NON-BLOCKING SYNCHRONIZATION AND SYSTEM DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Michael Barry Greenwald
August 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David R. Cheriton
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Serge Plotkin

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mary Baker

Approved for the University Committee on Graduate Studies:

Abstract

Non-blocking synchronization (NBS) has significant advantages over blocking synchronization: The same code can run on uniprocessors, asynchronous handlers, and on shared memory multiprocessors. NBS is deadlock-free, aids fault-tolerance, eliminates interference between synchronization and the scheduler, and can increase total system throughput.

These advantages are becoming even more important with the increased use of parallelism and multiprocessors, and as the cost of a delay increases relative to processor speed.

This thesis demonstrates that non-blocking synchronization is practical as the sole co-ordination mechanism in systems by showing that careful design and implementation of operating system software makes implementing efficient non-blocking synchronization far easier, by demonstrating that `DCAS` (`Double-Compare-and-Swap`) is the necessary and sufficient primitive for implementing NBS, and by demonstrating that efficient hardware `DCAS` is practical for RISC processors.

This thesis presents non-blocking implementations of common data-structures sufficient to implement an operating system kernel. These out-perform all non-blocking implementations of the same data-structures and are comparable to spin-locks under no contention. They exploit properties of well-designed systems and depend on `DCAS`.

I present an $O(n)$ non-blocking implementation of `CAS`n with extensions that support multi-objects, a contention-reduction technique based on `DCAS` that is fault-tolerant and OS-independent yet performs as well as the best previously published techniques, and two implementations of dynamic, software transactional memory (STM) that support multi-object updates, and have $O(w)$ overhead cost (for w writes in an update) in the absence of preemption.

Finally, I demonstrate that the proposed OS implementation of `DCAS` is inefficient, and present a design of an efficient, hardware, `DCAS` implementation that is specific to the R4000 processor; however, the observations that make implementation practical are *generally* applicable. In short, the incremental costs of adding binary atomic synchronization primitives are very low, given that designers have already implemented unary atomic synchronization primitives.

Acknowledgements

This thesis benefited from the contributions of many people:

Principally, my advisor, David Cheriton, who helped me clarify the ideas I present in this thesis. His advice, suggestions, and questions contributed significantly to making this thesis comprehensible.

Rockwell Corporation, for generously supporting me as a Rockwell Fellow for 5 years.

Ken Duda, Jonathan Stone, Stuart Cheshire, Hugh Holbrook, and other members of the DSG group for enlightenment, discussions, and arguments about the Cache Kernel and NBS and Computer Science in general.

My readers, Serge Plotkinand Mary Baker, for getting helpful comments back to me under incredibly tight schedules.

Several members of the PODC community (particularly Mark Moir, but also James Anderson, Nir Shavit, John Valois, Gideon Stupp, and others) for patiently explaining the world to me from a Theory perspective.

Larry Allen, who worked with me on the SWIFT operating system, and with whom I first discussed the interaction between locks and scheduling and with whom I worked out (blocking) solutions to priority inversion.

David P. Reed, whose work on NAMOS introduced me to alternatives to mutual exclusion. NAMOS and SWALLOW, are still, 20 years later, among the most original and interesting work on synchronization.

Richard Zippel, for first telling me about non-blocking synchronization while we were considering porting Genera to a multiprocessor.

I also must thank my family for bearing with me and often fending for themselves while I finished this thesis; for promises broken and vacations deferred; and for dealing with an end that never seemed to get any closer.

My daughter Alona, who solemnly told her teachers at day-care that the pictures she drew were

for my dissertation; who helpfully asked me every day “Have you finished your dissertation yet, so you can get a real job, so we can get a real house, so I can get a real pet?”; who responded to the question “What is your father writing his thesis on?” with “paper” when she very young, and “on a computer” as she got closer to kindergarten age; and who remembered, for a long while, my tortured and only semi-coherent explanation of non-blocking synchronization as applied to Geoffrey and the rest of her day-care friends simultaneously trying to pour a pitcher of milk.

My son Yaron, who is convinced that I wrote my thesis on what should be *his* computer, but who nevertheless generously deigned to grant me permission to use it whenever I needed it.

My son Meka, who always managed to be no trouble at all when a deadline loomed near.

And, finally, my wife Ronna, whose turn it is now.

PALO ALTO, CALIFORNIA
JULY 1999

A Note on Notation

Several code examples throughout the text (e.g. Figure 3.3 on page 45) include numbered annotations. I refer to these in the text using the following convention: (a:b) refers to the annotation numbered “b” in Figure “a”.

Several places in the code multiple variables are required to be stored inside a single word. I use angle brackets to denote a single machine word (that is, a word that can be read and written in a single processor operation and can be an operand of `CAS`). Thus, `<a,b,c>` denotes a single word containing `a, b,` and `c`.

Contents

Abstract **v**

Acknowledgements **vi**

A Note on Notation **viii**

1 Introduction **1**

1.1 Coordination Among Cooperating Tasks 1

1.1.1 Problems with Mutual Exclusion 3

1.1.2 Problems with non-blocking synchronization 4

1.1.3 Advantages of NBS over blocking synchronization 6

1.2 Results 9

1.3 Original contributions of this thesis 9

2 Related Work **12**

2.1 Background and Definitions 12

2.1.1 Correct algorithms and consistent states 12

2.1.2 Non-blocking synchronization: Definitions 15

2.2 Universality: are non-blocking objects always implementable? 18

2.3 Implementing Universal Primitives 19

2.4 Other Non-blocking primitives 20

2.4.1 Minor modifications to `CAS` and `LL/SC` 20

2.4.2 Richer variants of real hardware primitives 21

2.4.3 Hardware Transactional Memory 22

2.4.3.1 Transactional Memory 22

2.4.3.2 Oklahoma update 23

2.4.4 Constructions of higher arity operators out of Universal Primitives: `CAS`n . . 24
2.5 Non-blocking algorithms/Data-Structures . . . 25
2.5.1 Stacks and Queues . . . 25
2.5.2 List-based priority queues . . . 27
2.5.3 Other data-structures . . . 28
2.6 Applications of NBS . . . 29
2.6.1 Lock-Free Operating Systems . . . 29
2.6.1.1 Synthesis . . . 29
2.6.1.2 Cache Kernel . . . 29
2.6.2 Real-time . . . 30
2.7 Universal Transformations . . . 31
2.7.1 Herlihy's original protocol . . . 31
2.7.2 Transformations based on universal primitives . . . 33
2.7.3 Operating System Support . . . 35
2.8 Conclusion . . . 36

3 Universal Constructions **38**
3.1 Introduction . . . 38
3.2 Universal Algorithms for non-blocking updates . . . 40
3.2.1 The basic non-blocking protocol using `DCAS` . . . 43
3.2.1.1 Implementing `CAS`n from `Double-Compare-and-Swap` . . 43
3.2.1.2 Transformation from locks to NBS: software transactional memory 44
3.2.1.3 Description of the code . . . 48
3.2.2 Refinements to the preliminary versions: contention reduction . . . 50
3.2.2.1 Description of the modifications . . . 52
3.2.2.2 STM with roll-back is effectively non-blocking . . . 55
3.2.2.3 How do we know whether a process is currently running? . . . 55
3.2.2.4 Why use both exponential backoff and currentProcess? . . . 56
3.2.2.5 Summary of STM with contention-reduction . . . 57
3.2.3 Multi-object (multi-domain) updates . . . 59
3.2.4 Some notes on asymptotic performance . . . 61
3.2.5 Extra enhancements to the basic algorithms . . . 62
3.2.5.1 Disjoint-Access Parallelism . . . 63

