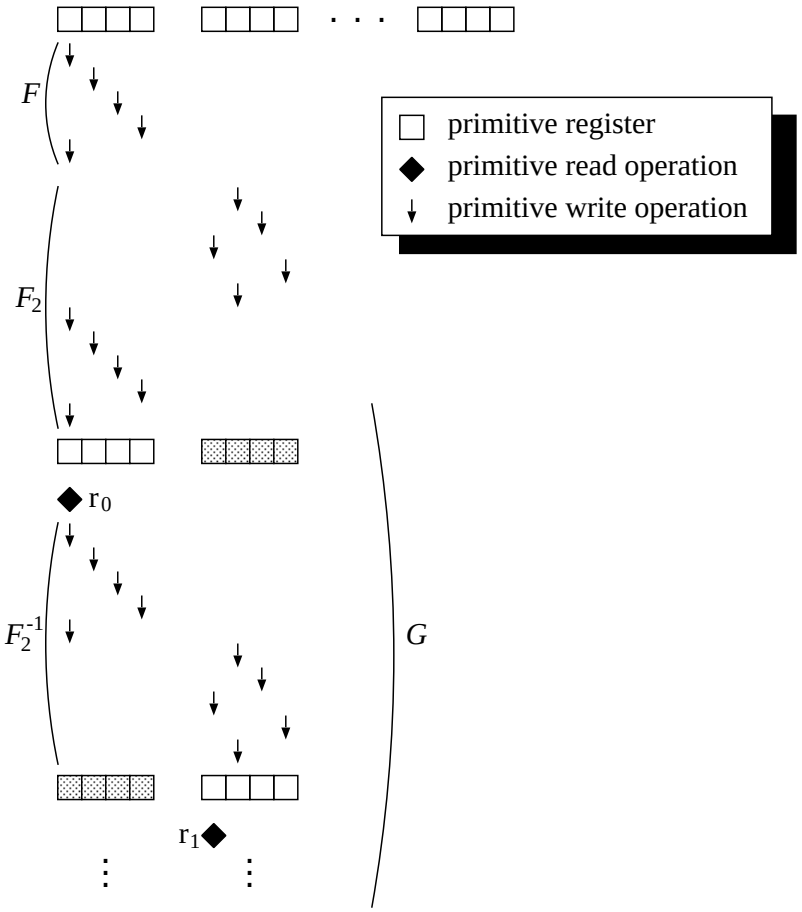


Figure 4.4: History fragment G scheduled during a history chosen such that each r_i returns the same value, yet the history is never in logical state l during r's execution.

4.3 Scalability and Large Snapshots

Theorem 4.2.3 shows that CAS cannot scalably implement DCAS, as the maximal disjointness of the latter is greater than that of the former. This leads to another question: can CAS scalably implement a wider CAS operation, DWCAS? Since the maximal disjointness of both is the same, the arguments above do not apply.

It is indeed possible to build a simple, scalable, blocking implementation of a $2n$-bit register from a shared memory of n-bit registers. Take $2^n + 2$ registers and set them all initially to zero. A $2n$-bit state l is stored by dividing l by $2^n - 1$, indexing into the primitive registers with the integer part of the result and storing the remainder plus one. For instance, 2^n divided by $2^n - 1$ gives 1, and a remainder of 1; thus, we would store 2 $(1 + 1)$ in the register at offset 1. All the other registers should be zero.

To read this $2n$-bit register, simply scan every primitive register until a non-zero value is found; if the register at offset i holds $j \neq 0$, the value of the $2n$-bit register is $(2^n - 1)i + j - 1$. Write and DWCAS are implemented by zeroing the sole non-zero register with a primitive CAS (effectively locking the object), then writing in the new value. (Figure 4.5.)

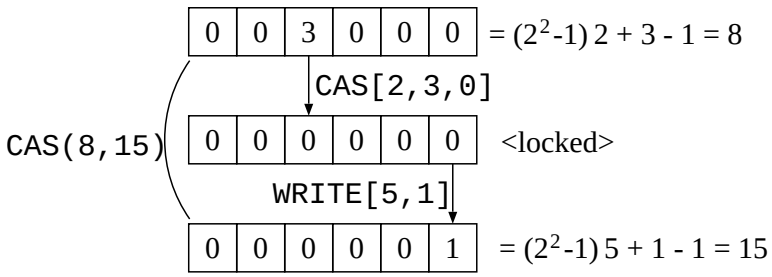


Figure 4.5: Implementing Compare-And-Swap from 8 to 15 in a simple, scalable, blocking implementation of a 4-bit register from a shared memory with only 2-bit registers. Offsets are counted from the left.

Aside from an exponential growth in execution times, which can be fixed, the main drawback of this algorithm is an exponential growth in storage costs as the number of bits grows. For instance, implementing a single 64-bit register on a 32-bit machine requires over **16 gigabytes** of space. Algorithms implementing stronger primitives from weaker ones with space costs growing exponentially in the number of bits is nothing new; for instance, Lamport presented a very similar algorithm to this one implementing multi-bit 'regular' registers from single-bit ones [51]. However, such space demands are unacceptable outside of pure theory. Nevertheless, I will now show that this algorithm is optimally space-efficient given the requirements.

Lemma 4.3.1 *Let $\mathbb{M}$ be a read parallel, population-oblivious, garbage-free implementation of a complete snapshot type $\mathbb{L}$ from an orthogonal, deterministic primitive $\mathbb{P}$. Then for any ordering $<$ on $\mathbb{S}_\mathbb{L}$, there exists a map $m : \mathbb{S}_\mathbb{L} \to \mathbb{S}_\mathbb{P}$, with $m(l)$ a sequentially-reachable state representing l $\forall l \in \mathbb{S}_\mathbb{L}$, such that the following holds:*

$$\forall l \in \mathbb{S}_\mathbb{L}, \exists r_l \in \mathbb{O}_\mathbb{P} \text{ s.t. } r_l(m(l)) \neq r_l(m(l')) \; \forall l' \in \mathbb{S}_\mathbb{L}, l' < l$$

Proof To prove the lemma, I choose a map m satisfying certain properties, and a sequence of history fragments connecting the sequentially-reachable states in the range of m, which can be run serially in any combination. By executing these fragments during a concurrent read operation, I establish that the map chosen indeed satisfies the requirements of the lemma.

Without loss of generality, I assume $\mathbb{S}_\mathbb{L} = \{0, 1, \ldots, s\}$, where $s + 1 = |\mathbb{S}_\mathbb{L}|$, such that the ordering $<$ on $\mathbb{S}_\mathbb{L}$ reduces to the standard ordering of the integers. In the following, I will refer to a sequential history fragment F that never passes through any states (either logical or sequentially-reachable) outside a set S as going *on* S; that passes through a (logical or sequentially-reachable) state s at least once as going *via* s; and that finishes in (logical or sequentially-reachable) state s as going *to* s.

I choose a map $m : \mathbb{S}_\mathbb{L} \to \mathbb{S}_\mathbb{P}$, sequential history H, and sequential history fragments $(F_l)_{l=1,\ldots,s}$ and $(F_l^{-1})_{l=1,\ldots,s}$ such that: H ends in logical state 0 and sequentially-reachable state $m(0)$; F_l extends H via $[0, l]$ to logical state l represented by sequentially-reachable state $m(l)$; and F_l^{-1} extends HF_l via $[0, l]$ to logical state 0 and sequentially-reachable state $m(0)$. (Figure 4.6)

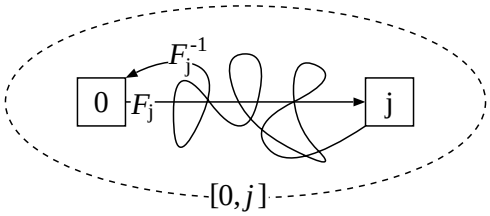


Figure 4.6: Each state j is connected to state 0 by fragments F_j and F_j^{-1}, following a path that can only go via states $[0, j]$, not $(j, s]$.

Such map, history and fragments must exist by garbage-freedom. This is most easily shown by induction. It is trivially true for $s = 0$, so suppose the theorem holds for some $s' >= 0$ and let $s = s' + 1$. By garbage-freedom, there exists some history H_s ending in logical state s and some sequentially-reachable state $m(s)$ such that for any history fragment F extending H_s, there exists another fragment F' extending H_sF to sequentially-reachable state $m(s)$. Let F_s^{-1} be a history fragment extending H_s to logical state 0. By restricting $\mathbb{M}$ to start from

the final sequentially-reachable state of HF, and disallowing any operations that reach state s, we obtain a new implementation, $\mathbb{M}'$, of a complete type with $s'+1$ states. By the inductive hypothesis, there is some map $m' : [0, s'] \to \mathbb{S}_{\mathbb{P}}$ satisfying the above requirements for $\mathbb{M}'$. The trivial extension of m' to the domain of $\mathbb{S}$, mapping s to $m(s)$, therefore satisfies the requirements for $\mathbb{M}$, as desired.

By determinism, I can now move between the given sequentially-reachable representations of each state without passing through a larger state by composing the various history fragments. I wish to use this to prove the lemma holds for this m.

The condition of the lemma is trivial for $l = 0$, so take any $l > 0$. By Lemma 4.2.1, since the implementation is population-oblivious and read parallel, it must have invisible reads. The history $HF_1F_1^{-1} \cdots F_{l-1}F_{l-1}^{-1}F_l$ can therefore be extended by a history fragment G consisting of a single logical event E s.t. $u_E =$ id, $\mathbb{L}$'s snapshot operation, and $\forall A \in HF_1F_1^{-1} \cdots F_{l-1}F_{l-1}^{-1}F_l,\ f^m(E) \cup f(A) = \emptyset$ and $E \overset{t}{\not\sim} A$. This event consists of the execution of a series of operations $r_1 \ldots r_k \in \mathbb{O}_{\mathbb{P}}$, some k, and must return l as its response. (Figure 4.7)

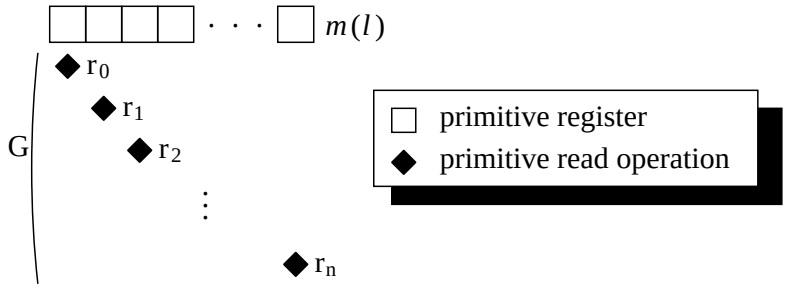


Figure 4.7: History fragment G executes id on logical state l, represented by sequentially-reachable state $m(l)$.

Suppose $\forall i \in [1, k], \exists l_i < l$ s.t. $r_i(m(l_i)) = r_i(m(l))$, and consider scheduling E during the history $HF_{l_1}F_{l_1}^{-1} \cdots F_{l_k}F_{l_k}^{-1}$ such that each step i is executed immediately after the corresponding fragment F_{l_i} (Figure 4.8). By determinism and by construction, in such a schedule, E would comprise the same primitives, (r_i), and would return the same value, l, as in the original history; yet, again by construction, $F_{l_1}F_{l_1}^{-1} \cdots F_{l_k}F_{l_k}^{-1}$ never passes through state l — a contradiction.

Hence the supposition must be invalid, and $\exists i \in [1, m]$ s.t. $r_i(m(l')) \neq r_i(m(l)) \forall l' < l$, as desired.

Theorem 4.3.2 *A scalable implementation of DWCAS from n-bit read, write and CAS operations requires more than 2^n registers.*

Proof By Lemma 4.3.1, each of the 2^{2n} states in a double-word must be associated with a unique word-sized register–state pair; specifically, the register read

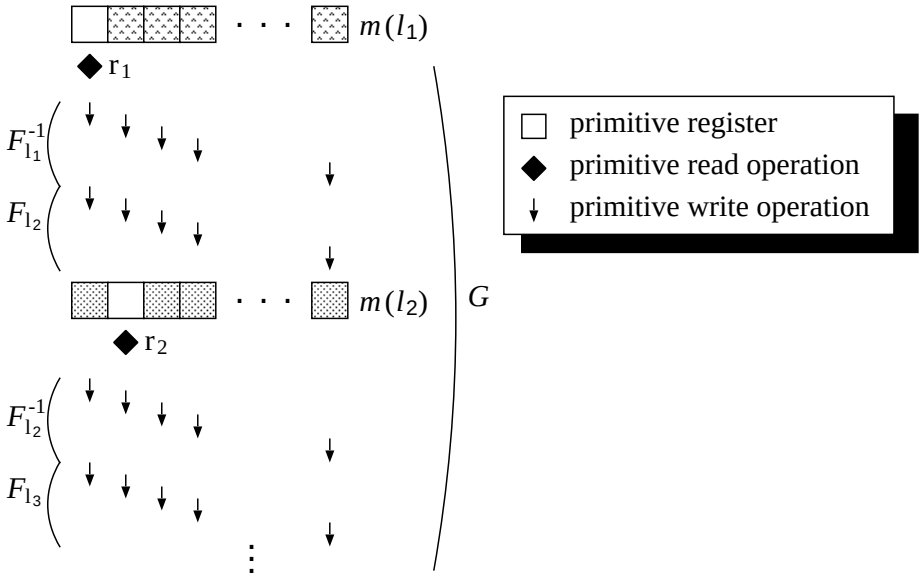


Figure 4.8: If no r_i returns a unique value, history fragment G can be scheduled during a history chosen such that each r_i returns the same value, yet the history is never in logical state l during id's execution.

by r_l and the state $r_l(m(l))$. This requires using a minimum of 2^n registers. However, each register must also reserve one state for representing all the double-word values that do not have a unique state paired with that register. The theorem follows.

4.4 Load-Linked/Store-Conditional

There is one primitive which is tricky to apply Lemmas 4.2.2 and 4.3.1 to: LL/SC, introduced in Section 3.1. This is because there are two ways of modelling the primitives, one orthogonal, one not. I will now briefly cover three variants of LL/SC: 'write-like', 'read-like' and 'weak read-like'. Surprisingly, the maximal disjointness of scalable operations built from LL/SC pairs depends on which variant is implemented, as only write-like LL/SC is orthogonal.

One way LL/SC can be modelled by attaching to each register an *ownership value*, storing the ID of the last thread to load-link the location. Writes reset the ownership value to empty. SC then succeeds only if the register is still owned by the current thread. This *write-like* LL is orthogonal, and the Theorem applies to it: write-like LL/SC cannot implement DCAS scalably.

The other way of modelling LL/SC is to attach to each register an infinite number of new "read-lock" synchronization points, one for each thread. LL flips

the read-lock for the thread, and SC succeeds only if it hasn't been cleared again. A write (and a successful, updating SC) will clear *all* the read-locks for that register. This *read-like* LL is not orthogonal, as write operations have an infinite footprint. The theorem thus does not apply to it.

In fact, it is possible to build a simple, scalable snapshot operation from read-like LL/SC by load-linking all locations in the snapshot, then using SC to verify each in turn without updating them. If all LL/SC pairs succeed, the snapshot linearizes to the last LL. Hence, the maximal disjointness of groups of read-like LL/SC pairs is in fact unbounded, even though LL/SC individually has a maximal disjointness of one. Further, the space requirements of such a snapshot grows linearly with the number of bits, not exponentially.

Both these characterizations of LL/SC are known as *strong*: LL/SC is guaranteed to succeed if and only if the linked location is not modified. Real implementations tend to weaken this to so-called *weak* LL/SC, where a pair is allowed to fail spuriously. In particular, weak LL/SC cannot be nested.

An analysis of Lemma 4.2.2 shows that the argument can be extended to cover this primitive: scalable implementations based on weak read-like LL/SC have a maximal disjointness of two. In particular, this means weak read-like LL/SC cannot scalably implement 3CAS, though it may be possible to implement DCAS.

I return to nestable read-like LL/SC operations in Section 7.4, where I discuss whether they can scalably implement transactional memory as well as atomic snapshots.

Chapter 5

Reasonable Scalability: Open-Addressed Hashtables

In this chapter, I present a novel non-blocking implementation of a partial function (also known as a map or dictionary), built from single-word read, write and CAS primitives, that provides good performance in a parallel benchmark by exhibiting *locality of reference* and *reasonable scalability*.

An algorithm exhibits *locality of reference* if the primitive operations implementing a logical operation tend to access adjacent words. Shared memory subsystems typically exploit locality of reference to improve performance by operating on *cache lines* rather than individual words: thus, reading a word will cause its entire line to be cached, speeding subsequent reads; while to update it, exclusive access must be negotiated for the whole line, speeding subsequent updates. Ensuring an algorithm exhibits locality of reference may therefore improve its straight-line performance.

I first introduce an open-addressed hashtable and briefly motivate why it is a good algorithm to adapt, then describe several problems to be overcome in parallelizing the basic single-threaded algorithm. I then dedicate a section to each of the solutions used: explicit bounds; whack-a-mole; version counters; compaction; and counters. In the process, I describe and motivate an informal property I call *reasonable scalability*.

5.1 Open-Addressing

In general, CAS-based, population-oblivious, disjoint-access and read parallel algorithms must, as proved in Chapter 4, produce garbage as they run. If the footprint of the data structure, and specifically the available read parallelism, grows and shrinks dynamically and continually, this demands a garbage collector to reclaim dynamically freed memory. Further, such algorithms typically cannot exhibit locality of reference: since locations cannot be reused until all readers

have been checked to ensure it is safe, which will typically be delayed to minimise communication costs, pointers will end up referencing memory far from them. This penalty is one reason to insist of strict garbage-freedom in a universal construct. However, locality of reference would then come at the cost of worse scaling in the number of threads.

If the footprint of an algorithm remains constant, however, the production of garbage can be restricted to specific disallowed values at specific locations, managed internally by the algorithm, avoiding the need for an out-of-line garbage collector. This potentially allows pertinent information to be kept in a single predetermined location, leading to locality of reference which can improve performance. Further, in real-world applications, it is a reasonable assumption that algorithms do not take an unbounded time to execute. Reuse of very old garbage values is thus safe. The simplest example of all of these properties is the use of *version counters* [50] to allow readers concurrent access to a shared variable; it is implausible for a 64-bit counter to overflow during a single operation, and hence such a counter can reasonably be treated as infinite, or alternatively as garbage-free. I refer to this informal property as *reasonable garbage-freedom*, and when combined with the remaining scalability properties under other reasonable assumptions, *reasonable scalability*.

A *hashtable* is an array of buckets for storing keys in. Each potential key that could be stored in the hashtable is assigned to a bucket using a static hash function known to all threads. An *open-addressed* hashtable stores its *collisions* (keys that cannot be stored in their preassigned bucket because it is already full) in other buckets, following a static *probe sequence*. This allows the memory footprint to remain constant, provided the hashtable does not fill up, and also exhibits locality of reference, as in the average case where a lookup hits or misses in the first bucket, only a single cacheline is involved.

An open-addressed hashtable is thus an ideal algorithm to parallelize, if it can retain its functional properties. However, there are a number of obstacles to overcome: first, terminating searches along the probe sequence as soon as possible; second, ensuring parallel insertions do not create duplicate keys; third, storing large keys and guaranteeing lock-free progress; fourth, allowing values to be stored alongside the keys and replaced atomically; fifth, allowing the hashtable to grow dynamically.

Subsequent sections address each of these points in turn. Terminating searches is done by allocating a *bound* to each probe sequence, beyond which it is guaranteed that no buckets contain any key in the sequence. Managing parallel key insertion is done with a consensus algorithm I call *whack-a-mole*. Adding *version counters* to each bucket allows large keys and a lock-free progress guarantee. A *compaction* algorithm allows both atomic value replacement and dynamic growth. Finally, I consider ways of implementing *concurrent counters* to determine when to grow the hashtable.

By the end of this chapter, I will have presented a reasonably scalable, lock-

free implementation of a partial function, exhibiting exploitable locality of reference, and evaluated its performance.

5.2 Bounding Searches

The first problem I address is that of bounding searches. Since a collision may be stored in any bucket on the probe sequence, assuming all previous ones were at some point full, some mechanism is needed to prevent lookups from having to search every bucket. The standard approach is to treat empty buckets as a kind of 'stop sign' for searches, but this complicates deletion, as buckets can no longer be marked empty lest subsequent searches miss collisions further down the probe sequence.

The canonical solution to this is to leave 'tombstones' when emptying a bucket, which can be reused for subsequent inserts but which do not act as a stop sign for a search. However, unless these tombstones are periodically removed by duplicating the hashtable, they will continue to multiply, resulting in degenerate search times.

Instead, I provide a 'stop sign' for each probe sequence, storing how far down the sequence the stop sign is currently located — a *bound* on how far searches need probe. By using quadratic probing, where each probe sequence depends only on the starting bucket, only a single bound is needed per bucket (Figure 5.1). In contrast, double hashing, where a key is hashed once to determine a starting bucket and again to choose a probe sequence stride, would have quadratic space growth for this scheme — one bound per sequence — an unacceptable overhead.

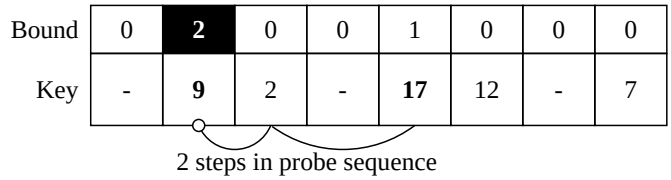
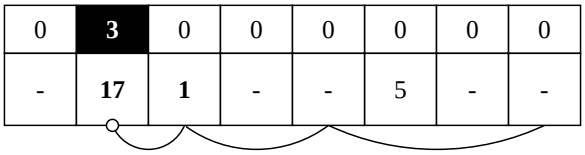
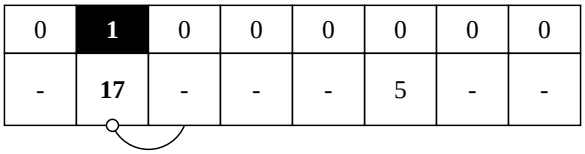


Figure 5.1: Bounds on collision indices for a hashtable holding keys 2, 7, 9, 12, 17. Hash function is $h(k) = k \bmod 8$, probe sequence is quadratic, p(k,i) = $(k + \frac{1}{2}(i^2 + i)) \bmod 8$. Key 17 is stored two steps along the probe sequence for bucket 1, so the probe bound is 2.

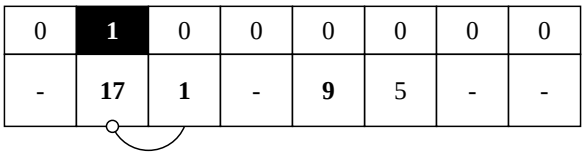
Maintaining these probe bounds concurrently is complicated by the need to lower them: simply scanning the probe sequence for the previous collision and swapping it into the bound field may result in the bound being too large if the collision is removed, slowing searches, or too small if another collision is inserted, violating correctness (Figure 5.2). Pseudocode for a correct algorithm can be found in Figure 5.3. I represent the packing of an int and a bit into a machine word with the $\langle ., . \rangle$ operator.



After a collision is removed, a thread scans for the previous collision.



If a concurrent erasure is missed, the bound may be left too large.



Worse, if a concurrent insertion is missed, the bound may be made too small.

Figure 5.2: Problems maintaining a shared bound after a collision is removed from the end of the probe sequence.

In order to keep the bounds correct during erasures, I use a *scanning phase* during which the thread erasing the last collision in the probe sequence searches through the previous buckets to compute the new bound (lines 18–22). A thread announces that it is in this phase by setting a *scanning bit* to true (line 18); this bit is held in the same word as the bound itself, so both fields are updated atomically.

Dealing with insertions is now easy: they atomically clear the scanning bit and raise the bound if necessary (lines 9–12). Deletions also clear the scanning bit (line 16), but are complicated by the scanning phase. I rely on the fact that at most one thread can be in the process of erasing a given collision, and that threads only start scanning when erasing the last collision in the probe sequence. The collision's index value thus identifies the scanning thread and, if it is still present as the bound when scanning completes, and if the scanning bit is still set, there cannot have been any concurrent updates (line 22). Otherwise, the scanning phase is repeated.

Given a lock-free atomic compare-and-swap (CAS) function, the pseudocode in Figure 5.3 is lock-free and parallelism preserving.

Next, I address the problem of implementing concurrent insertions and deletions, ensuring duplicate keys are never allowed.

```
1   class Set {
        word bounds[size] // ⟨bound,scanning⟩
3       void InitProbeBound(int h):
            bounds[h] := ⟨0,false⟩
5       int GetProbeBound(int h): // Maximum offset of any collision in probe seq.
            ⟨bound,scanning⟩ := bounds[h]
7           return bound
        void ConditionallyRaiseBound(int h, int index): // Ensure maximum ≥ index
9           do
                ⟨old_bound,scanning⟩ := bounds[h]
11              new_bound := max(old_bound,index)
            while ¬CAS(&bounds[h],⟨old_bound,scanning⟩,⟨new_bound,false⟩)
13      void ConditionallyLowerBound(int h, int index): // Allow maximum < index
            ⟨bound,scanning⟩ := bounds[h]
15          if scanning = true
                CAS(&bounds[h],⟨bound,true⟩,⟨bound,false⟩)
17          if index > 0 // If maximum = index > 0, set maximum < index
                while CAS(&bounds[h],⟨index,false⟩,⟨index,true⟩)
19                  i := index-1 // Scanning phase: scan cells for new maximum
                    while i > 0 ∧ ¬DoesBucketContainCollision(h, i)
21                      i--
                    CAS(&bounds[h],⟨index,true⟩,⟨i,false⟩)
```

Figure 5.3: Per-bucket probe bounds (code continued in Figure 5.8)

5.3 Whack-a-Mole

Before continuing, I must first introduce a consensus algorithm that will be used throughout the remaining chapter: the *whack-a-mole* algorithm.

The primitive type is $\mathbb{F}[\mathbb{A}, \mathbb{X}]$, a set of 'faulty' registers mapping an infinite set of locations, $\mathbb{A}$, to a set of values, $\mathbb{X} \cup \{\bot\}$. Using the nomenclature of Chapter 2:

$$\begin{aligned}
\mathbb{S}_\mathbb{F} &= \{f : \mathbb{A} \to \mathbb{X} \cup \{\bot\}\} \\
\mathbb{O}_\mathbb{F} &= \left\{ \begin{array}{c} \texttt{Insert[}x\texttt{]}, \texttt{Read[}a\texttt{]}, \texttt{CAS[}a\texttt{,}x\texttt{,}x'\texttt{]}, \\ \texttt{Erase[}a\texttt{]}, \texttt{Iterate} \end{array} : x, x' \in \mathbb{X}, a \in \mathbb{A} \right\}
\end{aligned}$$

$$\begin{aligned}
\texttt{Insert[}x\texttt{]} \circ f &= g \in \mathbb{S}_\mathbb{F} \\
\texttt{Insert[}x\texttt{]}(f) &= a \in \mathbb{A} \\
&\quad \text{where } f(a) = \bot, g(a) = x \\
&\quad \text{and } \forall a' \in \mathbb{A} \backslash \{a\}\ (f(a') = g(a')) \\
\\
\texttt{Read[}a\texttt{]} \circ f &= f \\
\texttt{Read[}a\texttt{]}(f) &= f(a) \\
\texttt{CAS[}a\texttt{,}x\texttt{,}x'\texttt{]} \circ f &= g \in \mathbb{S}_\mathbb{F} \\
&\quad \text{where } g(a) = \left\{ \begin{array}{ll} x' & f(a) = x \\ f(a) & \text{otherwise} \end{array} \right. \\
&\quad \text{and } \forall a' \in \mathbb{A} \backslash \{a\}\ (f(a') = g(a')) \\
\texttt{CAS[}a\texttt{,}x\texttt{,}x'\texttt{]}(f) &= \left\{ \begin{array}{ll} \text{true} & f(a) = x \\ \text{false} & \text{otherwise} \end{array} \right. \\
\\
\texttt{Erase[}a\texttt{]} \circ f &= g \in \mathbb{S}_\mathbb{F} \\
&\quad \text{where } g(a) = \bot \\
&\quad \text{and } \forall a' \in \mathbb{A} \backslash \{a\}\ (f(a') = g(a')) \\
\texttt{Erase[}a\texttt{]}(f) &= \bot
\end{aligned}$$

The insert function places a given value into a location that previously held $\bot$. The important point to note is that registers in $\bot$ state may become 'faulty', i.e. cannot have a value placed into them, for arbitrary lengths of time; thus, the best the insert function can guarantee is that *some* location will end up holding the value. Once a location starts holding a non-$\bot$ value, it cannot become faulty again until it is erased.

The last function, `Iterate`, returns an *iterator*, i, with a single operation, `Deref`, returning values in $(\mathbb{A} \times \mathbb{X}) \cup \{\bot\}$. This allows an algorithm to iterate over all the non-$\bot$ values in the registers; the iterator will return $\bot$ once all locations have been traversed. However, this iterator is not atomic. More specifically, if the iterating algorithm begins at time t_0, and ends at time t_1; the state of the shared object of type $\mathbb{F}$ at time t is f_t; and $\forall a \in \mathbb{A}$, $f(a) = x$ if the iterator returns (a, x) for some $x \in \mathbb{X}$, $\bot$ otherwise; then

$$\forall a \in \mathbb{A}\ (\exists t \in [t_0, t_1]\ (f_t(a) = f(a)))$$

The whack-a-mole algorithm implements a single register from such an infinite set of 'faulty' registers. The logical type of this register is $\mathbb{L}$:

$$\begin{array}{rcl} \mathbb{S}_{\mathbb{L}} & = & \mathbb{V} \cup \{\bot\} \\ \mathbb{O}_{\mathbb{L}} & = & \{\texttt{Read}, \texttt{Insert[}v\texttt{]}, \texttt{Erase} : v \in \mathbb{V}\} \end{array}$$

$$\begin{array}{rcl} \texttt{Read} \circ v' & = & v' \\ \texttt{Read}(v') & = & v' \\ \texttt{Insert[}v\texttt{]} \circ v' & = & \begin{cases} v & v' = \bot \\ v' & \text{otherwise} \end{cases} \\ \texttt{Insert[}v\texttt{]}(v') & = & \begin{cases} true & v' = \bot \\ false & \text{otherwise} \end{cases} \\ \texttt{Erase} \circ v' & = & \bot \\ \texttt{Erase}(v') & = & \begin{cases} true & v' \neq \bot \\ false & \text{otherwise} \end{cases} \end{array}$$

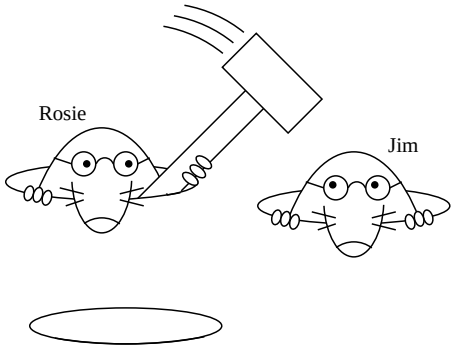


Figure 5.4: Moles and hammers: a uniqueness algorithm. Rosie reaches into Hammerspace and whacks Jim, preventing him from emerging simultaneously.

As a visual aid, I introduce an analogy: a group of moles are wanting to leave their holes and come into the air, but with two constraints: two moles cannot be out at the same time; and they can only communicate pair-wise. To achieve this, each mole first pushes its nose into the air. It then whacks any other moles back into their holes. If, after this, the mole has neither found any other moles fully emerged, nor been whacked itself, it emerges from the hole. Whacked moles continue to retry until one fully emerges.

To see that this indeed ensures a unique consensus winner, suppose that two moles, Rosie and Jim, are both fully in the air at once. Consider the last time each poked their nose out: for the sake of argument, say Rosie did this no earlier than Jim. Now, for Rosie to subsequently emerge, she *must* first try to whack Jim on the head, either preventing him from emerging or letting Rosie know he

has emerged, and hence preventing *her* from emerging. This is a contradiction; hence uniqueness holds.

To implement the whack-a-mole algorithm, the state space $\mathbb{V}$ is extended with a state machine:

$$\mathbb{X} = \{\text{'whacked'}, (\text{'nose in the air'}, v), (\text{'fully emerged'}, v) : v \in \mathbb{V}\}$$

The `Insert[`v`]` algorithm is then as in Figure 5.5.

```
bool Insert[v]:
  // Push nose into the air
  a := m.Insert[('nose in the air', v)]
  while (true)
    // Whack other moles back into their holes
    i := m.Iterate
    next := i.Deref
    while (next ≠ ⊥)
      (a', x) := next
      if (a' ≠ a ∧ x ≠ 'whacked')
        (s, v) := x
        if (s = 'nose in the air')
          m.CAS[a', (s, v), 'whacked']
          x := m.Read[a']
          if (x ≠ ⊥ ∧ x ≠ 'whacked')
            (s, v) := x
        if (s = 'fully emerged')
          Erase[a]
          return false
    // Emerge from the hole
    if (m.CAS[a, ('nose in the air', v), ('fully emerged', v)])
      return true
    // Retry
    m.CAS[a, 'whacked', ('nose in the air', v)]
```

Figure 5.5: The whack-a-mole algorithm. Inserting value $v \in \mathbb{V}$, given primitive object m of type $\mathbb{F}$.

To read the value of the logical register, iterate over the base type looking for a 'fully emerged' entry. If one is found, the operation can linearize to the moment it reads it. If an operation takes place in an interval of time in which the value of the register does not change (and is not $\perp$), then the properties of `Iterate` guarantee this entry will be found. Hence, if none is found, there must be some point during the operation in which the value of the set is $\perp$, and the operation can linearize at this point.

Erasing the register couples a read scan with a CAS of any fully emerged entry to 'whacked' state; proof of correctness follows the same pattern as for reads.

(A formal statement of these proofs can be found in the appendix of the extended version of "Non-blocking Hashtables with Open Addressing." [70])

This algorithm is obstruction-free, and does not fully implement a register as atomically replacing a non-$\perp$ value with another non-$\perp$ value is not possible. However, this is sufficient to implement an obstruction-free set in a hashtable. Subsequent sections will address these deficiencies, but for the sake of expediency only the resulting hashtable algorithms will be presented, not the underlying improvements to whack-a-mole consensus.

5.4 Inserting and Removing Keys

$$
\begin{array}{rcll}
\mathbb{S}_{\texttt{SET}} & = & \{\bot, \top\}^{\mathbb{K}} & \text{some keyspace } \mathbb{K} \\
\mathbb{O}_{\texttt{SET}} & = & \{\texttt{Lookup[k]}, \texttt{Insert[k]}, \texttt{Erase[k]} : k \in \mathbb{K}\} & \\
\mathbb{R}_{\texttt{SET}} & = & \{\texttt{Lookup[k]} : k \in \mathbb{K}\} & \\
R_{\texttt{SET}} & = & \{\bot, \top\} & \\
\mathbb{Y}_{\texttt{SET}} & = & \mathbb{K} &
\end{array}
$$

$$
\left.
\begin{array}{rcl}
\texttt{Lookup[k]} \circ \mathbf{s} & = & \mathbf{s} \\
\texttt{Lookup[k]}(\mathbf{s}) & = & s_k \\
f(\texttt{Lookup[k]}) & = & \{k\} \\
f^m(\texttt{Lookup[k]}) & = & \emptyset \\
\texttt{Insert[k]} \circ \mathbf{s} & = & (s_0, \ldots, s_{k-1}, \top, s_{k+1}, \ldots) \\
\texttt{Insert[k]}(\mathbf{s}) & = & \neg s_k \\
f(\texttt{Insert[k]}) & = & \{k\} \\
f^m(\texttt{Insert[k]}) & = & \{k\} \\
\texttt{Erase[k]} \circ \mathbf{s} & = & (s_0, \ldots, s_{k-1}, \bot, s_{k+1}, \ldots) \\
\texttt{Erase[k]}(\mathbf{s}) & = & s_k \\
f(\texttt{Erase[k]}) & = & \{k\} \\
f^m(\texttt{Erase[k]}) & = & \{k\}
\end{array}
\right\} \forall k \in \mathbb{K}, \mathbf{s} \in \mathbb{S}_{\texttt{SET}}
$$

Inserting keys when concurrent deletions are possible is complicated by the lack of a pre-determined bucket for any given key: once the bucket that once held a key is empty, it may be reused for other keys, forcing subsequent writers to come to consensus on a new bucket.

Fortunately, this is exactly the set of circumstances that the whack-a-mole algorithm addresses: building a single register (the value associated with a given key, in the case of a set either 'present' or 'absent') from a set of 'faulty' registers (buckets that may be storing other keys).

I employ a state machine (Figure 5.6) in each bucket. 'Nose in the air' moles are represented by the `inserting` state, and 'fully emerged' moles by the `member` state. Insertions are split into the three whack-a-mole stages (Figure 5.7). First, a thread *pushes its nose into the air* by reserving an empty bucket and storing the key it is inserting, putting the bucket into `inserting` state.

Next, the thread checks the other positions in the probe sequence for that key, looking for other threads with `inserting` entries, or for a completed insertion of the same key. If it finds another insertion in progress in a bucket then it *whacks it back into its hole* by changing that bucket's state to `busy`, stalling the other insertion at that point in time. If it finds another completed insertion of the same key, then its own insertion has failed: it climbs back into its hole, empties its bucket and returns `false`.

In the final stage, it attempts to *emerge from the hole*: to finish its own insert by changing its bucket from `inserting` to `member` state. It must do this

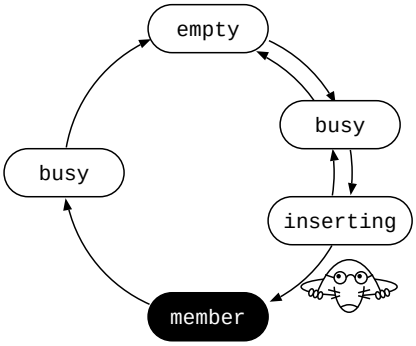


Figure 5.6: State machine used in hashtable. The mole represents a state transition which can only be taken after using the whack-a-mole algorithm to ensure uniqueness; only one bucket can be in the white-on-black `member` state at any one time for a given key. Note that the `busy` state intentionally appears twice.

atomically with a CAS instruction so that it fails if whacked by another thread; if stalled, the thread republishes its attempt and restarts the second stage.

(Note that the mapping from this state machine system to the whack-a-mole system is done on a *per-key basis*. If we are considering key k, for instance, then any bucket holding any key $k' \neq k$ maps to the $\perp$ state in the whack-a-mole system.)

Obstruction-free pseudocode implementing this algorithm can be found in Figure 5.8. Each bucket contains a four-valued state, one of *empty*, *busy*, *inserting* or *member*, and, for the latter two states, a key. The key and state must be modified atomically; I use the $\langle ., . \rangle$ operator to represent packing them into a single word. A key k is considered inserted if some bucket in the table contains $\langle k, member \rangle$. The `Hash` function selects a bucket for a given key. The three insertion stages can be found in lines 42–50, 51–60 and 61, respectively.

Unlike Martin and Davis' approach [56], empty buckets are immediately free for arbitrary reuse, so table replication is not needed to clear out tombstones. The algorithm preserves read parallelism and, assuming disjoint keys hash to separate memory locations, disjoint access parallelism. In the expected case where the bucket contains no collisions, the operation footprint is two words — a single cache line if buckets and bounds are interleaved.

Probe bound	0	2	0	0	1	0	0	0
State	empty	member	member	empty	member	inserting	empty	empty
Key	-	9	1	-	17	12	-	-

Initial state.

Probe bound	0	2	0	0	**2**	0	0	0
State	empty	member	member	empty	member	inserting	empty	**inserting**
Key	-	9	1	-	17	12	-	**12**

Push nose in the air in the third cell in the probe sequence,
raising the probe bound appropriately.

Probe bound	0	2	0	0	2	0	0	0
State	empty	member	member	empty	member	**empty**	empty	inserting
Key	-	9	1	-	17	**-**	-	12

Whack concurrent insertion attempt in the second cell in the sequence.

Probe bound	0	2	0	0	2	0	0	0
State	empty	member	member	empty	member	empty	empty	**member**
Key	-	9	1	-	17	-	-	12

Emerge fully into member state, linearizing insertion of key 12.

Figure 5.7: Inserting key 12 with the whack-a-mole approach.

```
23      word buckets[size] // ⟨key,state⟩

        word* Bucket(int h, int index): // Size must be a power of 2
25          return &buckets[(h + index*(index+1)/2) % size] // Quadratic probing

        bool DoesBucketContainCollision(int h, int index):
27          ⟨k,state⟩ := *Bucket(h,index)
            return (k ≠ - ∧ Hash(k) = h)

29  public:
        void Init():
31          for i := 0 .. size-1
                InitProbeBound(i)
33              buckets[i] := empty

        bool Lookup(Key k): // Determine whether k is a member of the set
35          h := Hash(k)
            max := GetProbeBound(h)
37          for i := 0 .. max
                if *Bucket(h,i) = ⟨k,member⟩
39                  return true
            return false

41      bool Insert(Key k): // Insert k into the set if it is not a member
            h := Hash(k)
43          i := 0 // Reserve a cell
            while ¬CAS(Bucket(h,i), empty, busy)
45              i++
                if i ≥ size
47                  throw "Table full"
            do // Attempt to insert a unique copy of k
49              *Bucket(h,i) := ⟨k,inserting⟩
                ConditionallyRaiseBound(h,i)
51              max := GetProbeBound(h) // Scan through the probe sequence
                for j := 0 .. max
53                  if j ≠ i
                        if *Bucket(h,j) = ⟨k, inserting⟩ // Stall concurrent inserts
55                          CAS(Bucket(h,j), ⟨k,inserting⟩, busy)
                        if *Bucket(h,j) = ⟨k,member⟩ // Abort if k already a member
57                          *Bucket(h,i) := busy
                            ConditionallyLowerBound(h,i)
59                          *Bucket(h,i) := empty
                            return false
61          while ¬CAS(Bucket(h,i), ⟨k,inserting⟩, ⟨k,member⟩)
            return true

63      bool Erase(Key k): // Remove k from the set if it is a member
            h := Hash(k)
65          max := GetProbeBound(h) // Scan through the probe sequence
            for i := 0 .. max
67              if *Bucket(h,i) = ⟨k,member⟩ // Remove a copy of ⟨k, member⟩
                    if CAS(Bucket(h,i), ⟨k,member⟩, busy)
69                      ConditionallyLowerBound(h,i)
                        *Bucket(h,i) := empty
71                      return true
            return false
73  }
```

Figure 5.8: An obstruction-free set (continued from Figure 5.3)

5.5 Lock-Freedom and Multi-word Keys

I now turn to two shortcomings in the above algorithm. The first is that concurrent insertions may live-lock, each repeatedly stalling the other: the algorithm is therefore only obstruction-free, not lock-free. As given, the hashtable cannot support concurrent assistance, as Figure 5.10 demonstrates: a bucket's contents can change arbitrarily before returning to a previous state, allowing a CAS to succeed incorrectly. This is known as the ABA problem [1], and I return to it in a moment.

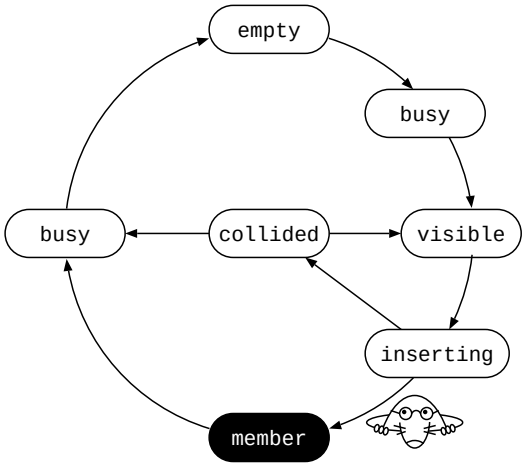


Figure 5.9: State machine of a single bucket in the lock-free hashtable. Only one bucket may be in the white-on-black `member` state at any one time for a given key; the mole represents a state transition that can only be taken after ensuring this uniqueness with the whack-a-mole algorithm. Note that the `busy` state intentionally appears twice.

The second problem is storing keys larger than a machine word: in the algorithm as given, this requires a multi-word CAS, which is not generally available. However, note that a bucket's key is only ever modified by a single writer, when the bucket is in busy state. This means we only need to deal with concurrent single-writer multiple-reader access to the bucket, rather than provide a general multi-word atomic update. Lamport's version counters [50] are therefore applicable. Pseudocode for performing lookups and erases with version counters, using the state machine shown in Figure 5.9, can be found in Figure 5.11.

If a bucket's state is stored in the same word as its version count, the ABA problem is circumvented, allowing threads to assist concurrent operations. This lets us create a lock-free insertion algorithm (diagram in Figure 5.12, pseudo-code in Figure 5.13).

Each bucket contains: a version count; a state field, one of *empty*, *busy*, *collided*, *visible*, *inserting* or *member*; and a key field, publically readable during

State	empty	inserting
Key	-	12

A single thread is about to complete its insertion of key 12. The next step is to atomically move the bucket from inserting to member state.

State	empty	member
Key	-	12

The thread is suspended, and its insertion assisted to completion by another thread.

State	member	inserting
Key	12	12

The key is now removed, and two other threads are concurrently attempting to reinsert key 12. One has just succeeded, and the other is about to remove itself. If the first thread wakes up at this point, it will still atomically move the bucket from inserting to member state, duplicating key 12.

Figure 5.10: Problems assisting concurrent operations

the latter three stages. The version count and state are maintained so that no state (except busy) will recur with the same version, assuming no wrapping.

As before, a thread finds an empty bucket and moves it into 'inserting' state (lines 64–75), and checks the probe sequence for other threads with 'inserting' entries, or a completed insertion of the same key (lines 85–105). However, if multiple 'inserting' entries are found, the earliest in the probe sequence is left unaltered, and the others moved into 'collided' state. When the whole probe sequence has been scanned and all contenders removed, the earliest entry is moved into 'member' state (line 104) and the insertion concludes (lines 77–82).

This version of the hashtable is lock-free. Further, given the reasonable assumption that the time taken for a version counter to repeat is longer than any operation will ever take to execute, and assuming a sufficiently large number of buckets, the algorithm is reasonably scalable.

```
23      struct BucketT {
            word vs // ⟨version,state⟩
25          Key key
        } buckets[size]

27      BucketT* Bucket(int h, int index): // Size must be a power of 2
            return &buckets[(h + index*(index+1)/2) % size] // Quadratic probing

29      bool DoesBucketContainCollision(int h, int index):
            ⟨version1,state1⟩ := Bucket(h,index)→vs
31          if state1 = visible ∨ state1 = inserting ∨ state1 = member
                if Hash(Bucket(h,index)→key) = h
33                  ⟨version2,state2⟩ := Bucket(h,index)→vs
                    if state2 = visible ∨ state2 = inserting ∨ state2 = member
35                      if version1 = version2
                            return true
37          return false

    public:
39      void Init():
            for i := 0 .. size-1
41              InitProbeBound(i)
                buckets[i].vs := ⟨0,empty⟩

43      bool Lookup(Key k): // Determine whether k is a member of the set
            h := Hash(k)
45          max := GetProbeBound(h)
            for i := 0 .. max
47              ⟨version,state⟩ := Bucket(h,i)→vs // Read cell atomically
                if state = member ∧ Bucket(h,i)→key = k
49                  if Bucket(h,i)→vs = ⟨version,member⟩
                        return true
51          return false

        bool Erase(Key k): // Remove k from the set if it is a member
53          h := Hash(k)
            max := GetProbeBound(h)
55          for i := 0 .. max
                ⟨version,state⟩ := Bucket(h,i)→vs // Atomically read/update cell
57              if state = member ∧ Bucket(h,i)→key = k
                    if CAS(Bucket(h,i)→vs, ⟨version,member⟩, ⟨version,busy⟩)
59                      ConditionallyLowerBound(h,i)
                        Bucket(h,i)→vs := ⟨version+1,empty⟩
61                      return true
            return false
```

Figure 5.11: Version-counted derivative of Figure 5.8 (continued in Figure 5.13)

Probe bound	0	2	0	0	1	0	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	empty	empty
Key	-	9	1	-	17	12	-	-

Initial state.

Probe bound	0	2	0	0	**2**	0	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	empty	**inserting**
Key	-	9	1	-	17	12	-	**12**

Push nose in the air in the third cell in the probe sequence,
raising the probe bound accordingly.

Probe bound	0	2	0	0	2	0	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	inserting	empty	**collided**
Key	-	9	1	-	17	12	-	12

Earlier `inserting` entry found;
whack *own bucket* into collided mode.

Probe bound	0	2	0	0	2	0	0	0
Version	18	2	3	6	4	3	24	7
State	empty	member	member	empty	member	**member**	empty	collided
Key	-	9	1	-	17	12	-	12

Assist *earlier entry* to emerge fully into `member` state.

Probe bound	0	2	0	0	**1**	0	0	0
Version	18	2	3	6	4	3	24	**8**
State	empty	member	member	empty	member	member	empty	**empty**
Key	-	9	1	-	17	12	-	-

Empty own bucket, lower probe sequence bound, and return `false`.

Figure 5.12: Inserting key 12 (lock-free algorithm). As in the obstruction-free algorithm, duplicated attempts to insert the key are moved to `collided` state; however, the presence of version counters now allows the collided thread to assist the conflicting insertion to completion. The version count is incremented every time a bucket passes through `empty` state.

```
63      bool Insert(Key k): // Insert k into the set if it is not a member
            h := Hash(k)
65          i := -1 // Reserve a cell
            do
67              if ++i ≥ size
                    throw "Table full"
69              ⟨version,state⟩ := Bucket(h,i)→vs
            while ¬CAS(&Bucket(h,i)→vs, ⟨version,empty⟩, ⟨version,busy⟩)
71          Bucket(h,i)→key := k
            while true // Attempt to insert a unique copy of k
73              *Bucket(h,i)→vs := ⟨version,visible⟩
                ConditionallyRaiseBound(h,i)
75              *Bucket(h,i)→vs := ⟨version,inserting⟩
                r := Assist(k,h,i,version)
77              if Bucket(h,i)→vs ≠ ⟨version,collided⟩
                    return true
79              if ¬r
                    ConditionallyLowerBound(h,i)
81                  Bucket(h,i)→vs := ⟨version+1,empty⟩
                    return false
83              version++

    private:
85      bool Assist(Key k,int h,int i,int ver_i): // Attempt to insert k at i
            // Return true if no other cell seen in member state
87          max := GetProbeBound(h) // Scan through probe sequence
            for j := 0 .. max
89              if i ≠ j
                    ⟨ver_j,state_j⟩ := Bucket(h,j)→vs
91                  if state_j = inserting ∧ Bucket(h,j)→key = k
                        if j < i // Assist any insert found earlier in the probe sequence
93                          if Bucket(h,j)→vs = ⟨ver_j,inserting⟩
                                CAS(&Bucket(h,i)→vs, ⟨ver_i,inserting⟩, ⟨ver_i,collided⟩)
95                          return Assist(k,h,j,ver_j)
                        else // Fail any insert found later in the probe sequence
97                          if Bucket(h,i)→vs = ⟨ver_i,inserting⟩
                                CAS(&Bucket(h,j)→vs, ⟨ver_j,inserting⟩, ⟨ver_j,collided⟩)
99                  ⟨ver_j,state_j⟩ := Bucket(h,j)→vs // Abort if k already a member
                    if state_j = member ∧ Bucket(h,j)→key = k
101                     if Bucket(h,j)→vs = ⟨ver_j,member⟩
                            CAS(&Bucket(h,i)→vs,⟨ver_i,inserting⟩,⟨ver_i,collided⟩)
103                     return false
            CAS(&Bucket(h,i), ⟨ver_i,inserting⟩, ⟨ver_i,member⟩)
105         return true
    }
```

Figure 5.13: Lock-free insertion algorithm (continued from Figure 5.11)

5.6 Value Replacement

$$
\begin{aligned}
\mathbb{S}_{\texttt{MAP}} &= (\{\bot\} \cup \mathbb{V})^{\mathbb{K}} \qquad \text{some keyspace } \mathbb{K}\text{, value space } \mathbb{V} \\
\mathbb{O}_{\texttt{MAP}} &= \{\texttt{Lookup[k]}, \texttt{Erase[k]} : k \in \mathbb{K}\} \cup \\
&\quad \{\texttt{Insert[k,v]}, \texttt{Replace[k,v]} : k \in \mathbb{K}, v \in \mathbb{V}\} \\
\mathbb{R}_{\texttt{MAP}} &= \{\texttt{Lookup[k]} : k \in \mathbb{K}\} \\
R_{\texttt{MAP}} &= \{\bot\} \cup \mathbb{V} \\
\mathbb{Y}_{\texttt{MAP}} &= \mathbb{K}
\end{aligned}
$$

$$
\left.
\begin{aligned}
\texttt{Lookup[k]} \circ \mathbf{s} &= \mathbf{s} \\
\texttt{Lookup[k]}(\mathbf{s}) &= s_k \\
f(\texttt{Lookup[k]}) &= \{k\} \\
f^m(\texttt{Lookup[k]}) &= \emptyset \\
\texttt{Insert[k,v]} \circ \mathbf{s} &= \begin{cases} (\ldots, s_{k-1}, v, s_{k+1}, \ldots) & s_k = \bot \\ \mathbf{s} & \text{otherwise} \end{cases} \\
\texttt{Insert[k,v]}(\mathbf{s}) &= s_k \\
f(\texttt{Insert[k,v]}) &= \{k\} \\
f^m(\texttt{Insert[k,v]}) &= \{k\} \\
\texttt{Replace[k,v]} \circ \mathbf{s} &= \begin{cases} (\ldots, s_{k-1}, v, s_{k+1}, \ldots) & s_k \neq \bot \\ \mathbf{s} & \text{otherwise} \end{cases} \\
\texttt{Replace[k,v]}(\mathbf{s}) &= s_k \\
f(\texttt{Replace[k,v]}) &= \{k\} \\
f^m(\texttt{Replace[k,v]}) &= \{k\} \\
\texttt{Erase[k]} \circ \mathbf{s} &= (\ldots, s_{k-1}, \bot, s_{k+1}, \ldots) \\
\texttt{Erase[k]}(\mathbf{s}) &= s_k \\
f(\texttt{Erase[k]}) &= \{k\} \\
f^m(\texttt{Erase[k]}) &= \{k\}
\end{aligned}
\right\} \quad \begin{array}{l} k \in \mathbb{K} \\ \forall\ v \in \mathbb{V} \\ \mathbf{s} \in \mathbb{S}_{\texttt{MAP}} \end{array}
$$

Until now, I have been implementing a set. I now consider implementing a *partial function*, also known as a *map* or *dictionary*, where each key has an associated value. Using the algorithm presented above, one cannot atomically replace a value associated with a key; removing the original value then inserting the new one is not an adequate substitute as it is not atomic. I will now show how to extend the state machine to allow replacement as well as insertion and deletion.