3.2.5.2 Wait freedom 64
3.3 Related Work 65
3.4 Conclusions 70

4 The Synergy Between Good System Structure and NBS 72

4.1 Introduction 72
4.1.1 Synchronization and system structure 74
4.2 Type-Stable Memory Management (TSM) 75
4.2.1 NBS benefits from type-stable memory management 76
4.2.1.1 The meaning of "type" in type-stability 78
4.2.2 Further benefits of TSM 79
4.2.3 Implementing Type Stable Memory Management (TSM) 80
4.3 Data Structures that Minimize Contention 80
4.3.1 NBS benefits from contention minimizing data structures 81
4.3.2 Further benefits of CMDS 81
4.3.3 Implementing contention minimizing data structures (CMDS) 82
4.4 Minimizing the Window of Inconsistency 83
4.4.1 NBS benefits from minimizing the window of inconsistency 83
4.4.2 Further benefits of minimizing the window of inconsistency 84
4.4.3 Reducing the window of inconsistency 84
4.4.4 Increased robustness through relaxed consistency requirements 84
4.5 Non-Blocking Synchronization Implementation 85
4.5.1 Direct Implementation of common data structures 86
4.5.1.1 The Base Approach: updates consisting of a single write 87
4.5.1.2 Enhancements to the base approach: multiple writes and multiple containers 87
4.5.2 Comparison to blocking synchronization and `CAS`-based non-blocking synchronization 88
4.5.3 Use of NBS in the Cache Kernel 90
4.6 Performance 92
4.6.1 Experimental Implementation 92
4.6.2 Simulation-Based Evaluation 93
4.6.3 Measurements of optimized data structures 98

4.6.4 Contention Reduction 102
4.6.5 Overall System Performance 113
4.7 Related Work 114
4.7.1 Lock-Free Operating Systems 114
4.7.2 General Methodologies for Implementing Concurrent Data Objects 115
4.7.3 Exploiting system properties to support NBS 115
4.8 Conclusions 116

5 A Hardware Implementation of `DCAS` Functionality **118**
5.1 Introduction 118
5.2 Extensions to `LL/SC` Instructions 119
5.2.1 Implementing `DCAS` out of `LLP/SCP` 122
5.3 R4000 Multi-processor Implementation 123
5.3.1 Atomically Storing Two Values 124
5.3.2 Detecting Failure 127
5.3.3 Changes to the System Interface: Cache collisions 127
5.4 Issues that affect the design 129
5.4.1 Cache-line issues and higher arity `CAS`n 129
5.4.2 Deadlock, livelock, and starvation 130
5.4.3 Performance concerns 131
5.4.4 Speculation and `LLP/SCP` 135
5.4.5 Hardware Contention Control 136
5.5 `DCAS`: Hardware vs. OS implementation 137
5.6 Related Work 145
5.7 Conclusions 148

6 Conclusions **150**
6.1 General directions for future research 151
6.2 The future of NBS 153
6.3 Perspective: A Continuum of Transactions 154
6.4 Concluding remarks 155

A Common Myths and Misconceptions about NBS **157**
A.1 Non-blocking algorithms never block 157

A.2 NBS increases concurrency 157

A.3 Concurrency is good 158

A.4 Complicated data structures need to be shared 160

A.5 NBS improves performance 160

A.6 Locks are not non-blocking 161

B Taxonomy of Universal Constructions **162**

B.1 Update-in-place vs. local-copy 162

B.2 Detecting conflicts between transactions 164

B.3 How to proceed when conflict is detected 167

C `CAS`n is Non-Blocking and $O(n)$ **171**

C.1 `CAS`n is non–blocking 171

C.2 `CAS`n is $O(n)$ 177

C.3 Multi-object updates are non-blocking and livelock free 179

D Non-blocking Implementations of Data-Structures **183**

D.1 Introduction 183

D.1.1 Approaches 185

D.2 Common cases 187

D.2.1 List-based stacks 187

D.2.2 Array-based stacks 189

D.2.3 FIFO queues 191

D.2.4 Priority queues: linked lists 191

D.2.5 Double-ended queues: deques 195

D.3 Single server model 198

D.3.1 Examples from other work 198

D.3.2 Single server heap 198

D.3.3 A FIFO queue with a single dequeuer 199

D.4 Reducing contention by adding read-only hierarchy 200

D.5 Relaxing consistency requirements 201

D.5.1 Making internal intermediate states consistent 202

D.5.2 Expose more states to robust clients 203

D.6 Copying small descriptors 206

D.7 Helping 209
D.7.1 Implicit Helping 209
D.7.2 Explicit Helping: roll-forward 210
D.7.3 Roll-back 213
D.7.4 Hybrid approach: roll-back and roll-forward 214
D.8 Adding bits to invalidate states 215
D.8.1 Tag Bits: Concurrent Heap 216
D.8.2 Invalidation: Concurrent Deque 217
D.9 Consistency and hints 218
D.9.1 Doubly-linked lists 221
D.9.2 Array based sets 222
D.10 Would `CAS3` make a difference? 227

E Implementation of Type-Stable Memory Management **228**
E.1 Kernel implementation 228
E.2 User-level implementations 229

Bibliography **231**

List of Tables

4.1 Some Standard Complexity Metrics for implementations of linked-list priority queues 89

4.2 Cyclomatic and Comparison Complexity Metrics for implementations of FIFO queues 89

4.3 Approximate instruction times of extra overhead to synchronize deletion from a priority queue. 93

4.4 Variance of Valois' algorithm, work = 2000 98

5.1 State used by processor and primary cache controller in implementation of `LLP/SCP` 120

List of Figures

1.1 Multiple processes concurrently accessing a single shared output queue 2

1.2 Three processes deadlock because of circular dependencies while waiting for resources. 3

1.3 Pseudo-code definition of `DCAS` (`Double-Compare-and-Swap`) 10

2.1 Specification of a `max` object, and implementation of `record` function. 16

2.2 Pseudo-code definition of `CAS` (`Compare-and-Swap`) 18

2.3 Pseudo-code definition of `CCAS` 21

2.4 Pseudo-code definition of `C-Lock` 21

2.5 Implementation of a LIFO stack used in Synthesis. 26

2.6 Herlihy's Large Object Protocol, a tree before modification. 32

2.7 Herlihy's Large Object Protocol, tree after modification, but before atomic update. 33

2.8 Herlihy's Large Object Protocol, list with last element modified, before atomic update. 33

3.1 Basic code used to manage logs. 42

3.2 Basic code supporting atomic transactions using `DCAS`. 44

3.3 Code for preliminary `CAS`n implementation using DCAS. 45

3.4 Transforming code updating data structure d using locks to a non-blocking update. 46

3.5 Routine implementing software transactional memory. 47

3.6 Enhanced `trans_open` to support contention reduction. 51

3.7 Enhanced `trans_conclude` and `trans_begin` to support contention reduction. 54

3.8 Routines implementing software transactional memory. 58

3.9 Multi-object enhancements (continued) to routines implementing software transactional memory. 59

3.10 Code changes for multi-object `CAS`n. 60

4.1 Deletion from the middle of list, protected by DCAS and version number. 77
4.2 Performance of several synchronization algorithms with local work = 20 and the number of processes per processor = 1 95
4.3 Performance of several synchronization algorithms with local work = 2000 and the number of processes per processor = 1 96
4.4 Performance of several synchronization algorithms with local work = 2000 and the number of processes per processor = 3 97
4.5 Performance of stacks with 1 process per processor 99
4.6 Performance of stacks with 1 process per processor, and one background computation 101
4.7 Performance of stacks with 1 process per processor, and 2 background computations 102
4.8 Throughput comparison of contention reduction techniques. Number of transaction pairs per 100k cycles. 104
4.9 Comparing contention reduction techniques: Bus contention 104
4.10 Comparison of cache hit ratios for several contention reduction techniques. 104
4.11 Average number of retries for each contention reduction technique, short transactions. 106
4.12 Average number of retries for each contention reduction technique, long transactions. 106
4.13 Average number of failing DCAS's, short transactions. 106
4.14 Average number of failing DCAS's, long transactions. 106
4.15 Pseudo-code explanation of "retries" and "aborts". 108
4.16 Throughput comparison of contention reduction techniques, with background process. 108
4.17 Bus contention of contention reduction techniques, with background process. . . . 109
4.18 Comparison of cache hit ratios for several contention reduction techniques in the presence of background computations. 109
4.19 Throughput comparison of contention reduction techniques. Number of transaction pairs per 100k cycles. 110
4.20 Bus contention. 110
4.21 Comparison of cache hit ratios of several contention reduction techniques. 110
4.22 Average number of retries for each contention reduction technique. This graph represents the performance of the short transactions. 111
4.23 Average number of retries for each contention reduction technique. This graph represents the performance of the long transactions. 111
4.24 Average number of failing DCAS's. Comparison between different contention reduction techniques. This data is for short transactions. 111

4.25 Average number of failing DCAS's. Comparison between different contention reduction techniques. This data is for long transactions. 111
4.26 Average number of retries for each contention reduction technique. This graph represents the performance of the short transactions. 112
4.27 Average number of retries for each contention reduction technique. This graph represents the performance of the long transactions. 112
4.28 Average number of failing DCAS's. Comparison between different contention reduction techniques. This data is for short transactions. 112
4.29 Average number of failing DCAS's. Comparison between different contention reduction techniques. This data is for long transactions. 112

5.1 Pseudo-code definition of DCAS (Double-Compare-and-Swap) 119
5.2 High level language description of LL rt,offset(base). 120
5.3 High level language description of LLP rt,offset(base). 121
5.4 High level language description of SCP rt,offset(base). 121
5.5 High level language description of SC rt,offset(base) 122
5.6 DCAS implementation using LL/SC and LLP/SCP. 123
5.7 Example showing why a distinct LLP is needed in addition to LL 124
5.8 Behavior of Cache Controller during SCP and SC 125
5.9 Throughput of system with work=100 using hardware DCAS. 140
5.10 Throughput of system with work=100 using hardware DCAS without acquiring exclusive access on an LL. 140
5.11 Throughput of system with work=100 using OS assisted DCAS. 140
5.12 Throughput of system with work=1000 using hardware DCAS, where LL(P) acquires exclusive access of the cache line. 141
5.13 Throughput of system with work=1000 using hardware DCAS without acquiring exclusive access on an LL. 141
5.14 Throughput of system with work=1000 using OS assisted DCAS. 141
5.15 Throughput of system with 64 structures using hardware DCAS. 143
5.16 Throughput of system with 64 structures using hardware DCAS without acquiring exclusive access on an LL. 143
5.17 Throughput of system with 64 structures using OS assisted DCAS. 143
5.18 Hardware DCAS, LL exclusive access, mostly local structures. 144

5.19 Hardware DCAS, LL no exclusive access, mostly local structures. 144
5.20 OS assisted DCAS, mostly local structures. 144

B.1 Relationships between categories of update algorithms. 163

D.1 Code implementing Treiber's non-blocking stacks 187
D.2 Push on stack (list). 188
D.3 Pop from stack (list). 189
D.4 Push on stack (array). 190
D.5 Pop from stack (array). 190
D.6 Enqueue on FIFO Queue. 192
D.7 Dequeue from FIFO Queue. 192
D.8 Insertion into middle of a singly-linked list. Note that list is initialized with a single, "dummy" entry. 193
D.9 Deletion from the middle of a singly-linked list. 194
D.10 Insertion and deletion at the top of the deque 196
D.11 Insertion and deletion from the bottom of a deque 197
D.12 Dequeue code for FIFO Queue with single dequeuer. 199
D.13 Definition of incrRefCount 207
D.14 Insertion into middle of a singly-linked list that maintains a count of length. 207
D.15 Deletion from the middle of a singly-linked list that maintains a count of list length. 208
D.16 Main step for insert and delete into doubly linked list. These routines set the forward pointer (next) for doubly linked lists. 211
D.17 Main utility routine to perform operations on doubly-linked lists. 212
D.18 Insertion and deletion calls from the middle of a doubly-linked list. 213
D.19 Insertion and deletion at the top of the concurrent deque 219
D.20 Insertion and deletion from the bottom of a concurrent deque 220
D.21 Insert in set (array). 223
D.22 Delete from set (array). 224

Chapter 1

Introduction

Over the past 20 years, the 'mutual exclusion phenomenon' has emerged as one of the best paradigms of the difficulties associated with parallel or distributed programming. It must be stressed at the outset that the implementation of a mutual exclusion mechanism is a very real task faced by every designer of operating systems.

— Gerard Roucairol, 1986

1.1 Coordination Among Cooperating Tasks

When many agents, whether human or machine, try to cooperate using shared objects, some form of coordination or synchronization is necessary. Coordination increases the cost of using shared objects. Therefore, we generally *first* try to design the system as a whole to *reduce* the sharing, and thus lessen the coordination needed.

In well-designed systems sharing is minimized, but not eliminated. The infrequent sharing arises from several causes. First, there are true dependencies between parts of the task: particles in a parallel simulation depend not only on the past state of this particle but on the state of other, independent, particles. Second, resource limitations may force sharing: multiple applications on a single computer communicating over a network all share access to a limited number of network interfaces. Third, some communication between otherwise independent sub-tasks may be necessary for, say, load-sharing.

When the infrequent sharing *does* arise, some form of synchronization is needed to ensure correct behavior of the system.

The simplest form of coordination is to take turns when using shared objects — each agent gets sole possession of the shared object(s) for the time needed to finish a job. In computer systems, the analogue of "taking turns" is called "*mutual exclusion*". If one agent has control of a shared object, it has *exclusive* access to that object. Other agents requiring access block or spin until the first agent is finished with it. In this simple form of coordination, the method we use to *synchronize* access to the object is mutual exclusion.

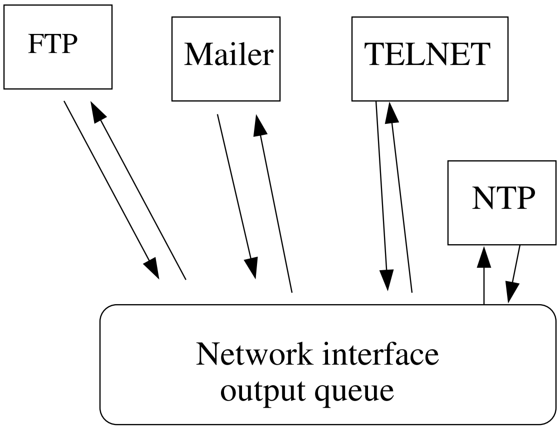

Figure 1.1: Multiple processes concurrently accessing a single shared output queue

An alternative to mutual exclusion is *non-blocking synchronization* (NBS). Rather than taking turns and waiting, NBS is based on the notion of giving back the shared object if it is part of an operation that takes too long. A synchronization algorithm is non-blocking if we can guarantee that even if some agent using the shared object stalls, fails, or even dies, some other agent can use that object after a finite amount of time. Non-blocking synchronization is defined more formally in Section 2.1.2.

Despite the many advantages non-blocking synchronization has over mutual exclusion (also referred to as blocking synchronization), NBS has been dismissed as impractical in general (although occasionally useful in specialized circumstances). NBS has been dismissed because it is unfamiliar, often complex to implement, and in many common cases has performance that is significantly worse than equivalent blocking algorithms.

This thesis is the first to identify properties of systems and missing hardware needed to make non-blocking synchronization generally practical. I demonstrate that by adding a modest amount of hardware support and exploiting properties already present in well-designed systems, NBS is practical and is a desirable alternative to mutual exclusion.

1.1.1 Problems with Mutual Exclusion

In some situations . . . the user is assumed to be a superprogrammer who delights in writing programs that explicitly deal with concurrency.

— Jim Gray, 1993

Mutual exclusion is attractive because it *seems* straightforward. However, care must be taken to avoid unexpected effects caused by multiple agents trying to acquire exclusive access to shared resources.