I present three algorithms for this purpose: one where keys *migrate* from bucket to bucket; one where values are updated *in-place*; and a final, hybrid scheme where the act of replacement also *compacts* the probe sequence, migrating keys to earlier buckets where possible.

5.6.1 Migration

My first approach appears to migrate a key around the hashtable each time its value is replaced, using the state machine in Figure 5.14.

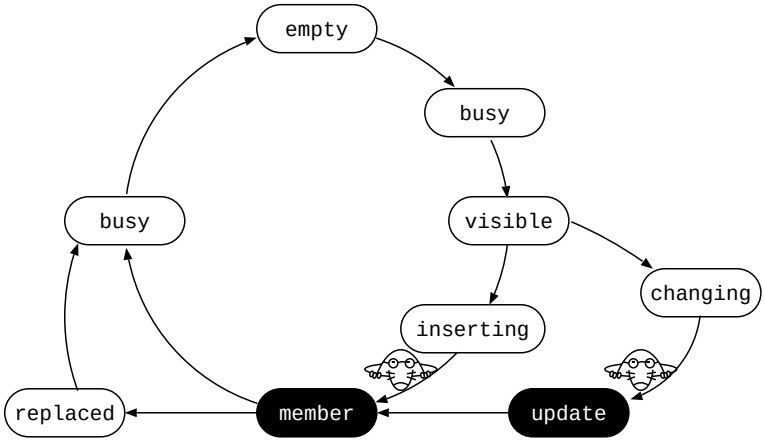


Figure 5.14: *Migrating value replacement* hashtable state machine, simplified. The `collided` state is not shown. Only one bucket may be in a given white-on-black state at any one time for a given key, as guaranteed by the uniqueness algorithm introduced in Section 5.3. See Figure 5.24 for a more detailed diagram.

As with insertion, concurrent updates must first achieve consensus on which value will replace the current one; this is done with the lock-free whack-a-mole algorithm described above, working with a pair of states, `changing` and `update`, which mirror the `inserting` and `member` states. Each key will have at most a single `update` bucket at any one time. (Figure 5.15)

Once an update value has been chosen, the `member` bucket is moved into `replaced` state, the linearization point of the replacement. A read encountering a bucket in `replaced` state must look elsewhere for the current value associated with the key. Finally, the `update` bucket can be moved into member state, and the `replaced` bucket reused. (Figure 5.16)

To allow the replacement algorithm to determine exactly what value is being replaced in the face of concurrent assistance, the `visible` state now serves the extra purpose of allowing a replacement to scan the probe sequence. Concurrent insertions and replacements must therefore move any bucket in `visible` state to `collided`, as well as those in `inserting` and `changing`. If the current value changes, the bucket will be knocked out of `visible` state, allowing the thread to rescan the probe sequence later.

As it stands, this modification requires lookups to take an atomic snapshot of the probe sequence if no key is found. This can be done by summing the version counters and looping until the sum remains unchanged between two sweeps of the sequence. This overhead is needed because finding no copy of the key in any bucket in isolation no longer guarantees a period of time when the key was not present in the table; the key may simply have been moved by a concurrent replacement. Snapshots are often needed during update operations, too.

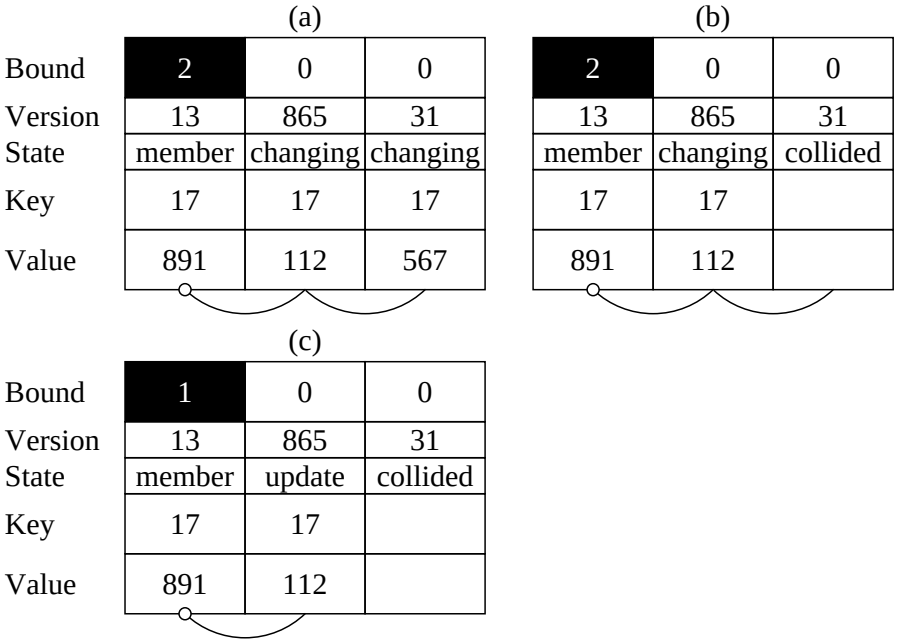


Figure 5.15: Migrating value replacement: A thread attempts to replace the value associated with key 17 from 891 to 112. The `changing` state represents a replacement 'mole' in the whack-a-mole consensus algorithm (a). Obstructing moles must be 'whacked' into `collided` state (b) before the replacement mole can move into `update` state (c).

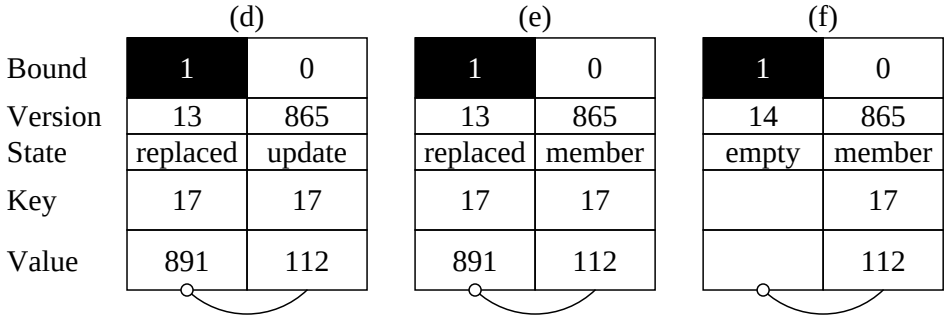


Figure 5.16: Once a unique replacement has been chosen, the current `member` bucket is moved into `replaced` state (d), the `update` bucket is moved into `member` state in turn (e), and the `replaced` bucket emptied (f).

5.6.2 In-Place

An alternative approach is to replace values in-place, rather than migrating the key. This approach allows faster lookups: as with the original set algorithm, a search finding no key can be sure there was a point in time when that key was not present. However, it requires further extensions to the state machine, shown in Figure 5.17.

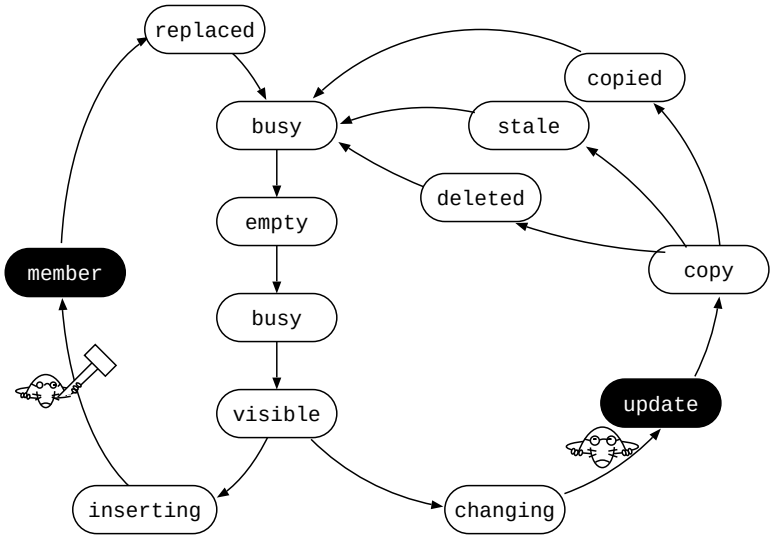


Figure 5.17: *In-place value replacement* hashtable state machine, simplified. Update buckets are no longer promoted to member state. Once again, the collided state is not shown. See Figure 5.24 for a more detailed diagram.

As before, the whack-a-mole algorithm is used to reach consensus on a single `update` bucket, and the current `member` bucket is moved to `replaced` state. Next, the `update` bucket is moved into `copy` state, and the replaced value is overwritten (Figure 5.18). To ensure linearization, subsequent operations must change the state of the `copy` bucket before touching the `replaced` one, and this requires three further states: a successful in-place update will move the bucket to `copied` state before returning the `replaced` bucket to `member` state (Figure 5.19); a concurrent deletion will move the bucket to `deleted` state before moving the `replaced` bucket to `busy` state (Figure 5.20); and a new replacement will move the bucket to `stale` state before promoting its own `update` bucket to `copy` state (Figure 5.21). All three can be assisted by concurrent operations once `copy` state has been left.

(Note that the actual in-place write of the new value cannot be assisted by concurrent threads; nor can the bucket be reused while the write is in progress. However, as the current value of the partial function will always be stored in another bucket, system-wide progress is never blocked.)

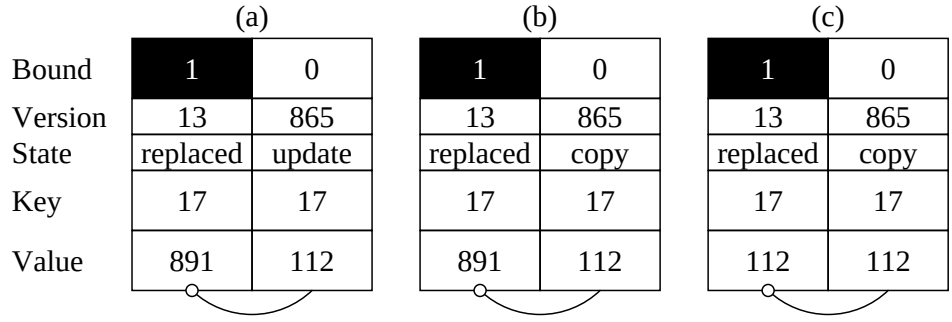


Figure 5.18: In-place value replacement: A thread attempts to replace the value associated with key 17 from 891 to 112. Once consensus on a unique replacement has been reached (a), the `update` bucket is moved into `copy` state (b), and the new value copied into the `replaced` bucket (c).

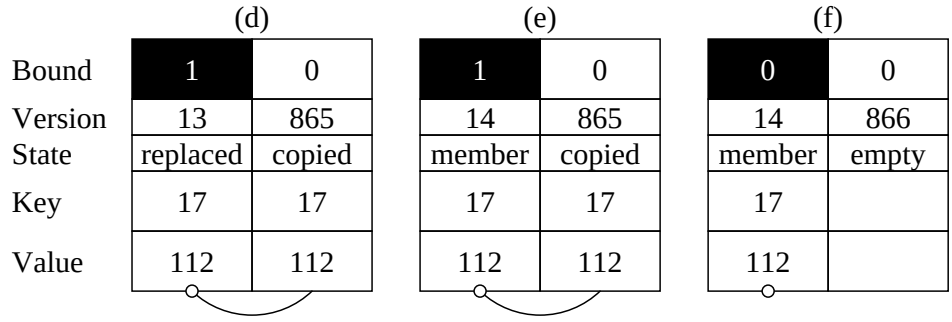


Figure 5.19: When the new value has been copied, the `copy` bucket is moved into `copied` state (d) before returning the `replaced` bucket to `member` state with a higher version count (e), and finally emptying the `copied` bucket (f).

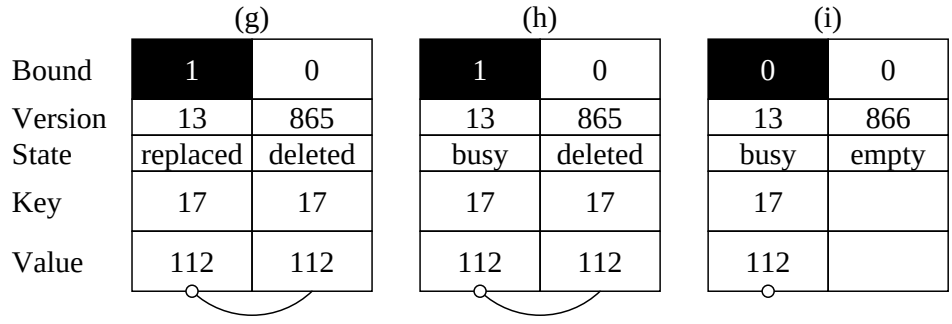


Figure 5.20: Alternatively, a concurrent operation may delete the key–value pair by moving the `copy` bucket to `deleted` state (g) before moving the `replaced` bucket into `busy` state (h) and emptying the `deleted` bucket (i).

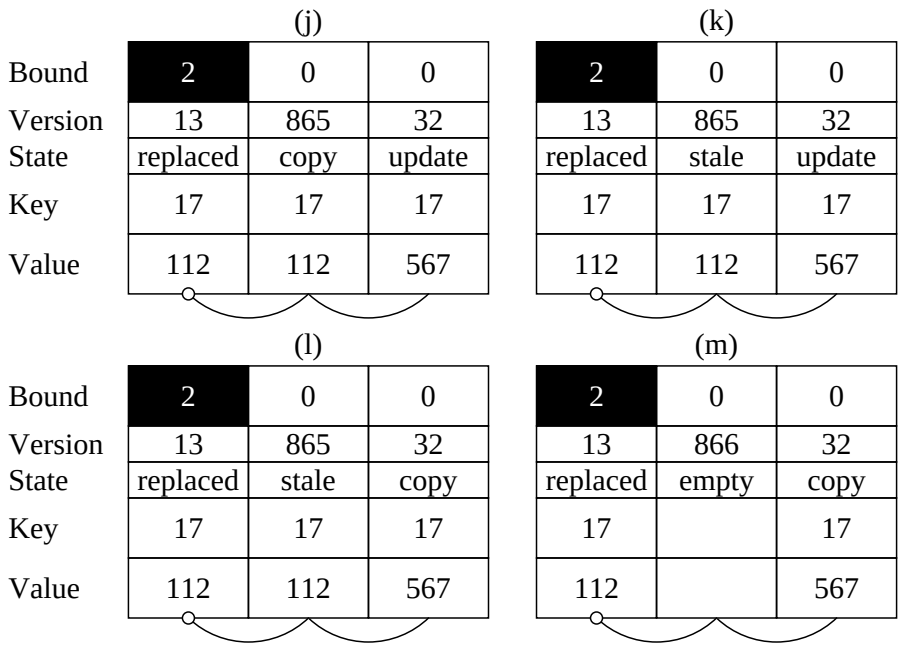


Figure 5.21: Alternatively, concurrent operations may reach consensus on a new replacement value (j), move the current `copy` bucket to `stale` state (k) and the `update` bucket into `copy` state (l), and finally empty the `stale` bucket (m). The thread copying the stale value in-place will then have to locate and copy the new value.

5.6.3 Compacting Hybrid

I have described both migration and in-place replacement as they both have benefits: the latter has cheaper operations in general, especially lookup misses, which can use the same single-pass algorithm as the hashtable-based set algorithm; while the former allows keys to be safely migrated to new, better-situated buckets without changing the associated value.

In fact, both styles of replacement can be used within the same hashtable. At first glance, this seems to complicate the migratory algorithm without providing the cheaper operations that in-place replacement allows. However, by constraining the migration of keys, using in-place replacement otherwise, the single-pass read algorithm can still be used.

Suppose a per-key partial order $<_k$ exists on the buckets, such that bucket B can only be moved from update to member state if the replaced bucket R satisfies $R <_k B$. For a simple example, suppose $<_k$ orders buckets in the opposite order to the standard probe sequence order used earlier (quadratic probing); keys can only migrate to an earlier position in the standard probe sequence. In combination with probe bounds, this allows long probe sequences with lots of holes to be *compacted* by migrating the keys and shrinking the probe bound (Figure 5.22). Further, key replacement will naturally migrate keys to the earliest position in the standard probe order.

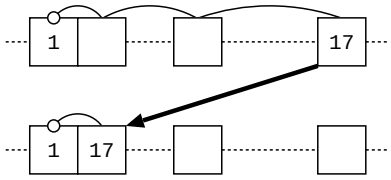


Figure 5.22: Key 17 migrates, allowing the probe sequence bound to be reduced.

Suppose also that all scans of the probe sequence respect $<_k$, i.e. any scan for key k scanning buckets B and C, where $B <_k C$, must read C *after* B. In the simple example, that means scanning the probe sequence in the opposite order from earlier code. Since keys can now only migrate ahead of a concurrent scan, not behind it, a single pass is sufficient to ensure linearizability (Figure 5.23). This means the costly multiple-pass snapshot of the basic migration scheme is no longer required.

I call such a hybrid in-place–migratory system a *compacting hybrid*. A compelling use-case for the compacting hybrid model is explored in Section 5.8.

Figure 5.24 gives the full state machine of the hybrid model, including necessary conditions on state changes. Positive conditions (e.g. "replaced") indicate that a state transition can only be made after observing another bucket in one of the given states for the same key. Negative conditions (e.g. "no member")

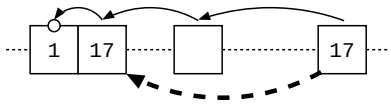


Figure 5.23: If, during a scan, a key is always present in the table, it may be seen more than once (due to concurrent migration), but it will never be missed.

indicate that a transition can only be made after observing every other bucket, and finding none in any of the listed states for the same key.

For instance, the `visible` → `changing` state transition for a bucket containing key 5, say, can only be performed after observing another bucket holding key 5 in either `member` or `replaced` state; the value observed in that bucket will be the value replaced if the replacement operation succeeds. However, it cannot be made if, during the scan of the probe sequence, any buckets were found in `changing` or `update` states for key 5. In either of these cases, the thread must assist the concurrent operations to completion before retrying.

The state transition marked '> replaced' (resp. '<= replaced') can only be performed after observing another bucket holding the same key in `replaced` state *before* (resp. *not before*) the bucket undergoing the state transition in the partial ordering; this encodes the rule that keys can only migrate ahead of a concurrent scan.

States with bold outlines are *unowned*. Whichever thread first moves them into an owned state becomes its *owner*; it is then responsible for moving it through any dashed state transitions until it reaches another unowned state. For instance, a thread moving a bucket out of `empty` state becomes its owner until it reaches `copy` or `member` state. If the bucket reaches `collided` state, it is blocked until the owner transitions it to `visible` or `collided` state, allowing the owner to determine whether their operation was successful.

(The compacting hybrid state machine is a superset of the full migrating and in-place machines, so I have not presented similar diagrams for either.)

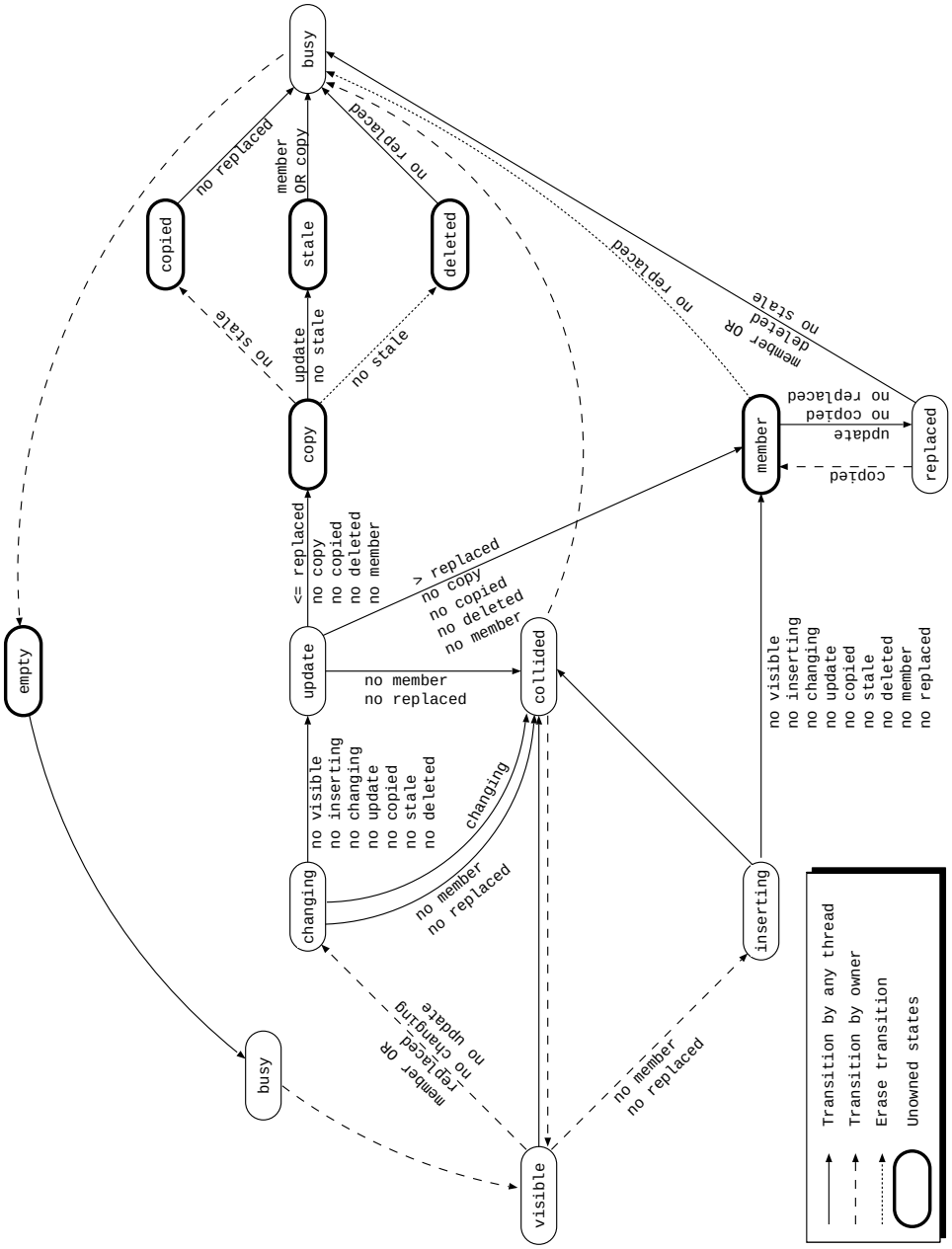


Figure 5.24: Conditions on state changes in the *compacting hybrid* value replacement model. Negative conditions must be observed on all buckets in the probe sequence, while positive conditions need only be observed on one.

5.7 Storing Values on the Heap

When implementing a set, efficiency is maximized by storing each bucket in a single cache-line: the common case is that buckets touched by reads are full, necessitating a read of the key.

When implementing a partial function, the values can be stored on the heap, and a pointer stored in each bucket. This reduces the memory footprint significantly if values are large or occupancy is low; it also allows values of unbounded size to be stored. The pointer can be followed safely as it is protected by the version counter.

The pointer cannot be changed in-place using CAS without garbage collection, as the same address may be reused for a different key-value pair (the ABA problem again). One of the above value replacement schemes must be used, even though only two words need to be changed in the hashtable.

5.8 Dynamic Growth

If the table occupancy becomes too high, a larger section of memory must be allocated and the table entries migrated to the new table. This is best achieved with the compacting hybrid replacement model introduced in Section 5.6: the key-value pairs can simply be replaced with identical pairs located in the new table. This implies every partial order $<_k$ satisfies $O <_k N$ for any buckets O and N, O in the old table, N in the new: that is, lookups must scan the old table before scanning the new one.

A key question is how to determine when to grow the table. Without keeping a count of the number of occupied buckets, growth may occur inappropriately, consuming resources. However, maintaining a single counter in a single cache line for the entire table denies scalability, as all update operations will have to contend for exclusive ownership of the single line.

If counter increments and decrements are to execute in parallel, each thread must modify a unique cache line. Reading such a counter is not population-oblivious, as the footprint grows with the number of threads. However, if a scalable, population-oblivious indicator were available that was highly correlated with table occupancy being excessive, reading the counter could be done only after checking said indicator. Under the reasonable condition of a stable population and sufficient room in the hashtable — a condition that will eventually hold if the population is bounded — such an approach would be *reasonably population-oblivious*.

A simple implementation of a counter is to keep individual increment and decrement fields per thread; since each is monotonic, the whole can be read atomically and lock-free by rereading until two snapshots observe the same set of values. Further, even if two successive snapshots differ, the actual value of the

counter is bounded by the interval $\left[\text{inc}_{\text{before}} - \text{dec}_{\text{after}}, \text{inc}_{\text{after}} - \text{dec}_{\text{before}}\right]$, allowing an informed decision about whether or not to grow the hashtable after only two scans in the majority of cases. I call this a *per-thread counter*.

A highly-correlated indicator is the presence in a probe sequence of a short sequence of occupied buckets (easily detected when looking for an empty bucket) followed by a long sequence of buckets of which a high proportion are occupied. By selecting the length of the latter sequence, the probability of a false positive can be made negligible. Further, as the indicator need only be verified after a mutator finds no empty buckets in the first stretch of a probe sequence, the high cost of the second check will very rarely be incurred. I call this the *chain indicator*.

An alternative indicator is the insertion of a large number of keys by a single thread. If n keys may be inserted before the counter must be read, and there are t threads, the total occupancy of the table is guaranteed to change by no more than $n.t$ between reads. n can therefore be chosen to keep the occupancy within bounds. I call this the *fluctuation indicator*. Provided individual threads tend to insert and delete similar numbers of keys, and provided n can be made large enough to cover fluctuations, this approach will again be reasonably population-oblivious. However, these requirements appear more restrictive than those of the chain indicator.

Another approach is to use the per-thread counter algorithm, but to only allocate an increment and decrement field per processor. On a machine which provides a fast method for determining the current processor ID, this approach is still reasonably disjoint-access parallel, under the assumption that preemption between determining processor ID and incrementing the relevant field is highly improbable, or that threads are each assigned to a single processor throughout their lifetime. If there are many more threads than processors, this approach may decrease the total footprint, and increase performance during periods of growth. I call this a *per-processor counter*.

Another solution is to maintain several counters for different portions of the hashtable, growing the entire hashtable if any individual part becomes over-populated. This approach has the twin advantages of simplicity and straight-line speed, but is still a bottleneck to performance if the number of counters is too low, or if the hash function does not distribute the keys evenly between the (small number of) counters. I call this a *split counter*.

Based on these brief analyses, I would expect the chain indicator to outperform the fluctuation indicator in most situations, while the split counter should produce equal or better performance in cases where the hash function is sufficient to distribute the keys. The per-thread counter should out-perform the per-processor counter, assuming a good indicator, as it does not have the overhead of determining processor ID; however, it does require adding a mechanism for determining each thread's ID.

5.9 Evaluation

In this section, I evaluate the lock-free open-addressed hashtable algorithm built up in this chapter, comparing it against several state-of-the-art hashtable designs from the literature.

5.9.1 Related Work

In order to assess performance, I implemented a range of designs from the literature, which I will now summarize.

Michael presented a lock-free hashtable based on external chaining [63]. The core of the algorithm is the linked list stored in each bucket; the hashtable itself is simply an array of pointers, one per bucket. Searching a linked list is simple: simply traverse the sorted list of keys until the relevant key is found or not. Inserting in the list is a matter of traversing the list until the correct location is found to insert the key, aborting if the key is already in the table, and then a new node is inserted with a single CAS operation on the relevant pointer (Figure 5.25).

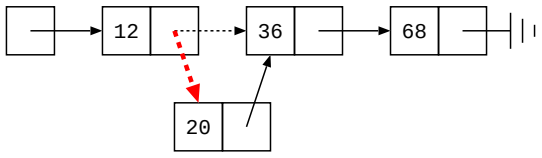


Figure 5.25: Michael's algorithm: To insert a key, use CAS to swap in the new node.

Erasing a key cannot be done simply by swapping out the node containing it with a CAS. To see why not, imagine that the node containing 12 in Figure 5.25 was being deleted concurrently with the insertion depicted. If the erasing thread read down the list before 20 was inserted, it would find the node containing 36 as the successor to 12. If the insertion of 20 now took place, subsequently swapping out the node containing 12 would cause the newly-inserted 20 to be deleted also.

Instead, in Michael's algorithm each node also contains a *deleted* flag, stored in the same word as the next pointer. The first step in a deletion is to set this flag. Any concurrent operation finding a deleted node must then assist the erasing thread by swapping the node out of the list. In the example just given, the concurrent insertion of 20 would not be able to continue once 12 had been marked as deleted without first assisting the erasing thread (Figure 5.26).

These techniques are also used in an earlier algorithm by Harris [34]. The novel part of Michael's approach is to prevent more than one node being removed from the list by a single CAS operation; I will return to this momentarily.

While Michael's hashtable can store any number of items, as the key population to bucket ratio grows, search times degenerate from $O(1)$. Shalev and Shavit

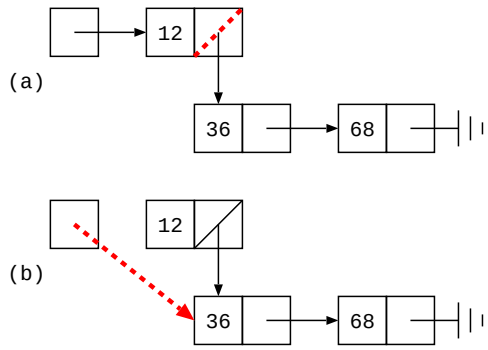


Figure 5.26: Michael's algorithm: To erase a key, (a) mark the node as deleted, then (b) swap it out of the list. This latter step must be assisted by concurrent operations.

addressed this limitation, allowing the number of buckets to grow as the table population does, using a lock-free algorithm they termed 'split-ordered lists'.

In a split-ordered list, every key is stored in a single linked list; the hashtable acts as a fast index into this list. In this way, searches can run safely even if the number of buckets changes concurrently. Each bucket points to a reserved key in the list, called a *dummy node*; in the ordering, a dummy node is less than any key which the bucket may store, and greater than all keys smaller than any key which the bucket may store.

The dummy nodes add overhead when compared with Michael's algorithm: the nodes themselves require space, increasing the size of the hashtable; and all operations must go through an additional level of indirection, namely a dummy node between the bucket and the relevant keys. If the population size cannot be bounded *a priori*, the overhead of a split-ordered list is likely to be less significant than the cost of choosing an incorrect hashtable size that cannot be dynamically varied.

The final hashtable algorithm I compared against is a blocking design by Lea [53]. Written for the Java Concurrency Package, this is regarded as a state-of-the-art blocking hashtable design, combining reasonable disjoint-access parallelism with read-parallelism and population-obliviousness.

Lea's algorithm stores keys in an unsorted list protected by a lock. Inserts happen at the start of the list, once the lock has been taken. Lookups can proceed without locking, simply scanning the list for the relevant key. This is made safe because erasing threads adopt a read-copy-update–style approach: instead of altering the list to remove a node, they duplicate the list up to the removed node, only altering the head pointer (Figure 5.27). Thus, once a lookup has read the head pointer, it has the top of a static list representing an atomic snapshot of the dynamic list.

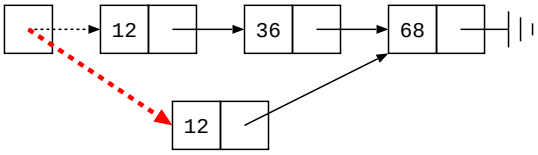


Figure 5.27: Lea's algorithm: To erase a key, the list is essentially duplicated node-for-node, though as an optimization the tail of the list after the erased node can be reused.

None of the algorithms are, as presented, garbage-free: searches assume nodes will not be reused during a scan, preventing the nodes from ever being freed. The final problem, therefore, is how to reclaim memory. Several garbage collection algorithms have been proposed.

In Valois' reference counting method [87], a *reference count* is stored in each node. When a node is allocated, its reference count is initially 1; when the node is removed from the list, its reference count is decremented atomically. When a reader first encounters a node while traversing a list, it increments its reference count atomically; when the reader is finished with the node, it decrements the count again atomically. When the count reaches zero, the node can safely be freed.

The main advantage of this method is conceptual simplicity: it is a staple of concurrent programming. Its main disadvantage is also well-known: it is not read parallel. Multiple readers traversing the same list will create a significant amount of communication on the memory subsystem, severely limiting the scalability of the approach.

In Michael's Safe Memory Reclamation (SMR [62]), each thread has a set of *hazard pointers* which store nodes which are not safe for reuse. When a reader first encounters a node while traversing a list, it *publishes* that node in one of its hazard pointers, then verifies that the node is still in the list. Before reusing a node, a thread must first scan the hazard pointers of all threads; the node is safe for reuse only if it is not found in any hazard pointer. By reclaiming memory lazily, a thread can amalgamate the cost of this scan over many deletions, at the cost of a higher memory footprint.

The main advantage of this method is low overhead and good scalability: since scans only take place when memory must be reclaimed, SMR is read parallel. Unfortunately, since the number of hazard pointers per thread is limited, SMR cannot be applied to all concurrent algorithms. This motivates the use of Michael's linked list design over Harris': the former can use SMR, while the latter cannot. SMR is neither population-oblivious nor disjoint-access parallel, but the runtime costs of both can again be reduced at the cost of a higher memory footprint by reclaiming memory lazily.

In Fraser's Epoch garbage collection [26], each thread has an *epoch number*.

A thread can progress to the next epoch only when all other threads have entered the same epoch, and memory can be reclaimed only after two epochs have passed.

The main advantage of this method is very low overhead together with good scalability: Epoch GC is read parallel. It is neither population-oblivious nor disjoint-access parallel, but the runtime costs of both can again be reduced by reclaiming memory lazily. The chief disadvantage is that the design is blocking: suspension of one thread will prevent other threads from reclaiming memory. Typically, memory use will not grow unboundedly, but will be very high compared with either SMR or reference counting. Unlike SMR, Epoch GC can be used for any concurrent algorithm.

5.9.2 Benchmark

I implemented a range of design combinations from the literature: Michael's hashtable with Epoch collection (M); Michael's hashtable with SMR (M-SMR); Michael's hashtable with reference counting (M-RC); Shalev and Shavit's split-ordered lists with Epoch GC (SS); and Lea's lock-based hashtable with Epoch GC (L), using both a basic spinlock and the MCS lock [61] at different locking granularities.

I compared these against the new compacting hybrid design presented in this chapter (P). The other designs I have covered perform the same actions in the average case for insertion, deletion and lookups; they also have the same performance in this benchmark, assuming all are optimized for this common case. For simplicity, therefore, I have only provided the compacting hybrid results.

My benchmark is parameterized by the number of concurrent threads and by the range of key values used. I present results for 1–24 threads (running on a SunFire 6800 with sixteen 1.2GHz UltraSPARC-III CPUs) and with 2^{15} keys chosen from $[0, 2^{16})$, each mapped to a value chosen from $[0, 2^{16})$. At each step, a random action is performed: a lookup, a move, or a replace. A lookup consists of a single call to the map's lookup function with a key chosen uniformly at random (from $[0, 2^{16})$). A move consists of repeated calls to the map's delete function with keys chosen uniformly at random; once a key has been removed, the map's insert function is called repeatedly with keys and values chosen uniformly at random, until a new key has been inserted. Finally, a replace consists of a single call to the map's replace function with a key–value pair chosen uniformly at random. The relative weighting of lookups, moves and replaces can be varied on starting the test, allowing the costs of each to be determined more accurately.

This set of steps was chosen to keep the number of keys in the table close to 2^{15} at all times. This avoids hashtable resizing, which simplifies my algorithm, as well as allowing a fine locking granularity and greater read-parallelism in Lea's, but which unfortunately negates the benefit of split-ordered lists.

Each trial lasted ten seconds, after a three second warm-up period to fill caches, and trials were repeated 40 times, interleaved to avoid short-lived perfor-

mance anomalies, to obtain a 90% confidence interval.

In all cases, Epoch GC provided better performance than SMR and reference counting, at the cost of a much greater memory footprint. This held true regardless of how lazy SMR was configured to be. Maged's hashtable design also outperformed split-ordered lists, due to avoiding the overhead of allowing table resizing. For clarity, the slower algorithms are not shown in the results. Lea's blocking implementation performs best with low-overhead spinlocking and a fine locking granularity; this is the configuration shown.

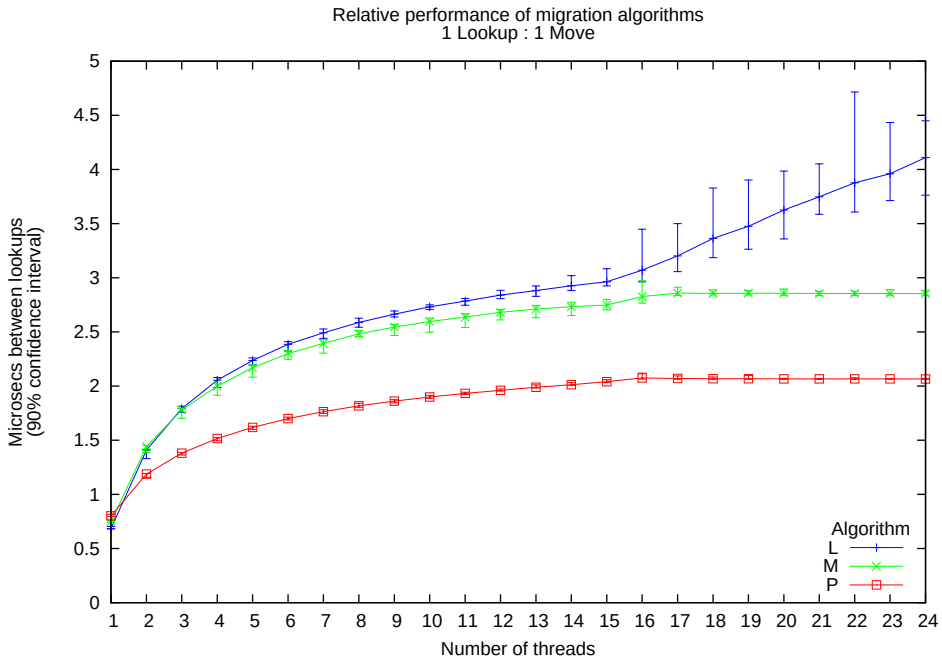


Figure 5.28: Performance of the competing map algorithms, without replacement, on a 16-way SPARC machine; lower is better.

The relative performance of the three different approaches (P, M and L) *without* replacement can be seen in Figure 5.28. M and L are very close while the number of threads is less than the number of processors, with L's overhead growing as the parallelism grows. This is because, despite different approaches, both M and L have identical operation footprints. Above 16 threads, the cost of blocked threads causes significant slowdown and variability in L, while M stays level.

With one thread, the fastest algorithm, L, is 15% faster than P. In all multithreaded tests, however, P is significantly faster than both L and M: over 35% faster with 16 threads. This can be attributed to two causes. First, the live memory of P (the memory accessible from a root pointer) is static, unlike the externally-chained designs. This minimizes capacity misses in the cache. Second, in the common-case code path for update operations and successful lookups, the P algorithm touches fewer cachelines: one rather than two. This lowers the cost

of concurrency misses when the required cachelines are not present in the cache in the required mode (shared or exclusive).

Inter-processor cacheline exchange dominates runtime in massively parallel workloads. By design, the P algorithm minimises this cost for lookups, inserts and erases; this results in the strong performance advantage shown. Applications with much larger, multi-cacheline keys would lose most of this advantage, and may well favour the externally-chained schemes that lower the memory footprint of empty buckets.

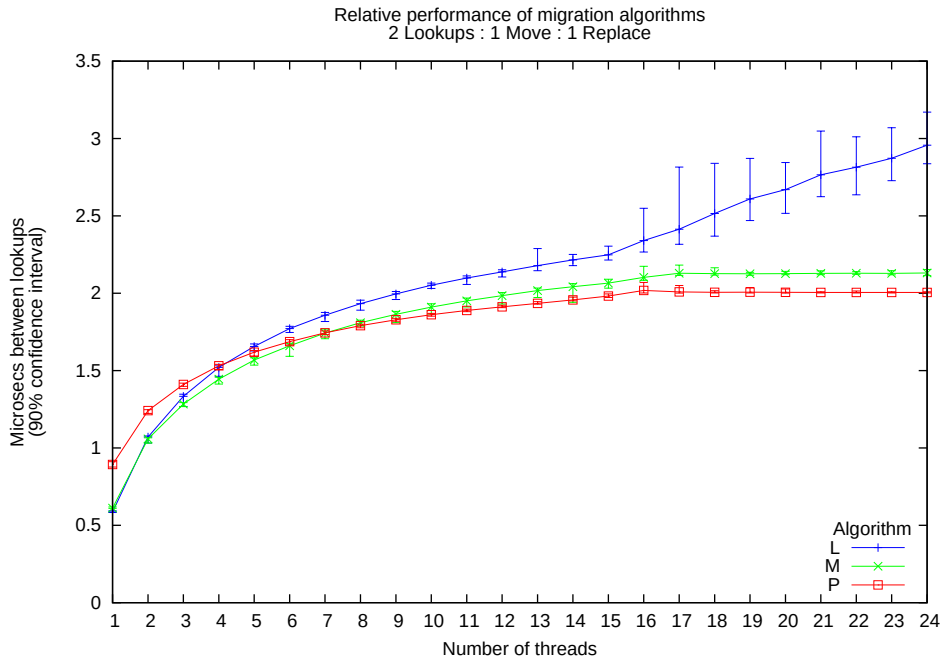


Figure 5.29: Performance of the competing map algorithms on a 16-way SPARC machine; lower is better.

The relative performance of the three approaches *with* replacement can be seen in Figure 5.29. Once again, M and L have similar results. This time, however, L is more than 50% faster than P with a single thread, and even with 16 threads, P is only 5% faster than M. As predicted, the state-machine–based replacement algorithm of P is extremely costly compared with the single CAS required for M. In fact, as Figure 5.30 shows, P is three times slower than M at replacement.

5.9.3 Discussion

The decision to create an algorithm exhibiting locality of reference as well as reasonable scalability has allowed my algorithm (P) to scale better as the number of threads grows; however, the complexity of implementing replacement causes severe penalties for workloads with many replace operations. The algorithm is

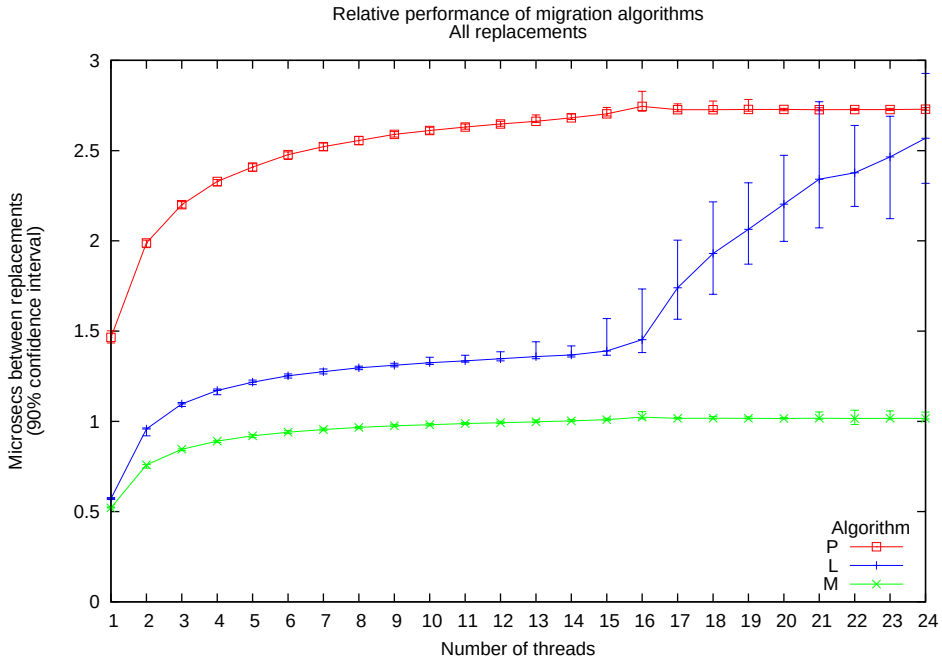


Figure 5.30: Performance of the replacement components of the competing map algorithms on a 16-way SPARC machine; lower is better.

therefore a viable *alternative* to existing externally-chained algorithms, rather than a replacement. Knowledge of the expected ratio of different operations, as well as how dynamic the actual population is likely to be, should inform the choice of algorithm.

The cost of replacement arises from the overhead of guaranteeing lock-free progress. A blocking open-addressed table, where cells are locked during value replacement, may achieve significantly better throughput — but likely at the cost of a performance degradation when the number of threads exceeds the number of processors.

Chapter 6

Diatomic Snapshot-Modify-Update

I have shown in Theorem 4.2.3 that single-word — and even small multi-word — atomic read-modify-update primitives are not sufficient for scalable universality. In Chapter 5, I showed that scalability could still be achieved under reasonable assumptions, but that this restricts the range of applicability of the resulting algorithm. I now approach the problem of extending traditional instruction sets to allow scalable construction of algorithms.

Amdahl's Law [6] means that performance is best increased by optimizing for the common case; hence, any new primitives that impair performance of common operations such as memory accesses will result in an overall negative performance impact. Equally, any proposal that demands extensive investment of silicon space, or requires expensive non-standard inter-chip architectures, would be highly unlikely to be adopted. In addition to theoretical scalability and progress guarantees, I therefore also require a demonstrably low-impact path to adoption.

The second half of my thesis is that there is a stronger primitive than CAS, scalably universally lock-free, and provably implementable without detrimental impact on other aspects of hardware. In this chapter, I introduce my proposed primitive, and illustrate its use with some examples, before showing that the impossibility results of Chapter 4 do not apply to it. In Chapter 7, I will show that the necessary changes can be implemented without incurring detrimental hardware costs.

6.1 Snapshot Isolation

To motivate my choice of primitive, I first describe some existing proposals, each of which is either too weak (not scalably universal) or too strong (too difficult to implement lock-free in hardware). I then introduce the new primitive, as well as a new pseudocode construct that simplifies the presentation of algorithms built

on the primitive.

Hardware transactional memory designs implement atomicity entirely in hardware, and hence can universally provide all four scalability properties. However, the scalable *lock-free* implementations presented thus far require radical changes to memory subsystems, chip designs and instruction set architectures. No evidence is available that these designs can be adopted by hardware designers without a detrimental impact on non-transactional operations.

Discarding lock-freedom on the hardware level is undesirable. Use of weaker progress guarantees such as obstruction-freedom is inappropriate because more complex contention managers (such as Polka [78]) make informed decisions based on detailed information about which operation is blocking what. This information is not currently made available over memory subsystems, and is application-specific. Without it, as work on obstruction-freedom shows, many workloads tend to livelock, denying any progress at all. Lock-free primitives allow this information to be shared between threads even when the algorithms built on them are not lock-free.

One option would be to implement contention management in software. However, without reliable, lock-free primitives, once again only naive contention management such as 'Polite' (exponential backoff) could be adopted, limiting throughput on many workloads.

TM thus appears too strong to meet my requirements, and a weaker primitive, or set of primitives, must be found. How weak can these primitives be and still be scalably universal?

- An *atomic snapshot* reads the values of multiple memory locations at some linearization point. Analysis of the Theorems of Chapter 4 show that an atomic snapshot cannot be emulated without losing one of the scalability properties. A scalably universal primitive set must therefore include an atomic snapshot.

- An *atomic read-modify-update* (RMU) operation reads a location, modifies the value found there, then updates the location with this new value, ensuring that no other updates separate the read from the update. Herlihy showed that a non-trivial read-modify-update operation is necessary for universality; a *scalably* universal primitive set clearly requires one too.

- For scalability, however, *separate* atomic snapshot and RMU operations are not sufficient, as single-location atomic updates demand a separate location, either thread-specific or with a unique garbage value, for each update that can occur disjointly. A scalably universal primitive set must therefore include an operation *combining* an atomic snapshot with an atomic read-modify-update operation.

An *atomic snapshot-modify-update* (SMU) provides a single linearization point at which a coupled snapshot and single-location update appear to occur. However, implementing this lock-free on a standard memory subsystem whilst preserving parallelism is non-trivial. The updated location must be held in exclusive mode, and the snapshot locations in read mode; if these are not grabbed in some total order, deadlock or livelock are inevitable without a contention manager. Even if they are obtained in a total order, exclusive mode may need to be held for a long time to ensure progress — again, the province of a contention manager. The arguments above ruling out TM therefore also apply to atomic SMU.

So: a scalably universal instruction set must include an operation combining an atomic snapshot with an atomic RMU, but the linearizable primitive which combines these, atomic SMU, is too strong. In fact, the primitive I refer to in my thesis occupies the middle ground between separate atomic snapshot and RMU operations and a combined atomic SMU operation. This middle ground is reached by dropping the requirement of linearizability, and adopting a weaker correctness requirement: snapshot isolation.

A transaction implements *snapshot isolation* if all reads are executed as an atomic unit, all writes are subsequently executed as an atomic unit, and the updated locations are not modified between the reads and writes. Any linearizable history is valid under snapshot isolation, but the converse is not true; hence snapshot isolation is a strictly weaker correctness requirement than linearizability.

Snapshot isolation was introduced in the context of databases [13] as a critique of earlier ANSI correctness requirements. Compared to full linearizability, less overhead is required to implement snapshot isolation, allowing simpler, better performing implementations; yet it can implement linearizable transactions [23]. Consequentially, it has been adopted by several major database management systems, such as Borland's InterBase 4 [84]. Hopefully, both these properties will transfer to hardware primitives: regardless of the isolation level of the primitive, the algorithms implemented with it must in general be linearizable to be considered correct.

A *diatomic snapshot-modify-update* operation (henceforth just 'diatomic operation') performs multiple reads and a single write under snapshot isolation. Two diatomic operations with the same footprint can succeed concurrently, with each reading the same set of starting values, provided they update disjoint locations. Diatomic operations are not atomic, as shown in Figure 6.1, but they are stronger than two independent atomic operations.