Deadlock Mutual-exclusion can cause the whole system to come to a halt if circular dependencies exist. In such a case, every job is stalled waiting for another job to complete. Suppose job P_1 acquires resource X and tries to acquire resource Y, but task P_2 already has acquired Y. P_1 must wait for P_2 to release Y. If however, P_2 is holding Y while waiting for resource Z, P_1 now has an indirect dependence on Z, and will be unable to proceed until after (at least) P_2 acquires Z and releases Y. If Z is held by job P_3, which is in turn waiting for X, we have a circular dependency, since P_1 cannot proceed to acquire Y and (eventually) release X, until X is released. This situation is called *deadlock*, or deadly embrace. Care must be taken to either avoid or recover from deadlock.

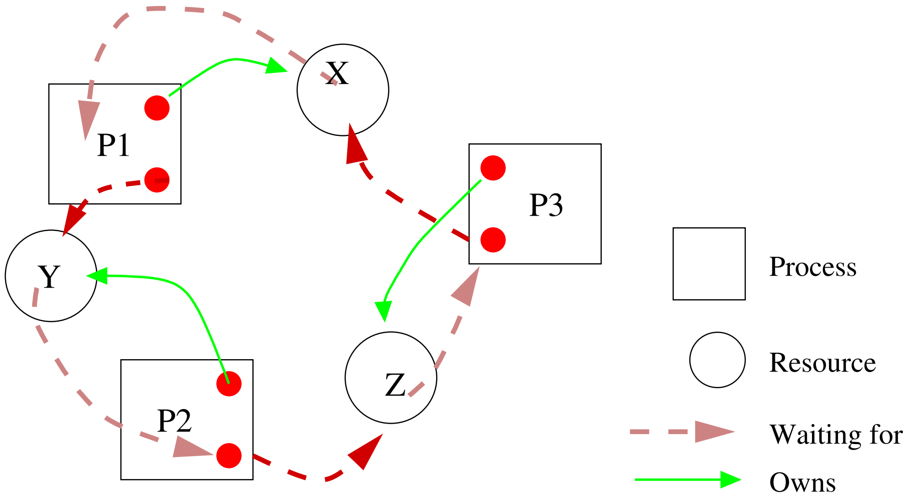

Figure 1.2: Three processes deadlock because of circular dependencies while waiting for resources.

Silberschatz and Peterson[90] quote a colorful example of deadlock from the Kansas State Legislature, early in the 1900s: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

Long Delays Mutual-exclusion can cause all jobs to experience long delays when one job happens to be delayed while being the exclusive owner of a shared resource. If P_1 acquires exclusive ownership of X, then *all* waiters are delayed when P_1 takes a long time to complete its task. If the delay is due to the length of an intrinsic part of the computation, then it is unavoidable. We may have been able to reorder the owners, but eventually this time must be spent.

However, some long delays may be avoidable. If P_1 takes a page fault while owning X, and P_2 is waiting and ready to go, then P_2 may not have had to wait if it had acquired X before P_1. If the average computation while the lock is held is on the order of a hundred instructions then this single page fault might dominate the execution time of many processes. Even worse is the ultimate long delay: failure. If P_1 fails while holding the lock for X, no other processes may ever proceed to work on X without some form of recovery. Recovery is not always possible, and P_2 may now effectively share P_1's fate and be permanently stalled.

Priority Inversion Mutual-exclusion can cause lower priority jobs to take precedence over higher priority jobs. Assume P_1, P_2, and P_3 are, respectively, low, medium, and high priority tasks. If P_1 and P_3 share a resource, and agree to synchronize by mutual exclusion, then if P_2 is ever running while P_1 holds the resource, P_2 will take precedence over P_3, subverting the scheduler's priorities.

Priority inversion[57] almost cost U.S. taxpayers $265 million[89] in 1997. The computer on the Pathfinder mission to Mars reset itself several times due to priority inversion[1], jeopardizing the success of the mission.

1.1.2 Problems with non-blocking synchronization

Non-blocking synchronization is attractive because it avoids the problems listed above. Despite these attractions, however, it has not been widely deployed to date. There are three main impediments to its use.

Conceptual Complexity Non-blocking synchronization may require subtle or complex algorithms. Very few data structures have obvious non-blocking implementations. To date, each non-blocking algorithm has constituted a publishable result. The algorithms that do exist were

[1] A detailed account is available in a well-publicized email summary by Mike Jones of a talk [101] given by David Wilner, CEO of vxWorks (the company that provided the real-time system used on Pathfinder), and also in followup email by Glenn Reeves [83], the JPL project leader for Pathfinder software.

hard to design; once designed they are difficult to verify as correct; once the algorithms are believed to be correct, implementations are difficult to debug.

Performance Non-blocking algorithms must guarantee that an object is always in a state where progress can be made — even if the current process dies. NBS can incur extra cost to support that guarantee. When special-purpose non-blocking implementations of data structures exist, they can usually out-perform equivalent locking implementations since they combine the synchronization with modifications to the data. Unfortunately, few such algorithms exist. Certain needed data-structures (such as priority queues) have no known efficient implementations using only conventional NBS approaches and current hardware support (c.f. [100, 46]). Most non-blocking data-structures must be derived through *universal transformations*. Universal transformations are mechanical translation protocols that take as input a sequential specification or algorithm and outputs a provably equivalent non-blocking concurrent algorithm. They are defined and discussed in Section 2.7. Universal transformations are easy to write, understand, and reason about (and, by definition, provide non-blocking implementations for all data structures), yet their performance is poor compared to spin-locks in the (common) absence of preemption.

Further, care must be taken when using non-blocking algorithms to avoid performance degradation due to contention. Naive non-blocking algorithms do not prevent multiple processes from simultaneously accessing a single data structure; such contention can degrade memory system performance. Solutions exist to reduce contention[39, 2], but they add to the complexity of the non-blocking algorithms.

Unfamiliarity Non-blocking synchronization is relatively new. Even when complexity and performance are comparable to locking implementations, programmers are apt to be most comfortable with techniques they have had most experience with.

NBS is sufficiently similar to other synchronization techniques that many people assume that they understand the goals, properties, and methods of non-blocking synchronization. Unfortunately, some people incorrectly carry over beliefs that are true of these other, similar, techniques, but are not true when applied to NBS. These beliefs may deter people from using NBS, or may raise expectations so that disappointment is inevitable. (Appendix A briefly corrects several common misconceptions. Although the appendix does not present any new results, reading it first may clarify the rest of the thesis.)

In simple cases, locks are easy to use. Although non-blocking algorithms out-perform blocking algorithms in the presence of delays, the common case has no delays and spin-locks perform well. Locks are a mature technology; although many problems exist, a well-known body of work has addressed these problems. Therefore, locks have seemed more attractive than NBS to some implementors.

1.1.3 Advantages of NBS over blocking synchronization

There is one fundamental problem that stands out from all those involved in controlling parallelism: mutual exclusion.

— Michel Raynal, 1986

There is consensus on one feasible solution [to concurrency control] — locking. *(Though there are many other solutions, one is hard enough. ...)*

— Jim Gray, 1993

The fundamental advantage of non-blocking synchronization over blocking synchronization is that NBS provides isolation between processes accessing a shared object. Using NBS, if process P_1 accesses a critical resource X, P_2 can still proceed and operate on X, so no artificial dependence exists between P_1 and P_2. Blocking synchronization not only links processes accessing a shared object, but introduces new dependencies, such as order (to avoid deadlock), and priority (to avoid priority inversion).

Although in general, the costs of NBS have been high enough to outweigh the benefits, there are systems in which the cost of NBS was comparable to the cost of blocking synchronization. When the costs of non-blocking synchronization are comparable, NBS is clearly preferable to blocking synchronization. For example, Massalin and Pu [62, 63, 64] designed the Synthesis Kernel to use only NBS for synchronization. Similarly, we chose to use NBS exclusively in the design and implementation of the Cache Kernel [24] operating system kernel and supporting libraries. This preference arises because non-blocking synchronization (NBS) has significant advantages over blocking synchronization.