One example of a diatomic operation is a *diatomic k-compare single-swap* (dkCSS), which atomically verifies the values of k memory locations, and updates one of them with a new value provided the snapshot matched expectations. dkCSS is sufficient, provided k is not bounded, to directly emulate an arbitrary diatomic operation, lock-free, by reading all affected memory locations, calculating the modification required, then performing a dkCSS to verify the k memory locations with a snapshot, retrying if the operation fails. Being the diatomic

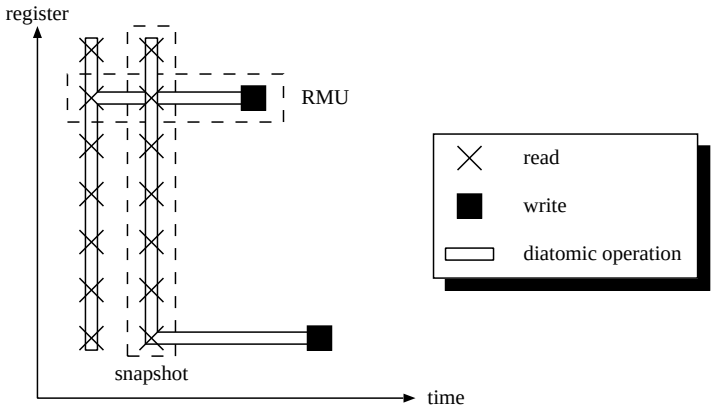


Figure 6.1: Two concurrent diatomic operations both succeed, even though the snapshot of one overlaps the RMU of the other. As neither sees the other's update, neither operation can be linearized after the other, and the history as a whole is not linearizable; yet it is valid under snapshot isolation.

extension of CAS+snapshot, it seems reasonable to assume hardware support for at least dkCSS.

In the following, I will assume the provision of a flexible interface to the hardware's native diatomic operation, where the user issues a sequence of (possibly dependent) reads and then a single write. It should be merely a mechanical exercise to build this from whatever instructions the hardware may expose, and its use greatly simplifies the presentation of algorithms.

To mark reads and writes as part of a diatomic snapshot-modify-update, I will enclose them in a `diatomically` construct. Note that this is simply a notational convenience used to present algorithms, not a proposal for extending programming languages.

Since it may be important to update local memory during a diatomic operation, publishing a pointer to the memory with the single swap, writes to thread-local memory should be allowed in the construct, with the assurance that these writes happen before the diatomic update. Any allocated memory should be freed if the update fails. Supporting these writes is, again, merely a mechanical exercise from any reasonable hardware primitive.

I now motivate the new primitive with some examples.

6.2 Value Replacement

My first example of using diatomic operations returns to the problem of value replacement in hashtables. As mentioned in Section 5.6, the problem with replacing the value associated with a key, even when values are stored externally, is that the CAS may be delayed, and subsequently alter the wrong key–value pair. The approach required for CAS was therefore quite complex and intricate. I now apply diatomic operations to find three alternative solutions of increasing complexity.

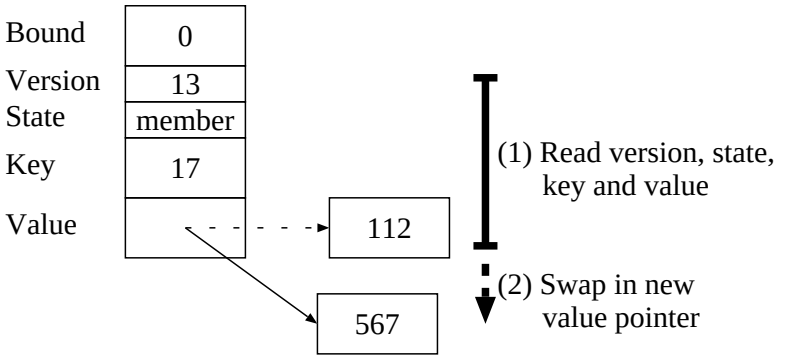


Figure 6.2: The simplest scalable solution combines reading the key–value pair (1) with the update of the value pointer (2) diatomically.

My first solution is to simply combine the read of the key–value pair and the update of the value pointer diatomically, as shown in Figure 6.2. Provided the value pointer is overwritten when the key is removed from the table (say by `NULL`), the diatomic snapshot-modify-update will fail if the key–value pair changes between the snapshot and the update; hence if the update succeeds, it has safely replaced the value for the correct key. Lookups must use an atomic snapshot to read the value if the key matches: the version counter is no longer modified when the value pointer changes, so in the absence of garbage collection, a non-atomic snapshot may return garbage data.

This algorithm involves far fewer operations on shared memory than the ones using CAS (see Figure 5.24). In the common case for a compacting hybrid replacement, the probe sequence will be a single bucket long, containing the key–value pair being replaced. Seven CAS operations will be required for the in-place update, and a further three to increase and decrease the probe bound: ten CAS operations. In contrast, the diatomic-based code requires a *single* CAS-like update operation.

Pseudocode implementing this algorithm is shown in Figure 6.3. Note that the `diatomically` construct wraps the allocation of a new `Value`. As mentioned before, if the diatomic update fails, this should be transparently freed to prevent

```
1   bool set::Replace(Key k, Value value): // Replace value associated with key k
        h := Hash(k)
3       max := GetProbeBound(h)
        for i := 0 .. max
5           ⟨version,state⟩ := Bucket(h,i)→vs
            if state = member ∧ Bucket(h,i)→key = k
7               diatomically
                    if Bucket(h,i)→vs = ⟨version,state⟩
9                       new_ptr := new Value(value)
                        old_ptr := Bucket(h,i)→val_ptr
11                      Bucket(h,i)→val_ptr := new_ptr // Diatomic update
                        delete old_ptr
13                      return true
        return false
```

Figure 6.3: Code to replace the value associated with a key in a hashtable, using the `diatomically` construct. For simplicity, the function does not return the value replaced; this can be addressed.

visible side-effects.

Determining linearization points for this algorithm is easy. For reads, the linearization point of the underlying snapshot operation is sufficient; for writes, that of the diatomic update. The remaining example algorithms in this chapter have similarly uninteresting linearization points.

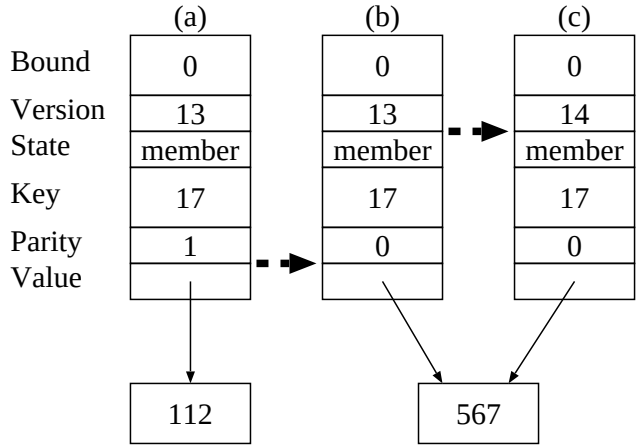


Figure 6.4: An alternative solution allows the version counter to change when the value does, allowing safe concurrent assistance with a parity bit. An update finding a bucket with the relevant key (a) first updates the parity–value pair (b); any thread can then correct the resulting version–parity mismatch by incrementing the version counter (c).

An alternative solution to the problem is shown in Figure 6.4. Changing both version counter and pointer simultaneously is not possible with diatomic operations, as they do not fit into a single word. Instead, I reserve a single bit in

the value pointer to store the *parity* of the associated version count; if the value pointer changes, the parity must be flipped; and if a lookup finds the parity of the value pointer does not match the version count, it must increment the version count and retry. This allows lookups to use the original version-counter read algorithm in the common case of no contention. This may increase performance, depending on the efficiency of diatomic snapshots.

While a lookup could safely read the pointer even when the parity does not match, it would be difficult, if not impossible, to linearize the resulting implementation in the face of concurrent deletions. If lookups assist concurrent replacements, a replacement can be linearized to the update of the version counter (or to just before the update, if the update performs a delete).

Note that this approach requires lookups to loop until a snapshot has been taken without the version counter changing, whereas previously a lookup could skip a location if the version count changed, as this would only happen if a concurrent delete linearized. The modified replacement and lookup algorithms are shown in pseudocode in Figures 6.5 and 6.6.

```
1   bool set::Replace(Key k, Value value): // Replace value associated with key k
        h := Hash(k)
3       max := GetProbeBound(h)
        for i := 0 .. max
5           do // Read cell atomically
                ⟨version,state⟩ := Bucket(h,i)→vs
7               if state = member
                    key := Bucket(h,i)→key
9                   if key = k
                        ⟨old_ptr,old_parity⟩ := Bucket(h,i)→⟨val_ptr,parity⟩
11          while state = member ∧ Bucket(h,i)→vs ≠ ⟨version,state⟩
            if state = member ∧ key = k
13              diatomically
                    if Bucket(h,i)→vs ≠ ⟨version,state⟩
15                      return true // Concurrent update has linearized
                    else if old_parity ≠ (version | 1)
17                      Bucket(h,i)→vs := ⟨version + 1,state⟩ // Diatomic update
                        return true // Concurrent update has linearized
19                  else
                        new_ptr := new Value(value)
21                      Bucket(h,i)→⟨val_ptr,parity⟩ := ⟨new_ptr,¬old_parity⟩ // Diatomic update
                diatomically
23                  if Bucket(h,i)→vs = ⟨version,state⟩
                        Bucket(h,i)→vs := ⟨version + 1,state⟩ // Diatomic update
25              delete old_ptr
                return true
27      return false
```

Figure 6.5: Alternative code to replace the value associated with a key in a hashtable, using the `diatomically` construct only during updates. Once again, the function does not return the value replaced; this could easily be addressed.

```
    Value set::Lookup(Key k): // Return value associated with k, or NULL if none found
25      h := Hash(k)
        max := GetProbeBound(h)
27      for i := 0 .. max
            do // Read cell atomically
29              ⟨version,state⟩ := Bucket(h,i)→vs // Read cell atomically
                if state = member
31                  key := Bucket(h,i)→key
                    if key = k
33                      ⟨val_ptr,parity⟩ := Bucket(h,i)→⟨val_ptr,parity⟩
                        value := *val_ptr
35          while state = member ∧ Bucket(h,i)→vs ≠ ⟨version,member⟩
            if state = member ∧ key = k
37              if parity ≠ (version | 1)
                    diatomically
39                      if Bucket(h,i)→vs = ⟨version,state⟩
                            Bucket(h,i)→vs := ⟨version + 1,state⟩ // Diatomic update
41              return value
        return NULL
```

Figure 6.6: Alternative code to lookup the value associated with a key in a hashtable, using the `diatomically` construct only during updates.

The third solution to the problem, shown in Figure 6.7, hybridises pointers with an in-place value-overwriting system as used in the CAS-based design. During the replacement period, the version-count–state field is replaced with a pointer to a dynamically-allocated update descriptor, containing the new version-count and the new value. While the version-count–state field contains an address, all operations must use atomic snapshots to read the key–value pair in the bucket, and concurrent mutations must assist the replacement algorithm in copying the new value into the static bucket. This approach retains the locality of reference of the CAS-based in-place replacement algorithm.

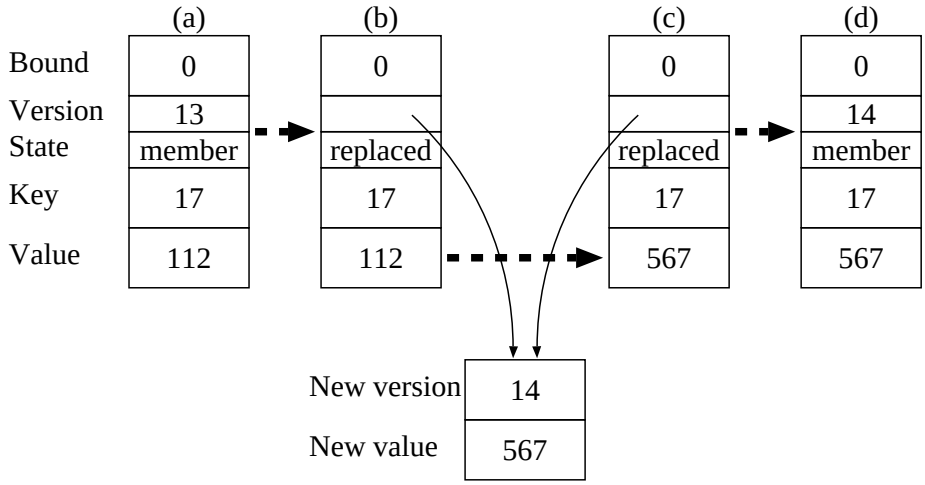


Figure 6.7: The third solution uses in-place copying. An update finding a bucket with the relevant key (a) writes a descriptor into the version–state field (b), updates the value in-place (c), then writes the new version–state pair (d). These last two steps can be concurrently assisted.

6.3 Linked Lists

My second example of using diatomic operations addresses the well-known problem of creating a lock-free linked list algorithm, specifically to implement a set. Michael has presented a lock-free linked list algorithm based on CAS [63]. New nodes are inserted by a single read-modify-update of the relevant next pointer, relying on garbage collection to ensure the node before and after the inserted node are not concurrently reused, which would invalidate the insertion. Nodes are inserted in an absolute ordering based on the stored key, ensuring concurrent operations will read and update the same location in the list for a given key.

Each node has a single bit reserved for a 'deleted' flag in the same machine word as its next pointer. A node is deleted by first setting this flag, then swapping it out of the list. This flag prevents a node's successor changing before it can be swapped out, making the read-modify-update of its predecessor's next node safe. Finally, readers simply follow the list, using the absolute ordering to determine if the relevant key is present or not.

Implementing this same algorithm with diatomic operations removes the need for garbage collection. Readers take a snapshot of the list as far as they need. Inserts and deletes also take a snapshot of the list, coupled diatomically with the needed update; the diatomic guarantee is exactly that provided by garbage collection, namely that the node being updated cannot be reused, nor its successor changed, without the diatomic update failing.

```
1   class LinkedList {
        struct NodeType {
3           Key key
            ⟨bool, NodeType*⟩ ⟨mark, next⟩
5       }
        NodeType* head

7       enum FindR { present, absent, retry }

            // Perform diatomic snapshot (must be enclosed in diatomic block)
9       FindR find(Key key, NodeType*** prev_p, NodeType** cur_p, NodeType** next_p)

    public:
11      bool exists(Key key)
        bool insert(Key key)
13      bool erase(Key key)
    }
```

Figure 6.8: Interface for a linked list-based set built on diatomic operations.

Pseudocode for this adaptation can be found in Figures 6.8–6.12. This approach extends to the externally-chained hashtables Michael presented based on his linked lists; I do not present this explicitly here.

Note that the basic linked list algorithm adapted by Michael is not disjoint-access parallel when implementing a set: any two updates, no matter where they occur in the chain, must conflict on the earliest updated location in the chain, which the operation updating a later location must read. As such, it is not

```
    bool LinkedList::exists(Key key):
17      while true
            diatomically
19              switch find(key, &prev, &cur, &next)
                    case absent:
21                      return false
                    case present:
23                      return true
                    case retry:
25                      break
```

Figure 6.9: Public lookup function. Attempts to find the given key, using a diatomic construct to take a snapshot of the list.

```
    bool LinkedList::insert(Key key):
27      while true
            diatomically
29              switch find(key, &prev, &cur, &next)
                    case present:
31                      return false
                    case absent:
33                      node := new NodeType
                        node→key := key
35                      node→⟨mark, next⟩ := ⟨false, cur⟩
                        *prev := node // Diatomic update: swing in new node
37                      return true
                    case retry:
39                      break
```

Figure 6.10: Public insert function. Diatomically locates the correct location and swings a new node into the list.

surprising that neither Michael's algorithm nor my adaptation is disjoint-access parallel.

However, using linked lists to store external chains in a hashtable preserves disjoint-access parallelism up to the granularity of the chosen hash function: updates to keys with the same hash value will not run independently in parallel.

```
    bool LinkedList::erase(Key key):
41      nodeIsDeleted := false
        while ¬nodeIsDeleted
43          diatomically
                switch find(key, &prev, &cur, &next)
45                  case absent:
                        return false
47                  case present:
                        cur→⟨mark, next⟩ := ⟨true, next⟩ // Diatomic update: mark node as deleted
49                      nodeIsDeleted := true
                        break;
51                  case retry:
                        break;
53      while true
            diatomically // Ensure node is removed
55              if find(key, &prev, &cur, &next) ≠ retry
                    return true
```

Figure 6.11: Public erase function. Diatomically locates the target node and marks it as logically deleted, before running the find function repeatedly to ensure the node is removed.

```
57  FindR LinkedList::find(key, prev_p, cur_p, next_p):
        *prev_p := head
59      while true
            ⟨pmark, *cur_p⟩ := **prev_p
61          if *cur_p = NULL
                return absent
63          ⟨cmark, *next_p⟩ := (*cur_p)→⟨mark, next⟩
            ckey := (*cur_p)→key
65          if ¬cmark
                if ckey = key
67                  return present
                if ckey > key
69                  return absent
                else
71                  *prev_p := &(*cur_p)→next
            else
73              **prev_p := *next_p // Diatomic update: swing out deleted node
                delete *cur_p
75              return retry // Must reenter diatomic construct
```

Figure 6.12: Private find function for linked list. If a marked node is found, diatomically swings it out, deletes it, and instructs the caller to retry. Otherwise, finds the location for the given key in the absolutely-ordered list, returning whether or not the key is present.

6.4 Unbalanced Binary Trees

For my third example of using diatomic operations, I present an algorithm that implements a lock-free unbalanced binary tree with immediate and arbitrary memory reuse. This is quite intricate, so for simplicity I describe the required tree transformations pictorially, providing only a small sample of pseudocode.

The basic tree algorithm I adapt stores all keys in the leaves. This increases the memory footprint, but greatly simplifies the algorithm as interior node deletion need not be implemented. It is also a sensible choice from a performance perspective: deleting a node high up in the tree disrupts a disproportionate number of concurrent operations, decreasing potential parallelism.

Each interior node has a key field, a left and a right pointer, and a control field. The first three are used as in a single-threaded binary tree: the left pointer is the root of a binary tree whose keys are all guaranteed to be strictly less than the key stored in the node; while the right pointer is the root of a binary tree containing all remaining keys. The control field stores any information needed to assist on-going updates to the node. As I will show, it is enough for the control field simply to point at another node, or to NULL if no modification of the node is in progress.

The structure and interface for the tree is shown in pseudo-C++ in Figure 6.13. For simplicity, the leaves also have left, right and control fields, which will always be NULL.

```
1   class Set
        {
3   private:
        struct Node
5           {
            Key key
7           Node* left
            Node* right
9           Node* control

            Node(Key k, Node* l := NULL, Node* r := NULL):
11              key := k
                left := l
13              right := r
            }
15      Node* head := NULL

            // Assist all operations in-progress on the path leading to k
17      void assist(Key k)

    public:
19          // Return whether key k is in the set
        bool exists(Key k)

21          // Insert key k, or return false if it is already present in the set
        bool insert(Key k)

23          // Delete key k, or return false if it is not present in the set
        bool delete(Key k)
25      }
```

Figure 6.13: Interface and data types for a lock-free unbalanced tree.

The steps needed to insert a node are shown in Figure 6.14. An insertion is in progress whenever a node's control field points at a leaf, unless that leaf is already a child of the node. It is therefore possible to identify which stage an insertion is at, and assist it to completion.

The steps needed to delete a node are shown in Figure 6.15. A deletion is in progress whenever a node's control field points at another interior node, or at a child of the node. It is therefore again possible to identify which stage a deletion is at, and assist it to completion.

There are three special cases when the leaf being inserted or removed is very close to the root of the tree. Inserting a leaf into an empty tree, or a tree with a single node, is very simple as there are no control nodes to update; a simple update to the root pointer will complete the operation. Similarly, deleting the last node of a tree is a single update. None of these cases need concurrent assistance.

The last special case is deleting a leaf two indirections from the root; in this case, the topmost control field in the tree should be updated analogously to (e)–(f), but the root pointer can then be modified directly to complete the operation, as shown in Figure 6.16.

To simplify the coding of the above algorithms, I implemented an `Assist` function (not presented), whose job it is to descend the tree, using a supplied key to pick a path, and complete any concurrent operations along the way. Inserting or deleting a node is then a simple matter of completing the first step of each operation, then calling the `Assist` function to clean up the tree. These are the seven states the `Assist` function must identify, and how to handle them:

Insert state 1. An interior node's control field points to a leaf, and the relevant child node (on the left if the leaf's key is less than the interior node's key, on the right otherwise) is a different leaf. Proceed as in (b)–(c).

Insert state 2. An interior node's control field points to a leaf, and the relevant child node is another interior node. Proceed as in (c)–(d).

Delete state 1. An interior node's control field points to one of its children, a leaf, and its parent's control field is NULL. Proceed as in (f)–(g).

Delete state 2. An interior node's control field points to one of its children, another interior node (whose control field will point to the node being deleted). Proceed as in (g)–(h).

Delete stage 3. An interior node's control field points to an interior node which is not one of its children. Proceed as in (h)–(i), and then free both the removed node and the leaf pointed to by its control field.

Stunted delete state. The control field of the top node of the tree points to one of its children, a leaf. Swap the node and its leaf out of the tree as in (j)–(k).

Stable state. All control fields point to NULL. No steps remain.

With this function, implementing insertion and deletion is now simple. Pseudocode can be found in Figures 6.17 and 6.18, respectively.

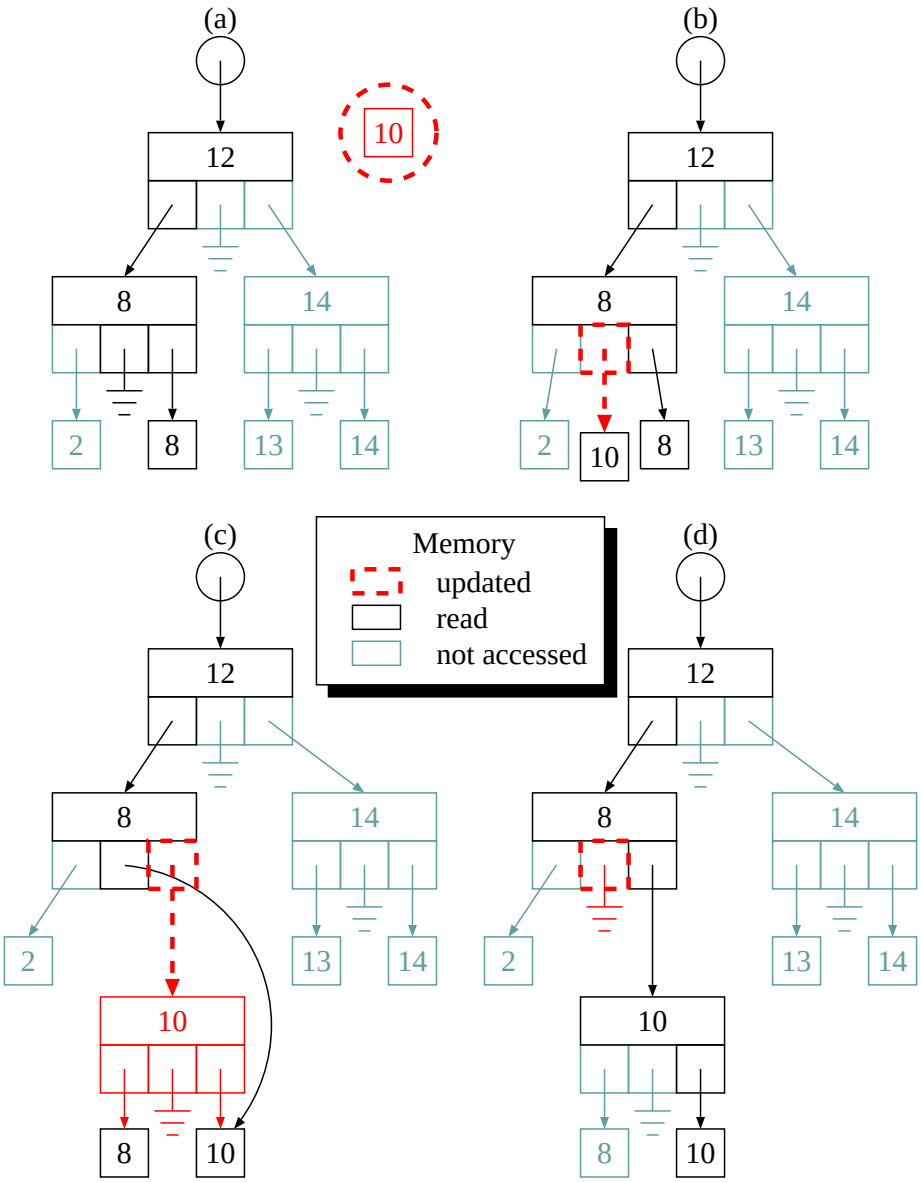


Figure 6.14: Steps in an example insertion of key 10. A thread encountering the tree in state (a) first descends the tree, searching for the correct place to insert the leaf, and ensuring no concurrent operations are in place that would obstruct it. In (b), the thread posts its new leaf into an existing node's control field. Any contending concurrent operations will now assist the insertion to completion, though searches will not yet find the new leaf. In (c), the thread swaps in a new interior node, making the new leaf visible to concurrent searches. Finally, in (d) the thread returns the control field to NULL.

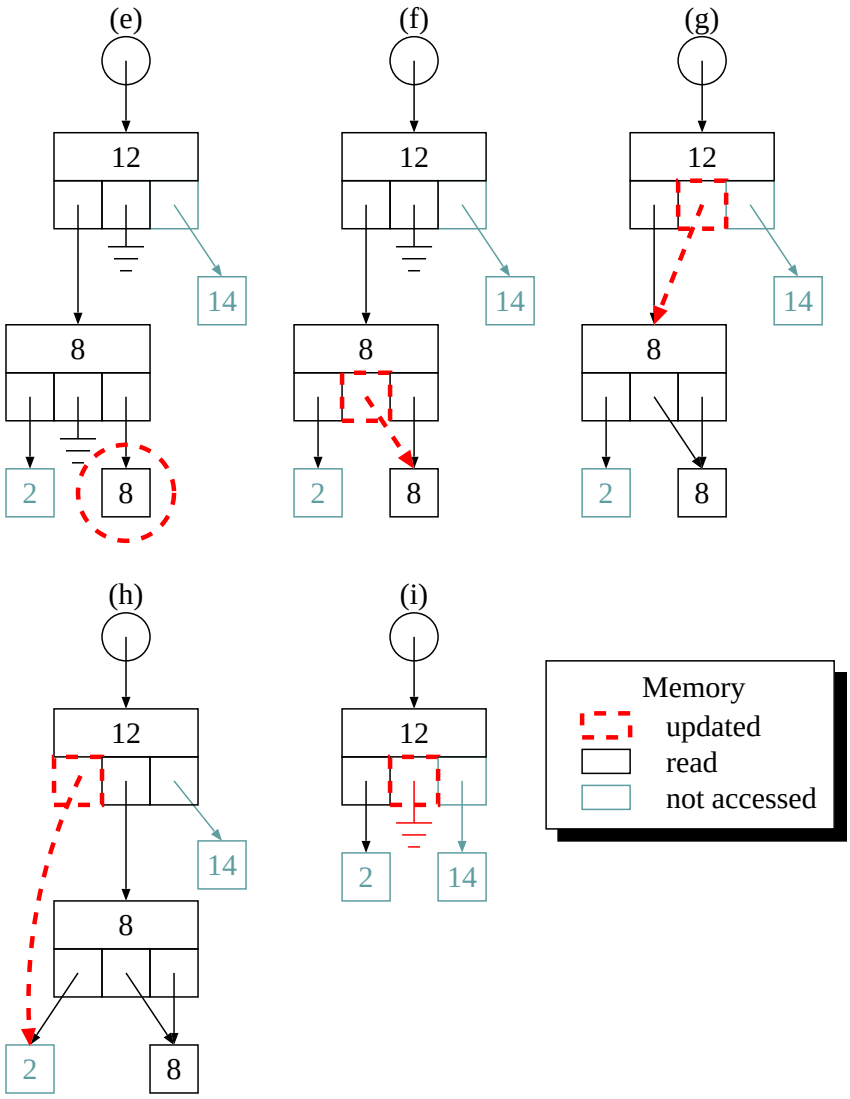


Figure 6.15: Steps in an example deletion of key 8. A thread encountering the tree in state (e) first descends the tree, searching for the correct leaf, and ensuring no concurrent operations are in place that would obstruct it. In (f), the thread posts the leaf into its parent node's control field. Any contending concurrent operations will now assist the deletion, though searches will still see the leaf in place. The thread will now take steps to remove this parent. In (g), the thread now posts the parent node to the grandparent node's control field. To see why this is necessary, imagine that the uncle leaf (containing 14) is concurrently removed, and note that the grandparent would be removed by this operation. This conflict must be prevented before the parent node can safely be swapped out. In (h), the leaf and its parent can now be moved out of the tree by pointing the grandparent node at the deleted leaf's sibling. The leaf is no longer visible to concurrent searches. Finally, in (i) the thread returns the grandparent's control field to NULL and frees the deleted nodes.