- First, *NBS allows synchronized code to be executed anywhere*. Non-blocking synchronization allows synchronized code to be executed in an interrupt or (asynchronous) signal handler without danger of deadlock. For instance, an asynchronous RPC handler (as described in [103]) can directly store a string into a synchronized data structure such as a hash table even though it may be interrupting another thread updating the same table. With locking, the signal handler could deadlock with this other thread.

- Second, *non-blocking synchronization minimizes interference between process scheduling and synchronization*. For example, the highest priority process can access a synchronized data structure without being delayed or blocked by a lower priority process. In contrast, with blocking synchronization, a low priority process holding a lock can delay a higher priority process, effectively defeating the process scheduling. Blocking synchronization can also cause one process to be delayed by another lock-holding process that has encountered a page fault or a cache miss. The delay here can be hundreds of thousands of cycles in the case of a page fault. This type of interference is particularly unacceptable in an OS like the Cache Kernel where real-time threads are supported and page faults (for non-real-time threads) are handled at the library level. Non-blocking synchronization also minimizes the formation of convoys which arise because several processes are queued up waiting while a single process holding a lock gets delayed.

- Third, *non-blocking synchronization aids fault-tolerance*. It provides greater insulation from failures such as fail-stop process(or)s failing or aborting and leaving inconsistent data structures. It allows processes to be killed with impunity — even with no cleanup. Non-blocking techniques allow only a small *window of inconsistency* (the window during which a data structure is in an inconsistent state), namely during the execution of atomic primitives themselves, such as compare-and-swap. In contrast, with lock-based synchronization the window of inconsistency may span the entire locked critical section. These larger critical sections and complex locking protocols also introduce the danger of deadlock or failure to release locks on certain code paths.

- Fourth, *non-blocking synchronization can reduce interrupt latencies*. Systems that synchronize using NBS can avoid disabling interrupts for long periods of time. Instead, interrupts can proceed, and the currently running operation will retry. It is important to note that interrupts are not *required* to preempt the currently running operation. (Indeed, work by Mogul and Ramakrishnan [72] notes that under high load it is important to *schedule* interrupt handlers to avoid receiver livelock). Rather, NBS gives the system the ability to *decide* on the highest priority task dynamically. In contrast, disabling interrupts unilaterally gives the current operation priority over interrupts.

- Fifth, *non-blocking synchronization supports system maintainability*. Independently designed modules can be composed without knowledge of their internal synchronization details. In contrast, consider two independently designed and implemented concurrent databases that use locking for synchronization. It is difficult to remove a relation atomically from one database and insert it into the other. If each operation manages its own locks, then the compound operation isn't atomic — an outside observer may observe the state of both systems before the insertion completes. This compound operation must then be aware of the internal locking strategies of both databases, in order to acquire all the locks before the deletion and hold ownership over them until after the insertion completes. An even more serious problem arises if the choice of internal locks is dependent upon the arguments passed to each database [82]. In such cases, the compound operation must mimic the internal behavior of each database to determine which locks to acquire! Worse, more complicated compound operations may result in deadlock.

- Finally, there are additional minor advantages. *NBS aids portability*: the same synchronized code can run in uniprocessors, asynchronous handlers, and on shared memory multiprocessors. It may even enhance portability across operating systems — Ryan [85] cites synchronization as the main impediment hampering portable device drivers (simultaneously synchronizing with interrupt handlers as well as with concurrent processes, and understanding different kernel interrupt levels). *Non-blocking synchronization increases total system throughput* by allowing other processes to proceed even if a single process modifying a shared data structure is delayed. It allows synchronization even where mutual-exclusion is forbidden due to the need to solve the confinement problem[2].

These advantages are becoming even more important as we see an increase in the use of parallelism and multi-processors, an increase in real-time programming, increased use of signals and class-libraries, and as the cost of a delay (e.g. page-fault, cache miss, or descheduling) increases relative to processor speed.

It is clear that there is often some cost to NBS. Objects must be kept in a state where they are always preemptible. One *should* expect to pay a higher cost (in time or space or conceptual complexity) to guarantee stronger properties. This thesis demonstrates *practical* non-blocking algorithms. That is, loosely speaking, algorithms in which the cost of implementation of a given

[2] The *confinement problem*[55] requires a system to confine a program during its execution so that it cannot communicate with anyone except its caller. Mutual-exclusion on a shared resource provides a mechanism that readers can modulate as a signal to establish a covert channel back to the writer, even if the information flow is unidirectional. In a non-blocking algorithm with unidirectional information flow, readers have no way of signalling the writer.

property is justified by the benefits obtained by possessing that property.

1.2 Results

This thesis demonstrates that non-blocking synchronization (NBS) is practical as the sole coordination mechanism in well-designed systems that provide the necessary underlying primitives. In support of this claim:

- I show that careful design and implementation of operating system software for efficiency, reliability, and modularity makes implementing simple, efficient non-blocking synchronization far easier.
- I demonstrate that `DCAS` (`Double-Compare-and-Swap`) is the necessary and sufficient primitive for implementing *efficient* non-blocking synchronization.
- I demonstrate that an efficient, hardware, `DCAS` implementation is practical for contemporary RISC processors that already support single `CAS` (`Compare-and-Swap`) functionality.

1.3 Original contributions of this thesis

In support of these results, this thesis presents several original contributions.

This thesis identifies properties of systems that can be exploited to make the writing of NBS algorithms easier and more efficient. It is possible for *any* system to reap the benefits of NBS without paying disproportionate costs, by ensuring that the system possesses these properties.

Using these properties and `DCAS` functionality[3], I designed non-blocking implementations of several common data-structures. These algorithms perform better than all previously published non-blocking implementations of the same data-structure and perform comparably to spin-locks even under no contention. They are simpler than previous algorithms because they are shorter, and certain race conditions are impossible and need not be addressed. These algorithms include a set of performance critical data structures sufficient to implement an operating system kernel. Some of these critical data structures have no known efficient non-blocking implementation using only unary `CAS`.

[3] Both the Cache Kernel and Synthesis were implemented on systems that provided a hardware implementation of `Double-Compare-and-Swap` (or `DCAS`), a 2 address generalization of `Compare-and-Swap` (or `CAS`). `DCAS` is formally defined in Figure 1.3

```
int DCAS(int *addr1, int *addr2,
         int old1,   int old2,
         int new1,   int new2)
{
 <begin atomic>
 if ((*addr1 == old1) && (*addr2 == old2)) {
   *addr1 = new1;  *addr2 = new2;
   return(TRUE);
 } else {
   return(FALSE);
 }
 <end atomic>
}
```

Figure 1.3: Pseudo-code definition of DCAS (Double-Compare-and-Swap)

As further evidence that DCAS is the necessary and sufficient primitive for non-blocking synchronization I present several algorithms for CASn built out of DCAS: An $O(n)$ non-blocking version, and an $O(n \min(n, p, M/S))$ multi-object version. An $O(np)$ wait-free version can be synthesized from my non-blocking CASn and conventional wait-free techniques such as proposed by Moir [74]. (p is the number of processes actively contending for a data structure, M is the total size of memory, and S is the average granularity of our unit of contention reduction.) Algorithms for CASn that use only unary CAS are at least $O(n \log p)$.

I present a contention-reduction technique based on DCAS that performs as well as the best previously published techniques, yet, unlike those, also offers fault tolerance, OS independence, and fairness.

I present two implementations of dynamic, software transactional memory that support multi-object updates, and have $O(w)$ overhead cost (for w writes in an update) in the absence of preemption. The version using roll-forward (so-called "helper" functions) is strictly non-blocking. The version using roll-back, however, is only "effectively non-blocking": if every process is infinitely preempted before completing, then progress is only "guaranteed" probabilistically. To limit the probability of failing to make progress to a given probability P, the algorithm must retry $O(1/P)$ times. These retries still allow a relatively efficient universal transformation from any synchronization using locks to an equivalent lock-free version.