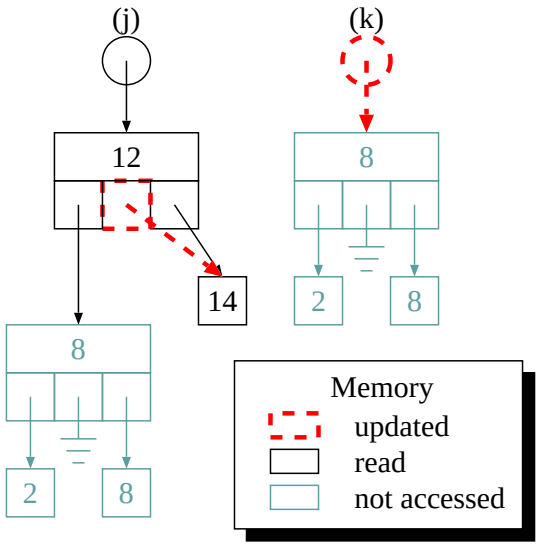


Figure 6.16: Deleting a leaf is simplified if, as in (j), its parent is at the top of the tree: once the parent's control field has been updated, the parent and leaf can be swung immediately out of the tree and freed (k).

```
1   bool Set::insert(Key k):
        insertCompleted := false
3       while ¬insertCompleted
            diatomically
5               parent := NULL
                parentNext := NULL
7               current := NULL
                currentNext := &top
9               currentKey := k - 1
                next := *currentNext
11              while next ≠ NULL
                    parent := current
13                  parentNext := currentNext
                    current := next
15                  currentKey := current→key
                    currentNext := (k < currentKey) ? &current→left : &current→right
17                  next := *currentNext
                if (currentKey = k)
19                      return false // Key is already inserted
                if (parentNext = NULL) // The set is empty
21                  top := new node(k) // Diatomic swap: add the new leaf directly
                    return true
23              if (parent = NULL) // The set only has one member; add the new leaf directly
                    if (k < currentKey)
25                          top := new node(currentKey, new node(k), current) // Diatomic swap
                    else
27                          top := new node(k, current, new node(k)) // Diatomic swap
                    return true
29              if (parent→control = NULL)
                    parent→control := new node(k) // Diatomic swap: post new leaf in control field
31                  insertCompleted := true
            assist(k) // Complete our operation, or any conflicting ones
33      return true
```

Figure 6.17: Insertion into the unbalanced tree, using the diatomically construction to ensure thread-safety (pseudocode continued in Figure 6.18)

```
1   bool Set::delete(Key k):
      deleteCompleted := false
3     while ¬deleteCompleted
        diatomically
5         parent := NULL
          current := NULL
7         currentKey := k - 1
          next := top

9         while next ≠ NULL
            parent := current
11          current := next
            currentKey := current→key
13          next := (k < currentKey) ? current→left : current→right

          if (currentKey ≠ k)
15          return false // Key is not present

          if (parent = NULL) // The set only has one member
17          top := NULL // Diatomic swap: delete member directly
            delete current // Free memory immediately
19          return true

          if (parent→control = NULL)
21          parent→control := current // Diatomic swap: flag the node for deletion
            deleteCompleted := true
23      assist(k) // Complete our operation, or any conflicting ones
      return true
```

Figure 6.18: Deleting from the unbalanced tree.

6.5 Universality: Scalability and Progress

In the last few sections, I have presented scalable solutions to three problems using diatomic operations. Though the primitive itself only satisfies snapshot isolation, the algorithms built from it have all been linearizable; this still remains the basic correctness requirement.

The next question that arises is: can diatomic operations *universally* provide scalable, linearizable implementations of arbitrary atomic operations? To conclude this chapter, I show the answer is yes. Together with Theorem 4.2.3, this demonstrates that diatomic operations are strictly stronger than single-word primitives.

I first providing a blocking implementation, then one with a guarantee of progress. The former is a practical construction, while the latter is intended merely to answer theoretical questions.

6.5.1 Scalability

First, I show that diatomic operations can implement scalable *lock-based* designs, such as a cacheline-granularity blocking transactional memory. The key step is to use atomic snapshots to scalably implement a revocable shared mode lock on a spinlock.

Theorem 6.5.1 *Diatomic snapshot-modify-update operations admit scalable, parallelism-preserving lock-based designs.*

Proof To prove this theorem, I model memory as a set of objects, $\mathbb{J}$, and describe how to scalably implement an arbitrary logical atomic operation; to illustrate, I provide pseudocode for a multi-object compare-and-swap primitive.

I assign each object $j \in \mathbb{J}$ a unique spinlock, $mutex(j)$, each of which can be held in *exclusive mode*, or in *revocable shared mode*. Exclusive mode is obtained by flipping the spinlock from `free` to `held`. Revocable shared mode is obtained by reading the spinlock in `free` state as part of an atomic snapshot; it is revocable as a concurrent operation can at any time obtain the spinlock for exclusive access, causing the atomic snapshot to fail.

A logical atomic operation is performed by atomically obtaining all objects in the operation's footprint in either exclusive or revocable shared mode — the operation's linearization point — before updating those objects held in exclusive mode and releasing the exclusive locks. (A revocable shared lock cannot be, and need not be, explicitly released.) Note that any information required from any object held in revocable shared mode must be read and stored prior to the linearization point, since after this point these objects may validly be mutated by concurrent threads.

To prevent deadlock, I impose on the shared objects a total order $<_l \in \mathbb{J} \times \mathbb{J}$. If an operation encounters an object j held in exclusive mode by another thread,

it must release any exclusive locks it holds on any objects j' with $j <_l j'$ before negotiating exclusive access to j and continuing. This can be partially avoided by obtaining exclusive access to all objects in this order. However, two concurrent operations may each obtain exclusive access on an object held in revocable shared mode by the other, in which case one must release its lock to prevent deadlock.

This rollback mechanism is similar to schemes used in non-blocking algorithms; however, it does not require update logging and the attendant data duplication as no memory locations are updated until the operation is guaranteed to succeed.

As it stands, this implementation is already parallelism-preserving and scalable. However, to allow a lower memory footprint in common algorithms, the memory used for spinlocks must be free for reuse for other purposes, for instance to be returned to the operating system, when they are no longer referenced by root nodes. (I assume that reading from such memory locations simply yields garbage values; on systems where memory protection exceptions are triggered, a standard approach to catching and recovering from such exceptions will be required.)

One solution is to impose a further restriction: each object must be obtained in revocable shared mode before any can be obtained in exclusive mode. Each lock can now be obtained for exclusive access in the order determined by $<_l$. The pseudo-code in Figure 6.19 uses this approach to implement a scalable atomic multi-object compare-and-swap primitive. This takes an array of objects, $objs$, which is assumed to be pre-sorted by $<_l$; an array of expected values, exp; and an array of new values, $swap$, which will be written in atomically only if all objects match their expected values. N is the size of the arrays.

Alternatively, in cases where the object reference graph is acyclic, each lock can be obtained in exclusive mode as it is reached, without causing deadlock. This optimization can be used for e.g. an unbalanced binary tree.

Replacing spinlocks with queue-based locks allows a thread to *request* exclusive mode on an object without immediately obtaining it; this exclusive access must subsequently be granted. A thread which obtains each lock in exclusive mode as it is reached can now avoid deadlock when blocked by another thread on an object j by requesting exclusive mode on j, then releasing any exclusive locks it holds on all objects j' with $j <_l j'$. This avoids the need to subsequently hold all these lock in revocable shared mode simultaneously, which spinlocking requires if the spinlock may be reused.

Allowing locks to be held in exclusive mode as they are reached is especially valuable if the maximum size of a snapshot may be constrained by hardware: it allows an algorithm to 'fall back' to non-scalable exclusive locking if the hardware cannot snapshot sufficient objects.

The linearization points of this algorithm, and the one in the next subsection, are interesting. Unlike earlier algorithms, which linearize at a single update which changes the basic structure — for linked lists, when a node is marked as deleted;

```
1   bool MultiObjectCompareAndSwap(int N, Object** objs, Object* exp, Object* swap):
        enum { retry, update, abort, wait } todo
3       should_hold[N] := { false, ..., false }
        is_held[N] := { false, ..., false }
5       for i := 1 .. N
            if exp[i] ≠ swap[i]
7               should_hold[i] := true
        do
9           todo := update
            release_from := 1
11          diatomically
                for i := 1 .. N
13                  if ¬is_held[i]
                        if mutex(objs[i]) = held // Lock is either held or invalid
15                          if todo = update
                                should_hold[i] := true
17                              todo := wait
                                release_from := i
19                      else if *objs[i] ≠ exp[i] // Object is either invalid or doesn't match expected
                            todo := abort
21                          release_from := 1
                            break
23              if todo = update // All locks are valid and available; take next one in ordering
                    for i := 1 .. N
25                      if ¬is_held[i] ∧ should_hold[i]
                            mutex(objs[i]) := held // Diatomic swap: obtain exclusive mode on object
27                          is_held[j] := true
                            todo := retry
29                          break
            if todo ≠ retry
31              for j := release_from .. N
                    if is_held[j]
33                      if todo = update
                            *objs[i] := swap[i]
35                      mutex(objs[j]) := free
                        is_held[j] := false
37          if todo = wait // At least one mutex must be both valid and held
                ExponentialBackoff() // Wait exponentially-increasing periods
39      while todo ≠ update ∧ todo ≠ abort
        return todo = update
```

Figure 6.19: Implementing a blocking, scalable multi-object compare-and-swap primitive using diatomic operations.

for trees, when a node is swung off the tree — there is no single update which can be identified as a linearization point. Instead, the linearization point of the primitive snapshot which *confirms the operation as successful* is used.

In this case, that means the snapshot executed in lines 11–29 after which `todo` is set to `update`. This is the only instant where we can state with certainty that (a) the structure is in the right state to perform the operation, and (b) conflicting operations will not occur until after the update has logically taken place. (Mutual exclusion ensures the second condition.)

6.5.2 Progress

I now show that diatomic operations can implement scalable designs with a *lock-free* progress guarantee. The key difficulty is permitting concurrent threads to safely assist obstructing operations

Theorem 6.5.2 *Diatomic snapshot-modify-update operations admit a scalable, lock-free and parallelism-preserving implementation of (object-based) software transactional memory.*

I prove this theorem constructively, by presenting such an implementation. The algorithm is not intended for practical use.

The first step, as with previous work on non-blocking software transactional memory in Section 3.8.2, is for each operation to supply a *descriptor*, allowing obstructed threads to assist them to completion instead of blocking. Deciding how to encode, and when to build, a descriptor is a key factor in optimizing a STM, but this has been adequately considered in previous work, and is not relevant to this theoretical result. For simplicity, I assume a multi-object–compare-and-swap descriptor has already been built up, as encoded in the `Transaction` class of Figure 6.20. The array of objects is assumed to be sorted by dependency: if the first j objects match their expected values, the $(j + 1)$th object must be a live object.

```
1   class Transaction {
        enum { installing, validating, committed, succeeded, failed } status
3       int N
        Object* objects[N]
5       Object expected[N]
        Object swap[N]
7       bool is_held[N] := { false, ..., false }

            // Attempt to commit the transaction
9       bool commit()
    }
```

Figure 6.20: A partial description of the `Transaction` class, containing a transaction encoded as a multi-object–compare-and-swap descriptor.

Before committing, a transaction must add its descriptor to a *control* field in each object it will be updating; after all objects have been updated, the descriptor will be removed. Unlike existing STMs, storing a single descriptor in the control field at any one time is not sufficient: since the control field may be reused arbitrarily once the object is not controlled, it is not safe for concurrent threads to add or remove an obstructing descriptor from an object. Instead, the control field stores a set of active descriptors, implementing a form of reference counting. Rather than detail this code, I simply note that the scalable, lock-free linked list algorithm of Section 6.3 can be adapted to provide the partial object interface of Figure 6.21.

```
    class Object {
11  public:
        // Add a transaction to the control field
13      // Execute within a diatomically construct
    void addTransaction(Transaction* t)
15      // Return whether any transactions are in the control field
        // Execute within a diatomically construct
17  bool controlled()
        // Returns whether a particular transaction is in the control field
19      // Execute within a diatomically construct
    bool containsTransaction(Transaction* t)
21      // Return an enumerator for iterating through transactions
        // Execute within a diatomically construct
23  TransactionEnumerator enumerateTransactions()
        // Remove a transaction from the control field
25      // Release the object's memory if necessary
        // DO NOT execute within a diatomically construct
27  void removeTransaction(Transaction* t)

        // Update some word of the object to match the swap object
29  void partialUpdate(Object swap)
    }
31  class TransactionEnumerator {
    public:
33      Transaction* next()
    }
```

Figure 6.21: A partial description of the `Object` class, showing the interface to its control field.

To decide which transaction will succeed in the event of contention, I reuse the whack-a-mole consensus algorithm of Section 5.3. Before committing, each transaction "pokes its nose up" into `validating` state, "whacks" obstructing transactions into `failed` state, and "fully emerges" into `committed` state. To ensure lock-freedom, in the event of obstruction the transaction with the higher address will assist the one with the lower, moving itself into `failed` state. Note that other contention-management schemes could be adopted in a practical algorithm.

The full code for the commit algorithm, split into several methods, can be found in Figures 6.22 and 6.23. The `commit` function installs the transaction in the control field of all necessary objects, assisting any `validating` and `committed` operations it encounters, and moving the transaction to `failed` state if any object fails to match its expected value. Once installed, it moves the transaction to `validating` state, at which point it can be concurrently assisted.

The `assist` function verifies a concurrent operation, found in `validating` state, is installed in all contended locations (read-only locations may have been controlled since the transaction entered `validating` state), and assists any `committed` operations it encounters, but does not attempt to install the descriptor; as mentioned above, this cannot be safely assisted. Instead, it rolls the descriptor back to `installing` state if it is not correctly installed in all control fields.

Both `commit` and `assist` now call the `validate` function. This validates that each object matches its expected value, and performs the 'whacking' part of the

```
35  bool Transaction::commit():
        while status = installing ∨ status = validating
37          diatomically
                if status = installing
39                  install()
                else if status = validating
41                  validate()
        while status = committed
43          diatomically
                if status = committed
45                  complete()
        for i := 1 .. N
47          if is_held[i]
                objects[i]→removeTransaction(this)
49      return status = succeeded

    void Transaction::install():
51      for i := 1 .. N
            // Assist validating and committed obstructions
53          e := objects[i]→enumerateTransactions()
            while (trans := e.next()) ≠ NULL
55              if trans→status = validating
                    trans→validate()
57                  return
                else if trans→status = committed
59                  trans→complete()
                    return
61          // Check whether the object matches its expected value
            if *objects[i] ≠ expected[i]
63              status := failed // Diatomic swap: fail transaction
                return
65          // Control all necessary objects
            if ¬is_held[i] ∧ (expected[i] ≠ swap[i] ∨ objects[i]→controlled())
67              objects[i]→addTransaction(this) // Diatomic swap: control object
                is_held[i] := true
69              return
        status := validating
```

Figure 6.22: The transaction commit method. Building the descriptor and retrying on failure are left as exercises for the reader.

whack-a-mole algorithm, assisting any concurrent `validating` operations with a lower descriptor address, and moving those with a higher descriptor address to `failed` state. If all locations match their expected value, and no obstructions remain, the operation linearizes and the transaction is moved to `committed` state.

Once a transaction has been committed, the `complete` function checks each controlled object in turn, and writes the new `swap` values over them, one word at a time. Once all new values have been swapped in, the transaction is moved to `succeeded` state.

The final step of the `commit` function is to remove the transaction from all control fields; again, this cannot be concurrently assisted. It then returns whether the transaction succeeded in committing its described changes, determined by the final status of the descriptor.

The linearization point of this algorithm is, once again, the linearization point of the primitive snapshot which *confirms the operation as successful*. In this case, that means the snapshot of whichever diatomic operation successfully moves the

```
71  Transaction::validate():
      for i := 1 .. N
73      // Assist committed transactions and perform contention management
        e := objects[i]→enumerateTransactions()
75      while (trans := e.next()) ≠ NULL
          if trans→status = committed
77          trans→complete()
            return
79        if trans ≠ this ∧ trans→status = validating
            if trans < this
81            status := installing // Diatomic swap: give way to obstruction
            else
83            trans→status := installing // Diatomic swap: block obstruction
            return
85      // Check whether the object matches its expected value
        if *objects[i] ≠ expected[i]
87        status := failed // Diatomic swap: fail transaction
          return
89      // Check all necessary objects are controlled
        if objects[i]→controlled() ∧ ¬objects[i]→containsTransaction(this)
91        status := installing // Diatomic swap: roll back transaction a step
          return
93    status := committed // Diatomic swap: commit transaction to completing

    Transaction::complete():
95    for i := 1 .. N
        if expected[i] ≠ swap[i] ∧ *objects[i] ≠ swap[i]
97        objects[i]→partialUpdate(swap[i]) // Diatomic swap: update object
          return
99    status := succeeded // Diatomic swap: no more assistable work remaining
```

Figure 6.23: Helper functions for the transaction commit method.

descriptor to `committed` state.

Chapter 7

Implementing Diatomic Operations

In Chapter 6, I introduced diatomic operations, and showed that they allow many practical and scalable implementations of shared objects, as well as proving strong theoretical properties. I now turn to the practicalities of implementing diatomic operations. I first present an instruction set extension for supporting snapshot isolation, before presenting several approaches to providing these instructions in hardware. This work leads on to a continuation of the observations of Section 4.4. Finally, I conclude the chapter with a quantitative examination of the performance of one of the implementations introduced.

Note that throughout this chapter, I use 'word' to refer to the largest unit of memory that can be read atomically with a single instruction. Previous chapters used the term 'register' here, in line with previous work in concurrent algorithms; however, in hardware the word 'register' traditionally refers to a unit of memory local to a single processor or core, so to avoid confusion, I use the unambiguous 'word'.

7.1 Instruction Set Extension

I now introduce an ISA extension to support diatomic operations. This is not the only possible extension, nor does it necessarily provide the best feature set. However, it does help frame the subsequent chapters, which discuss how to implement such a minimal extension, and which in turn suggest further features that could or should be provided by a real implementation.

My proposed ISA extension consists of two operation pairs: *snapshot-start* and *snapshot-verify*; *load-linked* and *store-conditional*.

The former of these are used to wrap a set of reads forming a snapshot. The snapshot-verify instruction should return a boolean value indicating the success of the snapshot: a return value of true guarantees atomicity, but failure may be

indicated spuriously. This design provides several benefits over a single, complex 'snapshot' operation taking a sequence of addresses:

- Standard memory subsystems only allow single memory reads, so the hardware would need to break the snapshot operation up into individual reads.
- A single operation reading multiple locations needs multiple ports to the register file (to read the locations, and later to write back the results) and multiple passes through any read phase; pipelining would thus be greatly complicated, and potentially slow the execution of other operations.
- A snapshot operation could also cause many exceptions during its execution.
- Reduced instruction-sets typically restrict the number of arguments that a single instruction can take, ruling out a single snapshot operation.
- Instruction sets often provide many forms of the read primitive to match different situations; providing special 'snapshot' versions of all of these would require adding many instructions.
- By breaking up the snapshot into individual operations, the programmer can decide on the read set dynamically; to use a monolithic snapshot operation, the algorithm would need to read all the locations then reread them as part of the snapshot, a needless duplication of effort.

Since these reads are not performed atomically, merely confirmed as atomic after the fact, processes may read inconsistent or even garbage data. This could cause an infinite loop or even a segmentation fault. Code with potentially unbounded loops should be able to periodically call the snapshot verify operation. Any code which follows pointers without using garbage collection should also be able to avoid or catch hardware exceptions: exception handlers that cannot recover quietly would impose an unnecessary overhead in all cases, as each snapshot would need to set up complex failure recovery information, for example using a C `setjump` call.

The second part of implementing a diatomic operation is providing the coupled read-modify-update operation. Here, I note that weak LL/SC fits the bill: a read operation coupled with a subsequent update of a modified value that succeeds only if the location has not been concurrently modified. If the load-linked forms part of the snapshot, the subsequent update will succeed only if the snapshot was atomic and the updated location is not modified between the linearization point of the snapshot and the update — exactly the semantics required for a diatomic operation.

Platforms that implement CAS rather than LL/SC may find a fused LL–snapshot verify–SC, or *isolated store*, instruction more convenient to implement.

Processors implementing LL/SC with cacheline locks, where the processor refuses to release exclusive mode until the store is completed, typically require a timeout period to ensure context switches and malformed programs do not cause deadlocks. An ISA containing only fused instructions need not introduce this complexity, even if they use LL/SC microcode internally.

I will now discuss how to implement the snapshot instructions. (LL/SC is well-known in the literature, e.g. [82], and the novel part of an isolated store instruction is how to manage the snapshot verify.)

7.2 Hardware Designs

I present three implementations of the snapshot-begin/snapshot-verify instructions. The *pragmatic* implementation requires the least investment, and additionally can be emulated on some existing hardware. The *snapshot set* implementation describes microarchitecture extensions to avoid the limitations of the pragmatic approach. Finally, the *timestamp* implementation describes major microarchitectural changes that achieve stronger theoretical properties.

7.2.1 Pragmatic Implementation

An observation: if a sequence of reads all hit in a cache, they must all have been present at the start of the sequence (Figure 7.1), provided words are only loaded into the cache on a miss.

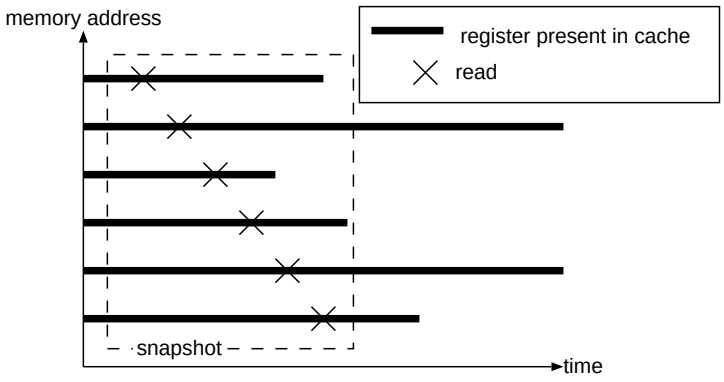


Figure 7.1: If a sequence of reads hits in the cache, they must all have been present at the start of the sequence, assuming data is fetched only on demand.

The *pragmatic implementation* of multi-word atomicity keeps track of the number of cache misses during a snapshot, confirming atomicity only if no misses were detected; otherwise the snapshot must be retried. Given a set of locations that fits into the cache, this approach is lock-free: each time the snapshot is

repeated, the words will be reloaded, so even with a random replacement policy some snapshot must succeed, unless a concurrent modification evicts one of the locations.

At the hardware level, this can be done with a bit field, cleared at the start of a snapshot, set on a cache miss, and verified on a snapshot verify. Context switches must also be tracked, since a preempting thread may change some of the read locations without causing a cache miss; the bit field should also be set every time there is a context switch.

An alternative approach leverages existing cache miss counters, if they are provided; the current cache miss count is checked before and after the snapshot, triggering a retry if the values do not match. Accurate cache miss counters are increasingly available as programmers demand greater ability to tune programs and locate problem spots: the PowerPC architecture has recently added accurate cache miss counters to its ISA. Since the PowerPC has had weak LL/SC since its inception, I have therefore been able to implement diatomic operations on a testbed, and evaluate their performance: see Section 7.5.

The pragmatic approach has several drawbacks, all of which can be derived from the original observation. First, a snapshot can only be taken if the sequence of reads involve *can* all hit in the cache. Large snapshots that overflow the cache capacity (capacity miss) will never succeed. Equally, caches cannot store any arbitrary set of words: a cache is typically divided into many small sets, and each word can only be stored in a particular set. The size of these sets is called the *associativity* of the cache. If a given snapshot contains more words that map to a single set than the sets can store (conflict miss), again, the snapshot will never succeed.

A direct-mapped cache, for instance, has an associativity of one, and a badly-placed snapshot covering just two words might never succeed. This complicates algorithm design, requiring slow exceptional code with very pessimistic assumptions about the level of read-parallelism which can be exploited. Some designs might even be impossible to adapt for caches with conflict misses in small snapshots.

Associativity influences the latency of a cache: typically, the higher the associativity, the higher the latency. Since the majority of instructions benefit from lower latency more than decreased conflict misses, an infrequently-used instruction like an atomic snapshot would not provide sufficient overall benefits to merit increased associativity. However, conflict misses can be eliminated in small snapshots by adding a *victim cache* [45], a buffer of cachelines evicted due to conflict misses. This will ensure a minimum number of memory locations that can be successfully read in a snapshot, and decrease the probability of conflict misses in larger ones.

Secondly, a snapshot must spin unless *all* reads hit in the cache. In a concurrent benchmark, or when a thread has a large active footprint, memory will rarely be cached before a snapshot starts, and diatomic operations will have to

spin at least once before they can succeed, even though failure due to conflicting updates may be rare (Figure 7.2). This overhead is intrinsic to the pragmatic approach, and may be visible in benchmark results.

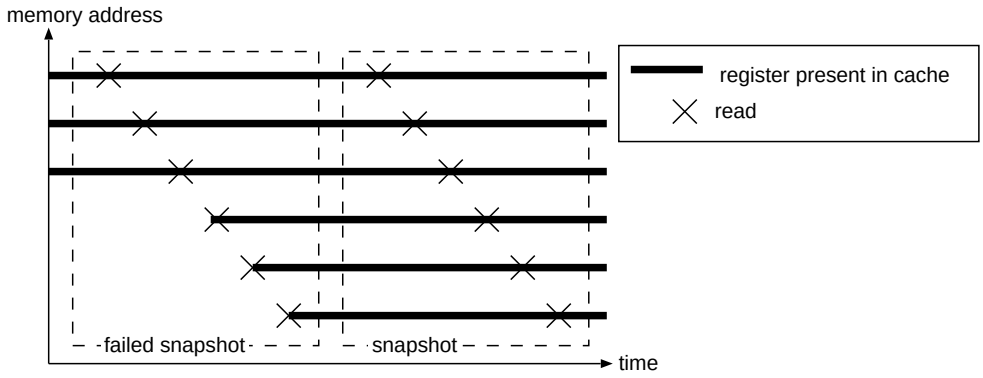


Figure 7.2: Capacity misses due to a large working set, such as a large shared tree, will cause a pragmatic implementation of atomic snapshots to retry even in the absence of conflicting updates.

The other problems with a pragmatic implementation arise from the assumption that words are only loaded into the cache on a miss. Hardware prefetching will silently break this assumption, as data that is in the cache may have been prefetched after the operation started. Disabling prefetching for the duration of a snapshot would likely be extremely costly both to implement and to execute, diminishing the benefit of using diatomic operations.

Finally, multi-core designs often use shared caches for latency and scalability reasons. If all caches are shared, the scheme cannot be used at all; and even if some caches are unshared, they will generally be low-latency designs, with the associated problems just mentioned. Splitting the cache into equal sections, one for each core, would allow pragmatic snapshots, but would likely result in worse performance, again diminishing the benefit of using diatomic operations.

The potential scalability benefits of using diatomic operations may outweigh the costs of disabling prefetching and splitting caches for massively parallel applications. However, a more compelling hardware implementation would be highly desirable.

7.2.2 Snapshot Set Implementation

I now introduce an alternative implementation of diatomicity, addressing some of the short-comings of the pragmatic implementation. This requires more resources on a chip, but avoids negative interactions with conflict misses, pre-fetching and shared caches, and improves performance in many situations.

The basic method is to store the set of locations read so far during a snapshot, the *snapshot set*, and snoop the bus for updates to those locations. Snapshots now linearize to the moment the hardware confirms no updates have been observed.

For small snapshots, it would be easy to provide a fixed-size snapshot set, fully-associative to prevent false negatives due to conflict misses. However, fixed-size sets are quite restrictive, always failing when snapshots grow too large, and hence forcing the user to know precise hardware details when designing algorithms.

A Bloom filter [14] is a probabilistic data structure for storing sets that removes the hard bound on set size in exchange for false positives when checking for set membership. An empty Bloom filter is a k-bit array, with all bits set to 0; each element in the key space hashes to m bits in the array (k and m are chosen to balance space requirements and false positive rates for various set sizes). To insert an element, the corresponding m bits are set to 1. To check if an element is in the set, the corresponding m bits are read, and if any are 0, the element is definitely not in the set. Using a Bloom filter when the fixed-size set overflows allows larger snapshots to execute safely, but with a risk of false conflicts and retries. See Figure 7.3.

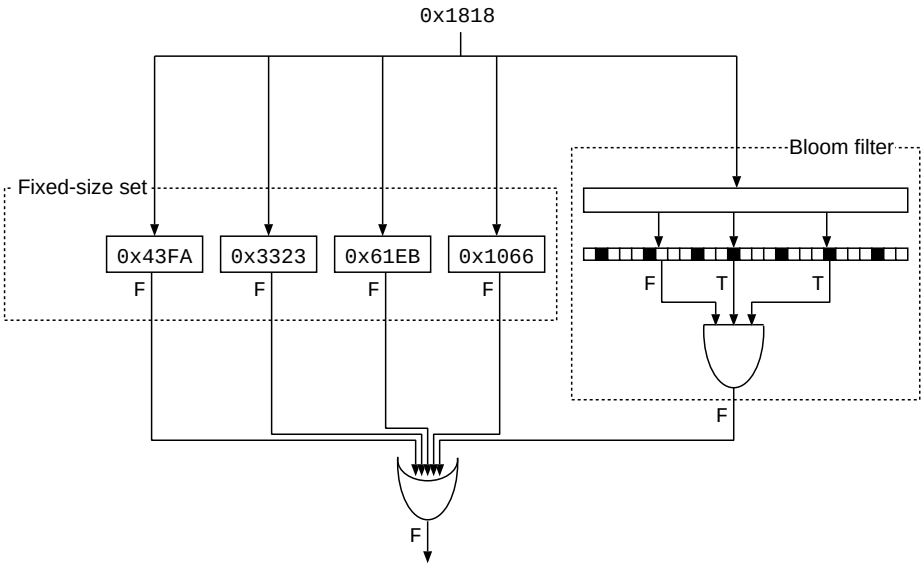


Figure 7.3: An update to location 0x1818 is detected and checked in parallel against the snapshot set. The location is not found in the fixed-size set, nor does it match the Bloom filter.

A Bloom filter can be very effective in allowing modest hardware to take large snapshots. Indeed, it may be practical to drop the fixed-size set and dedicate the space to the Bloom filter. The following compares three possible implementations of a snapshot set: a sixteen-entry fixed-size set; an eight-entry fixed-size set and a 64-byte Bloom filter; and a single 128-byte Bloom filter. The smaller Bloom

filter uses 12 bits per element (optimal for storing 32 elements) and the latter 18 (optimal for 40 elements). Assuming a 64-bit architecture, these implementations all require 128 bytes of storage.

Elements	Fixed-size set	Set and Bloom filter	Just Bloom filter
8	No false positives	No false positives	c. 1 in 2^{53}
16	No false positives	c. 1 in 1.5 billion	c. 1 in 100 billion
40	Always fails	c. 1 in 2,000	c. 1 in 200,000

As the table shows, for smaller sets, the Bloom filter is highly unlikely to fail; indeed, on a computer with less than a petabyte of memory, a sufficiently well-chosen hashing function would guarantee no false conflicts for almost all small snapshot sets. The choice of how to allocate resources is therefore likely to be made based on the complexity of implementing a strong hashing function.

If the hardware uses a write-like LL/SC (see Section 4.4), as all existing implementations do, combining it with this implementation of a snapshot will not be lock-free: two concurrent diatomic operations taking the same snapshot but updating separate locations can both succeed in negotiating exclusive mode for their respective cachelines, and subsequently both fail their subsequent snapshot check.

(Note that the pragmatic implementation of atomic snapshots does not suffer this problem, as cache misses are caused by reads, not concurrent updates; a write-like LL/SC is thus sufficient for implementing lock-free diatomic snapshots.)

A small extension to the design is thus required. When a concurrent update to a member of the snapshot set is detected, the location is added to a *change set*, again implemented using a Bloom filter (Figure 7.4). The isolated store can now check whether the location being modified has been updated without failing due to other updates.

A snapshot set implementation should provide better performance than a pragmatic implementation. As cache misses no longer cause the snapshot to be repeated, if a thread's memory footprint cannot fit into cache, or if coherence misses are common, the number of reads required to make a successful snapshot will be halved in the common case.

Perhaps most importantly, this approach frees the programmer from worrying about cache limitations like conflict misses preventing a snapshot from succeeding. Even large snapshots will have a chance to succeed, depending on the properties of the Bloom filter. Further, assuming reasonable constraints on the scheduler, system-wide throughput is guaranteed, as a snapshot failure will always be directly attributable either to a context switch or to a concurrent update (though this by no means guarantees high throughput, fairness or scalability).

One remaining question is how to allocate available storage between the snapshot and change sets. Allotting more bits to the snapshot set greatly decreases the number of false positives; doubling the number of bits reduces the typical probability by two orders of magnitude. Since the change set size will thus be

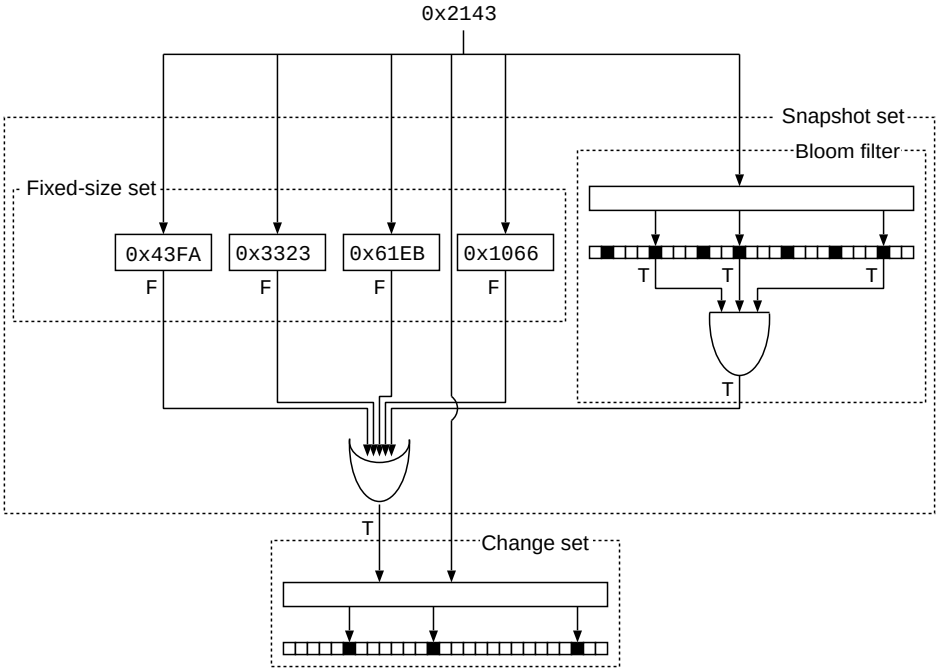


Figure 7.4: An update to location 0x2143 matches against the snapshot set, and is stored in the change set for later comparison.

orders of magnitude smaller, a small filter size will be sufficient; for instance, if the typical change set holds at most two elements, a single 8-byte filter with 20 bits set per element yields a false positive probability of around one in five million.

Thus, a large snapshot set filter and small change set filter seems to represent the best allocation of resources. The write-like LL/SC, representing the most costly part of a snapshot in the common case, requiring as it does negotiation for exclusive mode on a word, is also the most likely point in time for a snapshot to fail; the small change set may greatly improve throughput in the face of contention. Further, an algorithm that stores which words were read in a snapshot could potentially use this highly-accurate change set to greatly decrease false snapshot failures by verifying each location in turn against the change set. Even if the failure was due to a concurrent modification, the ability to pinpoint which word is experiencing contention may help improve contention management.

7.2.3 Timestamp Implementation

My final approach to implementing diatomicity adds a *modification timestamp* to each cacheline, stored in main memory as well as in the caches themselves. Every processor in a cluster has a globally-synchronized clock; whenever a cacheline is

modified, the timestamp is updated to the current value of the clock.

A snapshot-modify-update begins by taking a copy of the current clock value, called the *linearization time*. Each memory access operation compares the modification timestamp of its cacheline with the linearization time. A set of reads is atomic if all modification timestamps precede the linearization time. The final update can then be a write performed conditionally on the modification time preceding the linearization time.

This implementation has the advantage of being *strong*: it will only fail if one of the locations is modified between the start of the diatomic operation and one of the memory accesses. It can also be used to implement strong LL/SC. Large atomic snapshots will only fail due to concurrent modifications, not due to capacity or conflict misses caused by hardware constraints, greatly simplifying the task of the programmer. Like the snapshot set implementation, there will be no false retries.

Since a fixed-size counter can overflow, the hardware would need to periodically sweep through memory, replacing all sufficiently old counters with a reserved value, `old`, considered older than all current transactions. Extremely long transactions would need to be aborted to avoid the risk of running out of live timestamps; in any reasonable usage, such a long transaction would only occur due to a system failure.

Unfortunately, significant changes to the architecture would be needed to support modification timestamps. Every cacheline would need an entire word reserved for the timestamp. If this were stored in main memory, either cacheline sizes would need to be increased, preventing the use of off-the-shelf memory chips, or one of the standard words would need to be reserved, and the cacheline size would need to be halved to keep the arithmetic for computing cacheline location from memory address feasible. Alternatively, the modification timestamp for a cacheline fetched from main memory could be pessimistically estimated, perhaps by storing and using only the latest update timestamp; this would be a correct implementation, but would no longer be strong. A final option would be to store modification timestamps in reserved cachelines, fetching cacheline pairs in bursts; this could easily double required memory bandwidth and footprint.

Finally, even if a single word were reserved per cacheline, cache size and bandwidth demands would increase by 12% on a typical system. This overhead will be reduced on systems with many words per cacheline, but false sharing may then start to affect performance.

7.3 Combining Operations

One simple but effective optimization possible with all three implementations is to combine several sequential diatomic operations with overlapping footprints into one larger multiatomic operation. Rather than reread each location in the

snapshot, only new locations are read in; the linearization point of the larger snapshot is then allowed to occur before the linearization point of the first update. In the pragmatic implementation, for example, the second snapshot-verify will succeed only if the cache miss counter has not been modified since the start of the *first* snapshot.

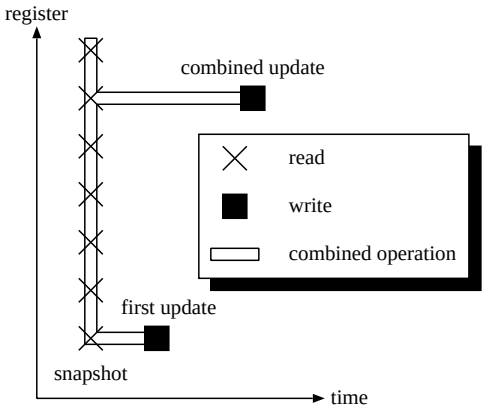


Figure 7.5: A multiatomic operation created by combining two sequential diatomic operations. The second snapshot is combined with the first, saving the thread from having to read every word twice. However, the second update may fail after the first has succeeded; the algorithm must be robust against such partial updates.

Operations cannot be combined arbitrarily: as Figure 7.5 shows, the result is essentially a single snapshot followed by multiple updates. If a snapshot must *follow* an update, as is the case at the linearization point of both universal constructions in Section 6.5, the snapshot *must* be redone from the start.

Combined operations provide a large performance improvement provided they do not frequently fail between the first and last updates; for the pragmatic implementation, this can be ensured in the common case by reading all affected memory locations in the first snapshot, performing only updates afterwards.

The snapshot set implementation allows for a slightly stronger optimization: each check of the snapshot set counts as an atomic snapshot of the words in the set. This means that, unlike the pragmatic implementation, *all* diatomic operations with overlapping read sets can be combined. As the pattern of snapshot-update-snapshot-update is required to linearize a transaction in the general case, this performance improvement should be significant.

The timestamp implementation is the least suited to providing this optimization: care would need to be taken to allow updates on the same cacheline to be combined, as the first update would modify the timestamp.

Figure 7.6 illustrates this optimization, applying it to the linked-list erasure function introduced in Figure 6.11. I use a `multiatomically/failure` construct,

similar to the `try/catch` constructs found in many languages: after the initial `diatomically` block, any subsequent diatomic operations that can be combined with the first are wrapped in `multiatomically` blocks, with cleanup code in optional `failure` blocks.

```
1   bool LinkedList::erase(Key key):
        nodeIsDeleted := false
3       while ¬nodeIsDeleted
            diatomically
5               switch find(key, &prev, &cur, &next)
                    case absent:
7                       return false
                    case present:
9                       cur→⟨mark, next⟩ := ⟨true, next⟩ // Diatomic update: mark node as deleted
                        nodeIsDeleted := true
11                      break;
                    case retry:
13                      break;
        multiatomically
15          *prev := next // Combined diatomic update: swap out node
            return true
17      failure
            while true
19              diatomically // Ensure node is removed
                    if find(key, &prev, &cur, &next) ≠ retry
21                      return true
```

Figure 7.6: Combining two diatomic operations on the fast path of Figure 6.11.

The fast path of erasure when the key is present consists of marking the node as deleted, then swinging the next pointer of the previous node past the deleted node. These two snapshot-modify-updates can be combined, saving the cost of a snapshot in the optimal case.

Since applying this optimization only adds to the length of code, as the slow path must still be present in case the multiatomic blocks fail, I did not introduce it earlier; nor will I add lengthy duplicates of earlier pseudocode to illustrate its use. The optimization has nevertheless been used whenever applicable in empirical evaluations (Section 7.5).

7.4 Nestable Read-Like LL/SC Synergies

In Section 4.4, I introduced nestable, read-like load-linked/store-conditional operations, and noted that Lemma 4.2.2 did not apply to them. As no hardware implementation of nestable LL/SC has yet been provided on any major architecture, I did not pursue the observation further in that section. As it turns out, however, like diatomic operations, nestable read-like LL/SC can scalably implement transactional memory. I now discuss the synergies and differences between the two primitives.

Diatomic operations have the following scalable, lock-free implementation from nestable, read-like LL/SC: load-link the words involved; ensure an atomic

snapshot has been taken by doing a non-modifying store-conditional on all locations except the one being updated; and finally update the remaining location with a store-conditional. Hence there is also a scalable implementation of transactional memory from the latter.

Further, the snapshot set implementation of diatomic operations can be coopted to implement weak-but-nestable, read-like LL/SC: each LL operation adds its word to the snapshot set, and each non-updating SC succeeds only if the location does not match the change set. When all LL/SC pairs have finished, the snapshot set is emptied. An updating SC can then be implemented by simply fusing a write-like LL/SC operation with a non-updating SC.

Diatomic operations provide a performance advantage over nestable read-like LL/SC: when taking a snapshot of memory, the number of memory-touching operations required is almost halved compared with LL/SC. Further, a large snapshot that caused conflict misses in the cache would trip each miss again when performing the subsequent SC.

Providing diatomic operations also simplifies dynamic snapshot algorithms: by offloading the task to the snapshot set in hardware, the algorithm is not required to remember which locations were linked. This also saves the time that would be needed to build a local stack of locations.

One interesting approach would be to provide sufficient primitives to implement both diatomic primitives and nestable read-like LL/SC. Diatomic operations and nestable read-like LL/SC can thus be seen as complimentary, requiring similar hardware in their implementations.

7.5 Evaluation

As mentioned earlier, the PowerPC platform provides weak LL/SC and low-latency cache-miss counters, allowing a direct hardware implementation of diatomicity, following the pragmatic design. While more recent PPC platforms have strong hardware prefetching, invalidating the assumptions that underlie the correctness of the pragmatic design, the Motorola G4 does not. To conclude this chapter, I evaluate the performance of diatomic operations on this platform.

7.5.1 Results

The test machine had two 1.25 GHz Motolora MPC7455 (G4) processors, each with a dedicated 8-way set-associative, 256K L2 cache. This high level of associativity compensates for one of the chief disadvantages of the pragmatic design, as small snapshots are almost guaranteed to fit in the cache. Unfortunately, the low level of concurrency in the hardware prevents the results from supporting (or refuting) the theoretical guarantees of scalability of the diatomic-based designs.

I evaluated three alternative designs for a concurrent, unbalanced binary tree. DB is a scalable blocking design built from diatomic operations using the universal approach presented in Section 6.5. DLF is the scalable lock-free design presented in Section 6.4, also built from diatomic operations. Both use a custom memory allocator, maintaining a small per-thread free-list for performance, and a common overflow list, necessary to preserve garbage-freedom.

Finally, CB is a best-of-breed blocking, CAS-based design due to Fraser[26], freely available under the GNU General Public License. As with other modern designs, this takes the form of a parallelism-preserving, population-oblivious algorithm coupled to a garbage collector. I chose an epoch-based collector scheme, as adopting Safe Memory Reclamation is highly non-trivial, and earlier results suggest that performance will be degraded.

Both blocking designs use simple spinlocks with exponential backoff. While some effort was expended selecting a good backoff protocol, an extensive parameter search was not performed. For CB, MCS locks were also trialled, but performance was degraded, and for clarity the results are not shown.

This section is not intended to be a rigorous evaluation, as the available hardware is not truly representative of a production-quality machine. Instead, it will evaluate the *viability* of diatomic operations as a hardware primitive. As such, the omissions in fine-tuning the algorithms chosen will not affect the validity of the conclusions drawn.

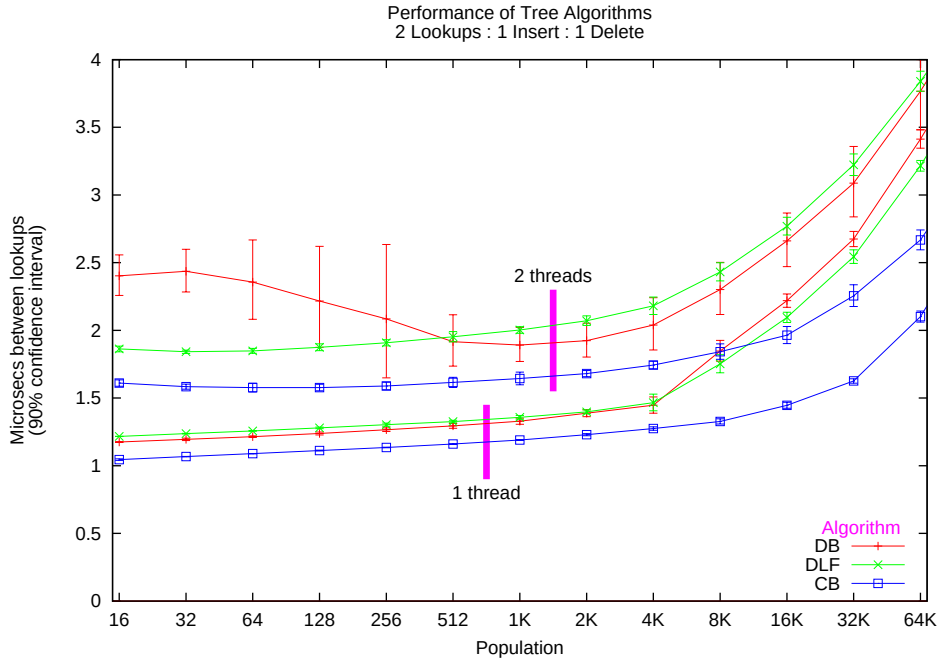


Figure 7.7: Performance of the competing tree algorithms, for smaller numbers of keys, on a 2-way PowerPC machine, with one and two threads; lower is better.

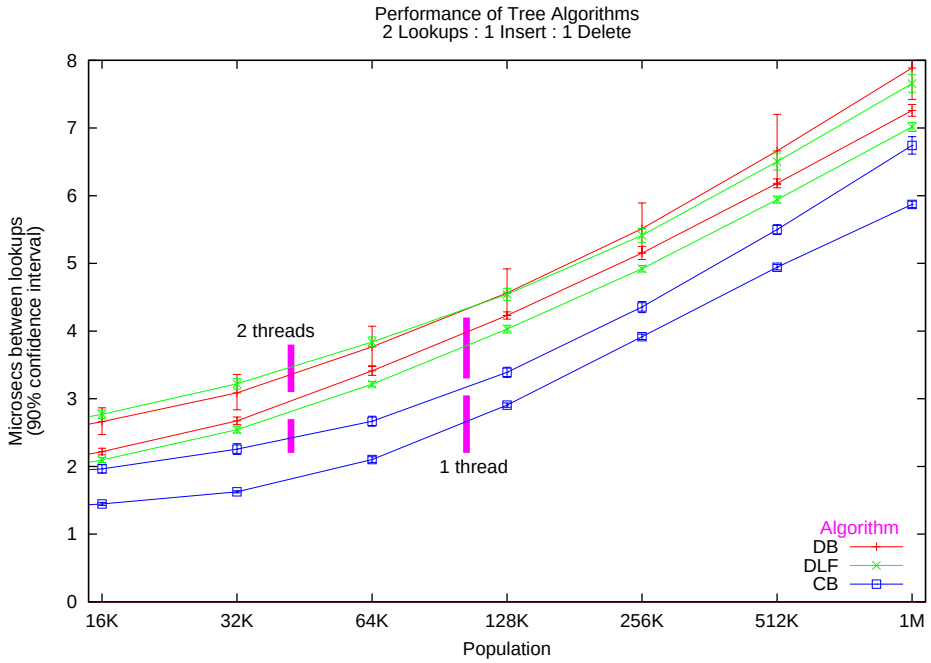


Figure 7.8: Performance of the competing tree algorithms, for larger numbers of keys, on a 2-way PowerPC machine, with one and two threads; lower is better.

As the results in Figures 7.7 and 7.8 show, the diatomic-based designs perform within a factor of two of the best-of-breed CAS-based algorithm at all times: there is no unanticipated penalty associated with using diatomic operations. Primarily, however, the results show the limitations of the test setup.

DB suffers performance penalties with two threads under high contention (small number of keys), yet this represents a significant improvement over busy-spinning (no backoff, not shown) even with a limited investment in optimizing the backoff strategy, and it is likely that further improvements could be obtained.

Earlier versions of DB had a 50% performance penalty over DLF in many cases; this was found to be due to the use of frequent isolation checks to prevent invalid-memory-access exceptions, which cannot be usefully caught in the system under test. Fortunately the memory allocator used does not free memory for arbitrary reuse, allowing these checks to be removed; in general, this would not be possible. This highlights the importance of allowing code to recover from such exceptions, a feature not required by traditional multi-threaded algorithms.

Each algorithm was run with one, two, three and four threads, but as the test machine was a two-way, results for three and four threads mainly show the overhead of blocking algorithms under such circumstances (DLF performed identically to the two-threaded case), and are not shown to keep the graphs comprehensible.