I present a design proposal for an efficient, hardware, DCAS implementation that is specific to the R4000 processor; however, the observations that make implementation practical are *generally* applicable. In short, the incremental costs of adding binary atomic synchronization primitives are very low, given that designers have already implemented unary atomic synchronization primitives.

Finally, I show that an OS-based implementation of `DCAS` using the technique of Bershad [17] is not a practical alternative to hardware support for `DCAS`, mostly due to memory contention on shared (global) locks.

Chapter 2

Related Work

This chapter describes significant related work in context, attempting to provide historical background to the contributions of this thesis. Here, I focus mostly on describing *what* other researchers have done. Detailed comparisons between their work and mine are left to the individual chapters in which the relevant contributions of this dissertation are presented.

2.1 Background and Definitions

Atomic operations are indivisible. That is, while an atomic operation is executing, no other operations may see any intermediate state of any objects touched by the atomic operation. This kind of atomicity is referred to as "atomicity under parallelism", or *coordination atomicity*. Atomic operations are either *committed* (completed successfully) or not committed (they may eventually commit, or they may already have aborted or failed). If an atomic operation aborts or fails no intermediate state may be left — it is as if the operation had never been attempted. This is known as "atomicity under expected errors", or *failure atomicity*.

2.1.1 Correct algorithms and consistent states

The results of applying primitive atomic operations to a shared object are, by definition, correct. When concurrent *compound* operations are applied to a shared object, a useful system should also ensure that the resulting state is "correct". The following definitions show that we must be careful to choose a definition of "correctness" that matches our intuition.

The most basic form of correctness is internal consistency. An object is in an *internally inconsistent* state S if there is a legal operation whose result, when the operation is applied to S, is undefined.

We are usually concerned with an even stronger level of consistency. More interesting conditions of correctness are usually defined with respect to results of equivalent operations applied sequentially to an unshared version of the object. Assume object D starts in some known state S_0. Assume that a sequence σ of N operations, $O_i, i = 1..N$, are applied (possibly concurrently) to D. The final state, S_N, of D is *consistent* w.r.t. σ if S_N corresponds to the state D would be in if some permutation of σ were applied to D (starting from state S_0) sequentially. That is, the result must be equivalent to some result of applying these same operations (in any order) when none of them overlap. If no operations overlap, then each operation only sees the *results* of previous operations — not intermediate states. This isolation is a prerequisite to considering the compound operations atomic. The sequential permutation of the O_i's is the *validating sequence*, and this sequence, σ, is *serializable*.

If an algorithm ensures that *every* legal sequence of operations is serializable, then the algorithm is serializable. Note that serializibility does *not* require that there be any correspondence between the order of the concurrent operations, O_i, and their permuted order in the validating sequence. Thus, serializable algorithms still may produce surprising results when we try to reason about them. They might not correspond to our intuitive notion of correct behavior.

A slightly stronger consistency requirement is *sequential consistency*. Sequential consistency imposes restrictions on the set of permutations acceptable as a validating sequence. The relative order of $\sigma|P$ (the set of O_i that were invoked on processor P), for each processor, must be identical to the order of the O_i in the validating sequence, but the relative order of operations on different processors is unconstrained.

An even stronger consistency requirement is *linearizability*. Loosely speaking, linearizability restricts the operations in the validation sequence to be in the same order (for non-overlapping operations) as the real-time order of non-overlapping concurrent operations. (No restrictions are made on the order of overlapping operations.) More precise treatments of linearizability are available in [44] which defined the notion, and in "Distributed Computing", by Attiya and Welch[13], pages 196–200. An alternate (but equivalent) formulation of linearizability is that each operation appears to take effect instantaneously at a particular point between the operation's invokation and response. This point is called the *point of linearization*.

The distinctions between these consistency requirements are more apparent in a concrete example. Consider a variable x, with an initial value of 20. Two processors simultaneously execute the following operations:

Timestep	Processor 1		Processor 2	
	Operation	Implementation	Operation	Implementation
0.0	$x = x + 1;$	`lw r0, x`	$x = x + 2;$	`lw r0, x`
0.1		`addi r0, 1`		`addi r0, 2`
0.2		`sw r0, x`		`sw r0, x`
1.0	$x = x/2;$	`lw r0, x`	$x = x/3;$	`lw r0, x`
1.1		`divi r0, 2`		`divi r0, 3`
1.2		`sw r0, x`		`sw r0, x`

Intuitively, the result should be 23/6. The additions should execute before the divisions. The order of the additions is indeterminate, but addition is commutative, so we would expect x to go from 20 to 23 and then to 23/6 (division is also commutative). In keeping with our intuition the only result of these operations that satisfies linearizability is 23/6.

Sequential consistency relaxes the consistency requirement between processors. Under sequential consistency, the result of applying both operations of Processor 1 (yielding 21/2), before Processor 2 even starts is acceptable. 21/2 + 2 = 25/2. 25/2 divided by 3 = 25/6. If Processor 1 executes before Processor 2 (or vice versa) the result is 25/6. Therefore there are *two* sequentially consistent results: 23/6 and 25/6.

Serializability relaxes the consistency requirements even further, as no order is imposed upon the validation sequence. Therefore, any permutation of the 4 operations is consistent with serializability. There are 13 distinct valid results under serializability: any multiple of 1/6 (except $5\frac{1}{6}$) between $3\frac{5}{6}$ and $6\frac{1}{3}$ are consistent under serializability.

Finally, there are internally consistent results which are inconsistent under all definitions of correctness. Assume the steps in the "Implementation" column are atomic, but the operations themselves are not, and so implementation steps of different operations may be interleaved. Consider the case when both Processor 1 and Processor 2 complete all steps in the implementations through timestep 1.1 and both are ready to execute 1.2. x contains 23, Processor 1's `r0` contains 23/2, and Processor 2's `r0` contains 23/3. If Processor 2 executes 1.2 after Processor 1 does, then x will contain 23/3, which is larger than any serializable result. $x = 23/3$ (or 23/2) will not raise any errors under normal arithmetic operations, but it is not consistent with any validating sequence starting with $x = 20$ and consisting of the 4 specified operations. Some form of synchronization is needed to ensure that x always takes on values consistent with the specified operations under some definition of correctness.

2.1.2 Non-blocking synchronization: Definitions

A synchronization algorithm is *interference-free* if it does not contain any sections of code where we grant exclusive access to one owner. Formally, an algorithm is interference-free over an object (or component of an object), O, if we guarantee that at least one process may legally modify O after waiting a finite time. Intuitively, this means that even if some process stalls or fails while logically owning an object, other processes can eventually proceed to operate on that object.

However, one can imagine pathological failure cases for interference-free algorithms. Assume that every time process P_1 performs some operation on O, process P_2 preempts P_1, undoes its work and starts some of its own. P_1 will never make progress. Further assume that some other process, P_3, similarly preempts P_2, and so on up to some process P_n which is always preempted by P_1, creating a cycle. Such a system will be interference-free, and will do a lot of (useless) work, but each process will undo the work of the previous (preempted) process, and nothing useful will ever get accomplished.

We can describe a stronger property than interference freedom by incorporating some guarantee that the modifications allowed by interference freedom are actually useful ones. An algorithm is *non-blocking* (sometimes called *lock-free*) if it guarantees that at least one process can complete a task, or *make progress*, within a finite time. Intuitively, "making progress" means performing some useful work that will never be undone. The precise definition of progress is dependent on the sequential specification of each particular object. For a simple banking example the legal operations are Deposit and Withdrawl. Progress consists of a completed deposit or withdrawl with balance updated. For a *max* object, for example, which recorded the largest argument passed to `record` since the last call to `reset`, many processes can concurrently make progress since `record(N)` can immediately return if `max` is already larger than N.

Finally, a non-blocking algorithm is *wait-free* if we guarantee that *every* process makes progress within some finite time. Wait-freedom encapsulates a weak notion of fairness as well as progress. We say that wait-freedom only weakly supports fairness, since it only guarantees that starvation is impossible, it does not guarantee any notion of "fair share". Non-blocking algorithms can use exponential backoff to avoid starvation with high probability. Wait-freedom provides a *deterministic guarantee* that starvation will be avoided. This guarantee comes at a cost. Herlihy [39] notes that as a practical matter "the probabilistic guarantee against starvation provided by exponential backoff appears preferable to the deterministic guarantee provided by [wait-freedom]".

Similarly, in practice, algorithms obeying the strict definition of non-blocking (let alone wait-free) are not always necessary — or rather the cost to obtain strictly non-blocking algorithms might

```
class max {
  int  maxValue_;           /* Largest value seen to date */
  int  initialized_ = 0;    /* Have we seen *any* values yet? */
public:
       max();
  void reset();            /* clears state of object */
  void record(int datum);  /* records new data point */
  int  value();            /* Returns largest value seen
                            * since reset(); */
}

void max::record(int datum)
{
  while (TRUE) {
    int lastMax = maxValue_;
    if (lastMax >= datum) { return(); }
    else                  { CAS(&maxValue_,lastMax,datum); }
  }
}
```

Figure 2.1: Specification of a `max` object, and implementation of `record` function.

not be justifiable. Slightly weaker properties are sufficient, and the overall behavior might be better. It is useful to capture the notion that an algorithm is strictly non-blocking in all the interesting cases (c.f. "swap-tolerant"[1] algorithms mentioned in [87]), or, alternatively, is non-blocking with very high probability (for example the algorithm is non-blocking with probability 1). Unfortunately, it is unrealistic to assume that the failure case ("bad luck") occurs with low probability. Hard won experience with systems has shown that some bad luck can pathologically recur. Instead of simply assuming that failure cases have low probability, we consider an interference-free algorithm to be *effectively non-blocking* if the algorithm is robust enough to still make progress with high probability *in spite of* a pathologically recurring failure case (say, infinite preemptions). In terms of the examples above, an algorithm is effectively non-blocking if it is *both*, say, swap-tolerant and non-blocking with probability 1.

We can define this notion of "effectively non-blocking" a little more formally. Given a model, an adversary, and a set of "pathological failures", we define the probability that an algorithm fails to make progress in time T as STALL(T). (Hence STALL(T) is non-increasing as $T \to \infty$.)

First, we consider the algorithm under a slightly weakened adversary: we assume an adversary

[1] "Swap-tolerant" algorithms (analogous to fault-tolerant) are algorithms that are tolerant (i.e. are non-blocking) in the face of a finite number of process swaps and long delays, but are *not* tolerant of failures.

with all the strengths of the original, but that can only cause a finite sequence of transactions to pathologically fail. We consider only algorithms that are strictly non-blocking against this slightly weakened adversary.

For such algorithms, we next examine the given model and the original stronger adversary. If $T \cdot \text{STALL}(T)$ is $O(1)$ or better, *even assuming an infinite sequence of pathological failures*, then the algorithm is *effectively non-blocking* under that model, adversary, and definition of pathological failure.

An example may motivate the notions of "pathological failure" and "effectively non-blocking algorithms" more clearly. Consider an algorithm which is strictly non-blocking in the absence of preemption, but may not make progress if a second process runs while the first is preempted. If the string of preemptions is finite, then a non-preempted run will occur, and progress is guaranteed. If there is a run during which no second process is available to run, then preemption is harmless, and, again, progress is guaranteed. We might be tempted to say that because preemption is rare one of these two cases will eventually occur, and the algorithm in question will progress. This is realistic, and corresponds to the definition of the weakened adversary.

However, although the weakened adversary is realistic, we must consider that a parallel algorithm may deterministically take a page fault near the start of *every* execution. In this example we can make the algorithm effectively non-blocking by adding exponential backoff. Recall that an effectively non-blocking algorithm guarantees that even in the face of such pathological failure the algorithm will make progress with probability 1.

Exponential backoff guarantees progress with probability 1, because after some point, no new processes are added to the set of actively contending processes (the number of processes is bounded). Doubling the waiting interval roughly halves the probability that any single process will wake up to make an attempt during the time the "owning" transaction is preempted — even assuming every process is preemted.

Of course, simply adding exponential backoff to an algorithm is not enough to make if effectively non-blocking — unless the original algorithm was strictly non-blocking under the restricted adversary.

There is a minimal cost necessary to implement *any* of the properties mentioned in this section, from non-interference to wait-freedom. Each object, O, must always be preemptible. That is, we know it is necessary that some other process may proceed should the currently executing process fail. Therefore, we know that there is enough data stored outside the current process (i.e. accessible in memory from O, and not explicitly in the registers or stack, or even implicitly in the PC, of any particular process) to transform O into a consistent state such that another process may operate upon it.

2.2 Universality: are non-blocking objects always implementable?

An object is *universal* if it can be used as a building block to provide a wait-free implementation of *any* other object.

Herlihy [38] showed the connection between universality and the *consensus problem* [79, 30]. He proved that any object with consensus number n is universal (in the sense that it can be used to build a wait-free implementation of *any* object) in a system of at most n processors. Objects with infinite consensus numbers are universal in all systems.

Herlihy [38] further demonstrated that if one assumed unbounded memory, then `Compare--and-Swap` (`CAS`) and the pair `Load-linked/Store-Conditional` (`LL/SC`) are both universal, and thus, that it is possible to implement non-blocking or wait-free algorithms for *every* data--structure given machines with either of those primitives. Plotkin [80] proved that an object called a *sticky bit* was universal even with bounded memory. A sticky bit is an object that can take on one of three values: $\perp, 0$, or 1. If multiple processes try to write the value of a sticky bit concurrently, only one of them succeeds. The sticky bit returns “failure” if the process was trying to write a value that disagrees with the value already written in the sticky bit, otherwise it returns success. Plotkin's result is important, since it shows that even a fixed width `CAS` (as long as it can hold at least three values) has infinite consensus number, and is universal in all systems.

Given that universal synchronization primitives exist on a particular platform, *every* data-structure can be implemented in a non-blocking or wait-free manner.

```
int CAS(T *location, T oldValue, T newValue)
{
 <begin atomic>
 if (*location == oldValue) {
   *location = newValue;
   return(TRUE);
 } else {
   return(FALSE);
 }
 <end atomic>
}
```

Figure 2.2: Pseudo-code definition of `CAS` (`Compare-and-Swap`)

`CAS` was introduced in the IBM 370 architecture [45], and `LL/SC` was introduced on the S-1 [51]. In recent years hardware designers have provided at least one universal primitive on almost all processor architectures, usually either `CAS` or `LL/SC`. (`CAS` is defined in Figure 2.2. The instruction pair `Load-linked/Store-Conditional(LL/SC)` is defined at the start of Chapter 5.) `LL/SC` is available on MIPS [36], ALPHA [92], and PowerPC [76] processors, among others. Pentium, SPARC [96] (as of v9), and others, support `CAS` directly.

2.3 Implementing Universal Primitives

Prior to Herlihy's work, very few modern processors supported *any* universal synchronization primitives. The RISC revolution strove for minimal instruction sets [37], and atomic `test-and-set` was sufficient to implement locks. RISC designers eschewed the `CAS` and `LL/SC` instructions in favor of the simpler, but non-universal, primitives.

Bershad proposed a relatively efficient software mechanism for implementing such missing simple primitives [16] on multiprocessors. It extended earlier work by Bershad et al. [18] on *restartable atomic sequences*, which only implemented atomic sequences on uniprocessors.

Both [18] and [16] are based on the observation that the operating system is aware of any "long delay" that may happen to a process. The OS can then restart an atomic action that experiences a delay before the operation completes. In the multiprocessor case, the sequence is protected by a lock. Waiting processors spin on the lock before performing the operation. The lock-hold time is bounded, because the critical section is short and the OS releases the lock if the process is delayed. The operation is atomic with respect to processes on the same processor since the OS will restart the sequence if preempted. The operation is atomic with respect to processes on *other* processors because it is protected by a lock. Thus, despite the lack of hardware implementation of universal non-blocking primitives on most processors, work could go forward using the technique of Bershad.