7.5.2 Avoidable Overhead

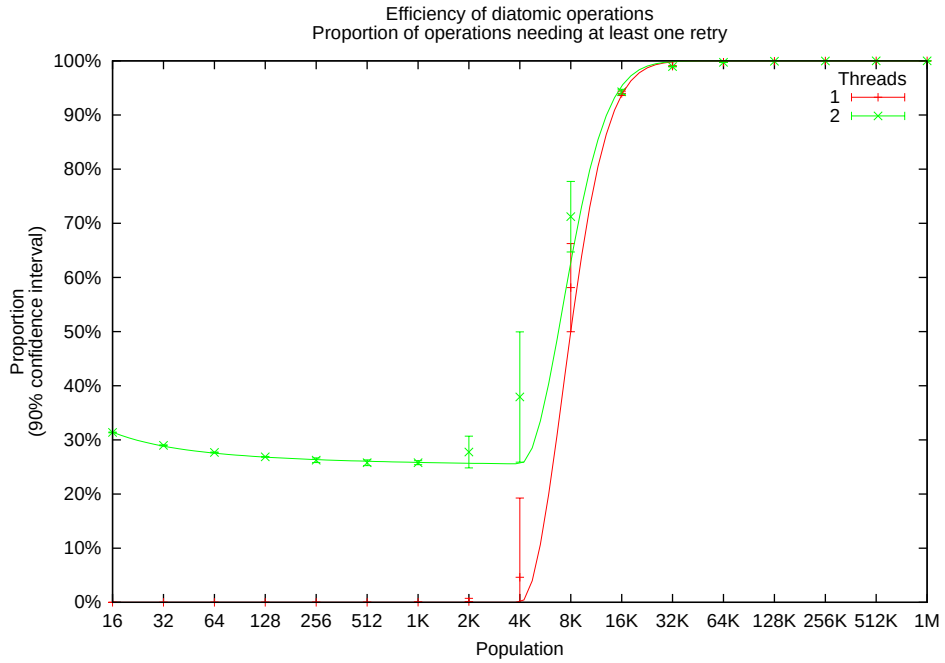


Figure 7.9: Overhead of pragmatic implementation of diatomicity, showing the proportion of operations requiring at least one retry as occupancy and number of threads grows; lower is better.

Both DB and DLF suffer performance penalties when run with two threads, a large memory footprint, or both. Under the pragmatic implementation of diatomicity, any snapshot which is not initially in cache must be performed twice before it can complete. If the memory footprint is large, capacity misses in the cache will be common, forcing many retries that, with a less inefficient design, could be avoided. Further, with two threads performing updates, concurrency misses in the cache will be common even when the entire active memory footprint can fit in cache.

This is quantified in Figure 7.9. The proportion of operations requiring at least one retry never drops below 25% for two threads: this is largely due to the relevant path in the tree not being in the cache in the required mode due to previous operations by the concurrent thread. Past a few thousand keys, the data structure becomes too large to fit into the cache, and initial miss rates rise rapidly; by a few tens of thousands of keys, almost all operations require a retry. These overheads are entirely avoidable.

Actual isolation failures (where work must be redone due to concurrent updates) and overrunning of scheduling quanta (where work must be redone because the thread was preempted by the kernel) are much rarer than these capacity and

concurrency misses. Figure 7.10 shows an estimation of the number of retries needed in a more efficient design, assuming that 25% of the overhead for two threads is due to avoidable concurrency misses. This is probably a conservative estimate, as most of the runtime of the benchmark for low population sizes is spent generating random numbers, so the window of opportunity for isolation failures is small.

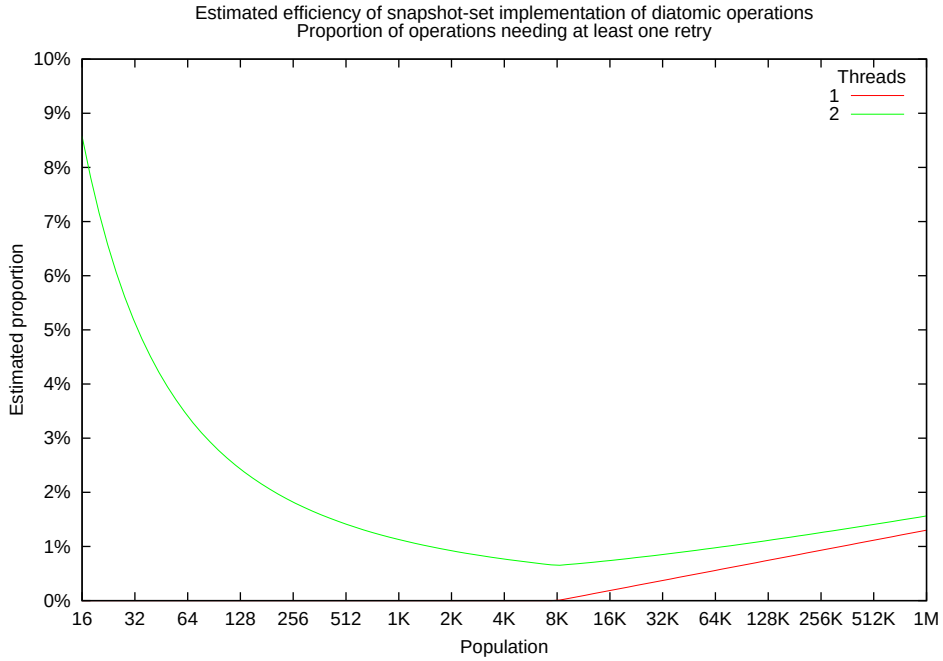


Figure 7.10: Estimated overhead of snapshot set implementation of diatomicity, showing the proportion of operations requiring at least one retry as occupancy and number of threads grows; lower is better.

Even with this conservative assumption, the estimated proportion of isolation failures drops to below 10%. Note that the rising number of retries needed as the population grows into the tens and hundreds of thousands is due to context switching during diatomic operations; these numbers have been estimated from data taken from the pragmatic implementation benchmark.

7.5.3 Memory Footprint

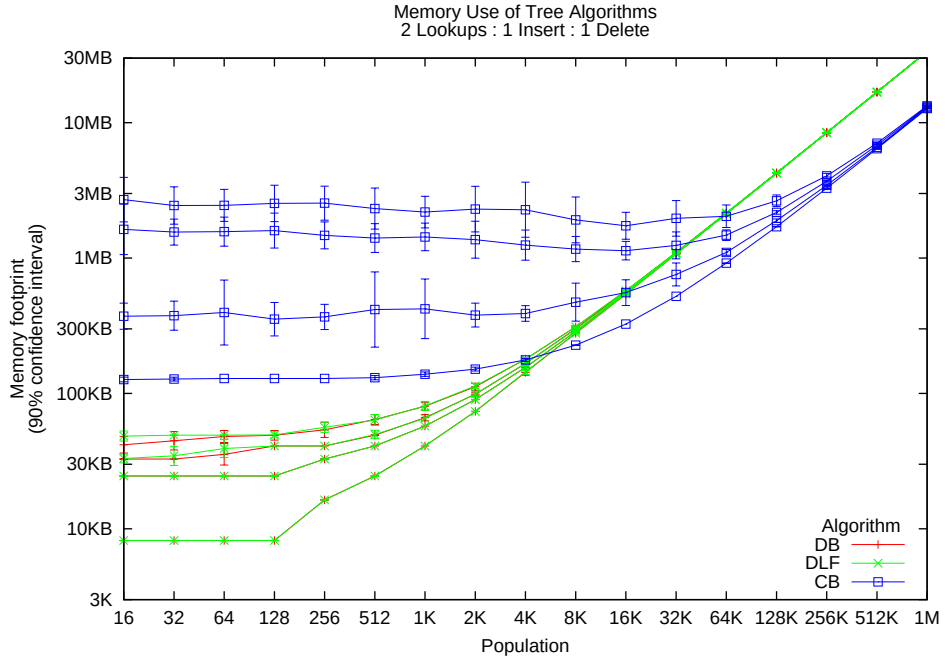


Figure 7.11: Memory use of the competing tree algorithms, with one to four threads; lower is better.

As Figure 7.11 shows, the high performance of the parallelism-preserving CAS-based algorithm comes at a cost: memory usage is always high, even when the tree itself is almost empty. As the number of threads grows, the scalable diatomic-based algorithms allocate a small per-thread memory pool, with only a small increase in the footprint.

For very large ($> 32K$ keys) trees, the overhead of using epoch-based garbage collection is small compared with the memory footprint of the tree itself; in this range, the CAS-based algorithm, which stores keys in interior nodes as well as leaf nodes, has a lower footprint than DB and DLF, which do not. DB could be modified to decrease this footprint, adapting the CB design, but this would complicate the algorithm, as well as requiring considerable time to verify the new code; this work would not have contributed to the conclusions of this chapter, and was therefore decided against.

7.5.4 Discussion

In conclusion, diatomic operations appear to be a viable hardware primitive. In a system using the snapshot set implementation (Section 7.2.2), the impact of capacity and concurrency cache misses should be avoided, giving performance

matching the best-of-breed CAS design tested. Further, the memory footprint is bounded with only a minimal investment in a fast memory allocator, while the fast epoch garbage collector has a very high overhead.

Unfortunately, due to the limited parallelism in the test machine, it has not been possible to confirm that the practical scalability of diatomic operations matches the theoretical potential.

Chapter 8

Conclusions

In this dissertation, I have defined some theoretical properties that allow the performance of an implementation to scale with the number of concurrent threads, and shown that existing hardware primitive operations are insufficient for universally constructing such scalable algorithms. In this chapter, I summarize my contributions, which stem from this result, and suggest possibilities for future research.

8.1 Summary

In Chapter 1, I gave informal definitions of the main theoretical properties considered in this dissertation, and introduced my thesis: that existing instruction set architectures are insufficient for universally constructing scalable algorithms; but that they can be suitably extended without incurring detrimental hardware costs.

In Chapter 2, I formally defined the terms used in the dissertation. The terminology related to progress and general theory has been introduced elsewhere. The four theoretical scalable properties have been used informally in earlier work; part of my contribution is setting them a strong theoretical framework to allow general theorems to be framed and proved.

In Chapter 3, I covered prior work related to the subject of my thesis. While the scalable properties have been considered individually, the implications of combining them have not previously been studied. Much theoretical work has also focused on progress guarantees, which are independent of the scalability properties.

In Chapter 4, I showed that existing single- and double-word primitives cannot implement transactional memory with all four scalability properties. This motivates recent work on obstruction-free algorithms: by ignoring garbage collection, they allow the trade-off between the four scalability properties to be determined by choosing a garbage collection algorithm. It also provides additional incen-

tive to support transactional memory in future hardware, avoiding the scalability trade-off altogether; however, this introduces other problems.

Since it is impractical to entirely abandon existing hardware, in Chapter 5, I described how to implement a lock-free, reasonably scalable set based on open-addressed hashtables using the widely-available compare-and-swap instruction. This is scalable under reasonable assumptions and restrictions, achieving good performance and scalability in benchmarks without requiring the implementer to select and fine-tune a garbage collector. However, the assumptions will restrict the algorithm's range of applicability.

In Chapter 6, I suggested that transactional memory is too complex to be reliably adopted in future instruction sets, and introduced an alternative hardware primitive, the *diatomic operation*. After presenting several algorithms built from it, I showed that it is universal for scalable, lock-free algorithms. It is thus as strong as transactional memory on a theoretical footing, and stronger than existing primitives.

In Chapter 7, I outlined three possible hardware implementations of diatomic operations with different properties and costs. All three are lock-free, allowing contention to be detected and handled in software; this provides a strong motivation for providing diatomic operations rather than transactional memory in future hardware. Further, the most pragmatic implementation can be emulated on existing hardware, allowing the design to be evaluated empirically. The results, though limited by the available hardware, suggest that diatomic operations can provide good practical performance.

In conclusion, my thesis — that existing instruction set architectures are insufficient for universally constructing scalable algorithms, but can be suitably extended without incurring detrimental hardware costs — is justified as follows. Firstly, I provided rigorous definitions of four properties of scalable algorithms in Chapter 2, and showed that they cannot all be universally satisfied with existing primitives in Chapter 4. Secondly, I evaluated several CAS-based algorithms, including one not previously introduced, in Chapter 5, showing that dropping the scalable properties does indeed cause practical problems. Finally, I introduced a new hardware primitive, with compelling theoretical (Chapter 6) and practical (Chapter 7) benefits. Future hardware adopting this primitive can provide performance, scalability and progress for concurrent algorithms.

8.2 Future Research

As future hardware provides increasing concurrency potential, scalability will continue to grow in importance. Providing algorithms that are scalable, but only under reasonable assumptions, is a promising avenue of exploration. For instance, coding a reasonably scalable hashtable would be considerably simplified if the requirement of a progress guarantee were dropped; will this simplification

translate to a faster algorithm?

Creating a theoretically strong yet simple to implement hardware primitive relied on discarding linearizability for snapshot isolation, a weaker consistency constraint. Would transactional memory also be simplified if atomicity were relaxed? And are there other suitable consistency constraints?

I have shown that diatomic operations can implement a scalable lock-free software transactional memory (STM), but my design was not intended for practical use. It remains to be seen whether existing research into STMs can be applied to produce a scalable, lock-free STM that provides usable performance.

Busy-waiting for a blocked data structure to be updated by a concurrent thread can be a performance bottleneck on heavily-contended locks, as the cost of updating shared memory grows considerably when many threads are repeatedly concurrently reading it. While software solutions to this exist, the *snapshot set* implementation of diatomicity allows a thread to wait for an update to a set of memory locations to be published on the memory interconnect without requiring software support in the publishing thread. This could allow algorithms to degrade better under contention.

Finally, I have assumed that diatomic operations will be used to implement software transactional memory and off-the-shelf optimized data structures. An alternative solution may be to provide programmers with weaker isolation, lower-level abstractions, or even direct access to the diatomic operations themselves, potentially allowing finer control and targeted optimizations.

8.3 Acknowledgements

I would like to thank my first and third year supervisors, Tim Harris and Keir Fraser, for giving my research direction, support, and endless hours of proof-reading; my parents, for getting me here; and my wife, for everything.

Bibliography

[1] IBM System/370 Extended Architecture, Principles of Operation. *IBM Publication No. SA22-7085*, 1983.

[2] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M. and Shavit, N. Atomic Snapshots of Shared Memory. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, August 1990, pp. 1–13.

[3] Afek, Y., Stupp, G. and Touitou, D. Long-Lived and Adaptive Atomic Snapshot and Immediate Snapshot (Extended Abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, July 2000, pp.71–80.

[4] Alemany, J and Felten, E. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, August 1992, pp.125–134.

[5] Agesen, A., Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Shavit, N. and Steele, G. DCAS-based Concurrent Deques. In *Theory of Computing Systems*, Volume 35, Number 3, 2002, pp. 349–386.

[6] Amdahl, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings, (30)*, 1967, pp. 483–485.

[7] Ananian, C. Scott, Asanovic, K., Kuszmaul, B., Leiserson, C. and Lie, S. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High- Performance Computer Architecture*, February 2005, pp.316–327.

[8] Anderson, J. Composite Registers. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, August 1990, pp.15–29.

[9] Anderson, J. Multi-Writer Composite Registers. In *Distributed Computing*, Volume 7, Issue 4, May 1994, pp.175–195.

[10] Anderson, J. Lamport on Mutual Exclusion: 27 Years of Planting Seeds. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, August 2001, pp.3–12.

[11] Attiya, H. and Rachman, O. Atomic Snapshots in O($nlogn$) Operations. In *SIAM Journal on Computing*, Volume 27, Issue 2, April 1998, pp.319–340.

[12] Barnes, G. A Method for Implementing Lock-Free Shared Data Structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1993, pp.261–270.

[13] Berenson, H., Bernstein, P., Gray, J., Melton, E., O'Neil, E. and O'Neil, P. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Managemeent of Data*, May 1995, pp. 1–10.

[14] Bloom, B. Space/time trade-offs in hash coding with allowable errors In *Communications of the ACM*, July 1970, Volume 13, Issue 7, pp. 422-426.

[15] Chou, Y., Spracklen, L. and Abraham, S. Store Memory-Level Parallelism Optimizations for Commercial Applications. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2005, pp. 183–196.

[16] Chung, J., Chafi, H., Minh, C., McDonald, A., Carlstrom, B., Kozyrakis, C. and Olukotun, K. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006, pp.266–277.

[17] Detlefs, D., Flood, C., Garthwaite, G., Martin, P., Shavit, P. and Steele, G. Even Better DCAS-based Concurrent Deques. In *Proceedings of the 14th International Symposium on Distributed Computing*, October 2000, pp. 59–73.

[18] Detlefs, D., Martin, P., Moir, M. and Steele, G. Lock-Free Reference Counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, August 2001, pp. 190–199.

[19] Detlefs, D., Doherty, S., Grove, L., Flood, C., Luchangco, V., Martin, P., Moir, M., Shavit, N. and Steele, G. DCAS is Not a Silver Bullet for Nonblocking Algorithm Design. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, June 2004, pp. 216–224.

[20] Do Ba, K. Wait-Free and Obstruction-Free Snapshot. Senior Honors Thesis, *Dartmouth Computer Science Technical Report TR2006-578*, June 2006.

[21] Fatourou, P., Fich, F. and Ruppert, E. Space-Optimal Multi-Writer Snapshot Objects are Slow. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, July 2002, pp.13–20.

[22] Fatourou, P., Fich, F. and Ruppert, E. A Tight Time Lower Bound for Space-Optimal Implementations of Multi-Writer Snapshots. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, June 2003, pp.259–268.

[23] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P. and Shasha, D. Making Snapshot Isolation Serializable. In *ACM Transactions on Database Systems*, Volume 30, Issue 2, June 2005, pp.492–528.

[24] Fich, F., Hendler, D. and Shavit, N. On the Inherent Weakness of Conditional Synchronization Primitives. In *Proceedings of the 23rd Annual Symposium on Principles of Distributed Computing*, July 2004, pp.80–87.

[25] Fich, F., Luchangco, V., Moir, M. and Shavit, N. Obstruction-Free Algorithms can be Practically Wait-Free. In *Proceedings of the 19th International Symposium on Distributed Computing*, September 2005, pp.78–92.

[26] Fraser, K. Practical Lock-Freedom. *University of Cambridge Computer Laboratory, Technical Report number 579*, February 2004.

[27] Gao, H., Groote, J. and Hesselink, W. Almost Wait-Free Resizable Hashtables In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004, p.50a.

[28] Greenwald, M. Non-blocking Synchronization and System Design. Technical Report *STAN-CS-TR-99-1624*, Stanford University, June 1999. Ph.D. Thesis.

[29] Greenwald, M. Two-Handed Emulation: How to Build Non-blocking Implementations of Complex Data-Structures Using DCAS. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, July 2002, pp.260–269.

[30] Grinberg, S. and Weiss, S. Investigation of Transactional Memory Using FPGAs. In *Proceedings of the 2nd Workshop on Architecture Research using FPGA Platforms*, February 2006.

[31] Guerraoui, R., Herlihy, M. and Pochon, B. Toward a Theory of Transactional Contention Managers. In *Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2005, pp.258–264.

[32] Guerraoui, R., Herlihy, M., Kapalka, M. and Pochon, B. Robust Contention Management in Software Transactional Memory. In *Proceedings of the OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[33] Hammond, L., Wong, V., Chen, M., Carlstrom, B., Davis, J., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C. and Olukotun, K. Transactional Memory Coherence and Consistency In Proceedings of the 31st Annual International Symposium on Computer Architecture, June 2004, pp. 102–113.

[34] Harris, T. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, October 2001, pp.300–314.

[35] Harris, T., Fraser, K. and Pratt, I. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, October 2002, pp.265–279.

[36] Herlihy, M. and Wing, J. Axioms for Concurrent Objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987, pp.13–26.

[37] Herlihy, M. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, 1988, pp.276–290.

[38] Herlihy, M. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990, pp.197–206.

[39] Herlihy, M. Wait-Free Synchronization In *ACM Transactions on Programming Languages and Systems*, Volume 13, Issue 1, January 1991, pp. 124 – 149.

[40] Herlihy, M. and Moss, J. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 289–300.

[41] Herlihy, M. A Methodology for Implementing Highly Concurrent Data Objects. In *ACM Transactions on Programming Languages and Systems*, Vol. 15, Issue 5, November 1993, pp.745–770.

[42] Herlihy, M., Luchangco, V. and Moir, M. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, May 2003, pp.522–529.

[43] Herlihy, M., Luchangco, V., Moir, M. and Scherer, W. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, July 2003, pp.92–101.

[44] Jayanti, P. An Optimal Multi-writer Snapshot Algorithm. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, May 2005, pp.723–732.

[45] Jouppi, N. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Archictecture*, May 1990, pp. 364–373.

[46] Kirousis, L., Spirakis, P. and Tsigas, P. Reading Many Variables in One Atomic Operation: Solutions With Linear or Sublinear Complexity. In *IEEE Transactions on Parallel and Distributed Systems*, Volume 5, Issue 7, July 1994, pp.688–696.

[47] Knight, T. An Architecture for Mostly Functional Languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, August 1986, pp. 105–112.

[48] Knuth, D. The Art of Computer Programming. Part 3, Sorting and Searching. Addison-Wesley, 1973.

[49] Kumar, S., Chu, M., Hughes, C., Kundu, P. and Nguyen, A. Hybrid Transactional Memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006, pp.209–220.

[50] Lamport, L. Concurrent Reading and Writing. In *Communications of the ACM*, 1977, pp.806–811.

[51] Lamport, L. On Interprocess Communication — Part 2: Algorithms. In *Distributed Computing* 1, 1986, pp.86–101.

[52] Lanin, V. and Shasha, D. Concurrent Set Manipulation Without Locking. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 1988, pp.211–220.

[53] Lea, D. Hash table `util.concurrent.ConcurrentHashMap`, revision 1.3. In JSR-166, the proposed Java Concurrency Package.

[54] Lie, S. Hardware Support for Unbounded Transactional Memory. Doctoral thesis, Massachusetts Institute of Technology, 2004.

[55] Marathe, V., Scherer, W. and Scott, M. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proceedings of the 7th Workshop on Languages, Compilers and Run-Time Support for Scalable Systems*, October 2004.

[56] Martin, D. and Davis, R. A Scalable Non-Blocking Concurrent Hash Table Implementation with Incremental Rehashing. Unpublished manuscript, 1997.

[57] Martin, P., Moir, M. and Steele, G. DCAS-based Concurrent Deques Supporting Bulk Allocation. Tech Report *TR-2002-111*, Sun Microsystems Laboratories, 2002.

[58] Massalin, H. and Pu, C. A Lock-Free Multiprocessor OS Kernel. Tech Report *TR CUCS-005-9*, Columbia University, New York, 1991.

[59] McDonald, A., Chung, J., Chafi, H., Minh, C., Carlstrom, B., Hammond, L., Kozyrakis, C. and Olukotun, K. Characterization of TCC on Chip-Multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, September 2005, pp.63–74.

[60] McDonald, A., Chung, J., Carlstrom, B., Minh, C., Chafi, H., Kozyrakis, C. and Olukotun, K. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006, pp.53–65.

[61] Mellor-Crummey, J. and Scott, M. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. In *ACM Transactions on Computer Systems*, Volume 9, Issue 1, February 1991, pp. 21–65.

[62] Michael, M. Safe Memory Reclamation for Dynamic Lock-Free Objects using Atomic Reads and Writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, July 2002, pp.21–30.

[63] Michael, M. High performance dynamic lock-free hash tables and list-based sets In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures*, August 2002, pp.73–82.

[64] Moore, K., Hill, M. and Wood, D. Thread-Level Transactional Memory. Technical Report 1524, Computer Sciences Dept., UW-Madison, March 2005. Presented at *Wisconsin Industrial Affiliates Meeting*, October 2004.

[65] Moore, K., Bobba, J., Moravan, M., Hill, M. and Wood, D. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture*, February 2006.

[66] Moss, J. and Hosking, A. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proceedings of the ACM OOPSLA Workshop on Synchronization and Concurrency in Object Oriented Languages*, October 2005.

[67] Peterson, G. Concurrent Reading While Writing. In *ACM Transactions on Programming Languages and Systems*, Volume 5, Issue 1, January 1983, pp.46–55.

[68] Plotkin, S. Sticky Bits and Universality of Consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, 1989, pp.159–175.

[69] Purcell, C. and Harris, T. Brief Announcement: Implementing Multi-Word Atomic Snapshots on Current Hardware. In *Proceedings of the 23rd Annual Symposium on Principles of Distributed Computing*, July 2004, p.373.

[70] Purcell, C. and Harris, T. Non-blocking Hashtables with Open Addressing. In *Proceedings of the 19th Annual Symposium on Principles of Distributed Computing*, September 2005, pp.108–121. Extended version published as *University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-639*, September 2005.

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

[72] Rajwar, R. and Goodman, J. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 5–17.

[73] Rajwar, R., Herlihy, M. and Lai, K. Virtualizing Transactional Memory. In *In ACM SIGARCH Computer Architecture News*, Volume 33, Issue 2, May 2005, pp. 494–505.

[74] Ramadan, H, Rossback, C. and Witchel, E. The Linux Kernel: A Challenging Workload for Transactional Memory. In *Proceedings of the Workshop on Transactional Memory Workloads*, June 2006.

[75] Reinholtz, K. Atomic Reference Counting Pointers. In *C/C++ Users Journal*, December 2004.

[76] Riany, Y., Shavit, N. and Touitou, D. Towards a Practical Snapshot Algorithm. In *Theoretical Computer Science*, Volume 269, Numbers 1–2, October 2001, pp.163–201.

[77] Scherer, W. and Scott, M. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[78] Scherer, W. and Scott, M. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, July 2005, pp.240–248.

[79] Scherer, W. and Scott, M. Randomization in STM Contention Management (poster paper). In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, July 2005.

[80] Shavit, N. and Touitou, D. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, August 1995, pp.204–213.

[81] Shriraman, A., Marathe, V., Dwarkadas, S., Scott, M., Eisenstat, D., Heriot, C., Scherer, W. and Spear, M. Hardware Acceleration of Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[82] Sites, R. and Witek, R.. *Alpha AXP Architecture Reference Manual*, Second Edition. Digital Press, 1995.

[83] Sukha, J. Memory-Mapped Transactions. Master's Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2005.

[84] Thakur, M. Transaction Models in InterBase 4. In *Proceedings of the Borland International Conference*, June 1994.

[85] Turek, J., Shasha, D. and Prakash, S. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1992, pp. 212–222.

[86] Vallejo, E., Galluzzi, M., Cristal, A., Vallejo, F., Beivide, R., Stenström, P., Smith, J. and Valero, M. Implementing Kilo-Instruction Multiprocessors. In *Proceedings of the 2005 IEEE International Conference on Pervasive Services*, July 2005.

[87] Valois, J. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, August 1995, pp.214–222.