Despite the current ubiquity of universal primitives, Bershad's work is still important to this thesis because I propose a hardware implementation of `DCAS`. Bershad's software techniques provide an alternative to my hardware proposal. I discuss his technique in detail in Section 5.5.

2.4 Other Non-blocking primitives

Several proposals for adding hardware support for richer non-blocking synchronization primitives exist. They fall into two camps: the minimalist approach and the maximalist approach. The minimalists propose modest extensions to existing universal primitives that can be used to implement non-blocking transactions using software protocols. The maximalists propose systems which would completely support non-blocking transactions in hardware.

Why is there a need for richer primitives if `LL/SC` and `CAS` are universal? Based on work by Attiya and Dagan [12], software implementations would not necessarily scale. Attiya and Dagan prove that an $O(1)$ binary `LL/SC` cannot be implemented using only `CAS` or `LL/SC`. They show that the *maximal independence set* problem[2] has an $O(1)$ solution assuming an $O(1)$ binary universal primitive (e.g. binary `LL/SC` or `DCAS`), and depend upon a proof by Linial [60] that the lower bound for a solution using only unary `CAS` or `LL/SC` must be sensitive to the number of processes in the system. Thus, no $O(1)$ implementation of binary operators out of unary primitives exist, and therefore software support for higher arity primitives based only on hardware single `CAS` has potential scalability problems.

2.4.1 Minor modifications to `CAS` and `LL/SC`

Most processors provide at most single `Compare-and-Swap` (`CAS`) functionality to support non-blocking synchronization. Many processors provide a double *width* version of `CAS` — such instructions do not support two independent words, they merely allow `CAS` (or `LL/SC`) to operate on wider words (e.g. 64 bits).

A few older processors provided an independent multi-word atomic instruction. The IBM 370 architecture supported an extension to `CAS` — an instruction called `CDS` (`Compare and Double Swap`). `CDS` atomically stores `new1` in `*addr1` and `new2` in `*addr2` if-and-only-if `addr1` contains `old`. Motorola supported full `Double-Compare-and-Swap` functionality with the `CAS2` [75] instruction in the 680x0 family of microprocessors. (`DCAS` is described in Figure 1.3 on page 10)

Both `CDS` and `CAS2` are complex, expensive, instructions. Equivalent functionality is not yet present in any modern RISC processor.

Anderson et al. [6] proposes `Conditional-Compare-and-Swap` (`CCAS`, see Figure 2.3).

[2] The maximal independence set problem is defined on a ring of processors. A solution returns a maximum sized subset of the processors such that every processor in that subset has no neighbors in the set.

```
boolean CCAS(Word *addr1, Word *addr2,
             Word oldval1, Word oldval2,
             Word newval)
{
  boolean retval = FALSE;
  <begin atomic>
  if ((*addr1 == oldval1) &&
      (*addr2 == oldval2)) {
    *addr1 = newval;
    retval = TRUE;
  }
  <end atomic>
  return(retval);
}
```

Figure 2.3: Pseudo-code definition of `CCAS`

```
boolean C-Lock(Word *addr1, Word oldval,
               Word *addr2, Word newval);
{
  boolean retval = FALSE;
  <begin atomic>
  if (*addr1 == oldval) {
    *addr2 = newval;
    retval = TRUE;
  }
  <end atomic>
  return(retval);
}
```

Figure 2.4: Pseudo-code definition of `C-Lock`

James [48] proposes `Conditional Lock` (`C-Lock`, see Figure 2.4). These each modify one location, but need to deal with two addresses. `CCAS` differs from `CAS` in that it atomically checks *two* locations before performing the single write. Like `CAS`, `C-Lock` checks only one location, but unlike `CAS`, it writes a *different* location if the checked location is unchanged. Both `CCAS` and `C-Lock` are motivated by the desire to better support universal constructions; they do not necessarily improve direct implementations of specific data structures. `CCAS` is useful in hard real-time environments where, if certain conditions are met, `CCAS` can be implemented in software. `C-Lock` is a weaker version of `DCAS`, designed to avoid the complexity introduced by implementing starvation avoidance in hardware.

2.4.2 Richer variants of real hardware primitives

Many non-blocking algorithms (e.g. [47, 87]) assume systems that support richer primitives than the hardware allows. In some cases software implementations support the extended primitives.

A common "unrealistic" assumption is support for an arbitrary number of outstanding, overlapping, `LL/SC` pairs. Real implementations of `LL/SC` support at most one `LL/SC` at a time on each processor. Moir [73] showed that the richer version of `LL/SC` can be constructed on any system that supports either `LL/SC` or `CAS`. This implementation is achieved by storing a version number in *every* location (thus all memory locations contain both data and a version number). On systems that support double-width `CAS` or `LL/SC` two words of memory are used to represent each word of data. On systems that support only single width `CAS`, version numbers use fewer bits and full width data cannot be stored atomically in a single word. Every `LL` returns both the data and the version number. `SC` is implemented as an `LL` that verifies the version number or aborts, followed by an `SC` that stores both the requested data and increments the version number. These extended `LL/SC` pairs do not immediately support `CAS`n in general, or `DCAS` in particular (even overlapping `LL/SC`s cannot guarantee that a first `LL` will only succeed if the second will be successful). However, several algorithms implementing `CAS`n from the extended `LL/SC` exist (e.g. [47, 74]).

Similarly, some researchers assume `CAS` and `LL/SC` can operate on words of arbitrary width (e.g. [47]). Anderson and Moir [3] showed that single-word `CAS` or `LL/SC` are sufficient to implement relatively efficient ($O(W)$ worst-case time for a W-word variable) atomic operations on arbitrarily wide words.

Both of these software implementations require a substantial amount of space per word (ranging from $O(\log p)$ bits per word, to $O(P)$ bits per word, where p is the number of processes actively contending at a given moment and P is the maximum number of processes in the system).

2.4.3 Hardware Transactional Memory

2.4.3.1 Transactional Memory

Herlihy and Moss' *Transactional Memory* [43] proposes hardware support for multiple-address atomic memory operations. It supports arbitrary read-modify-write operations with arity limited only by the size of a processor's cache.

The basic insight behind Transactional Memory is the observation that the ownership protocol already required to keep caches coherent is sufficient to detect transaction conflicts. By piggybacking the transactional protocol on top of the cache line ownership protocol, atomic transactions can be supported with no extra bus traffic. Commit and abort then become purely local operations.

Implementation of Transactional Memory requires significant enhancements to the processor and cache. Transactional memory requires six new instructions and a second primary cache. This transactional cache is fully associative, and contains space to buffer two copies (the original and modified values) of each cache-line.

Within a transaction, transactional loads and stores are treated normally, except that stores are contingent upon `commit`. Until then, new values are written to the "modified" field of the line in the transactional cache. `Commit` simply verifies that the processor still owns all the cache lines read or written in this transaction. If so, it copies the modified values out of the transactional cache to regular memory. Otherwise it aborts, clearing the transactional cache.

Attempting to acquire ownership of a cache line already owned by another transaction results in an abort. (Such attempts normally happen in the course of a `store`.) Slow/delayed transactions are aborted, locally, by timer interrupts or quantum expirations. Thus, only undelayed transactions that conflict with no one are able to commit. The `commit` operation executes atomically with respect to interrupts, so once a slow transaction begins the commit operation, it can safely proceed, and commit is guaranteed to complete successfully.

Herlihy and Moss simulated a transactional memory implementation and reported results that out-performed locking implementations for all data structures they tested under all circumstances.

2.4.3.2 Oklahoma update

Stone et al. [93] independently proposed the *Oklahoma Update*, which exploits the existing ownership protocol in cache coherency protocols in a fashion similar to Transactional Memory. The Oklahoma Update adds a set of special *reservation register*s to the processor. Each special reservation register consists of 5 fields: `address`, `data`, `valid`, `write-privilege`, and `updated`.

Transactional loads behave like normal loads; in addition, they also locally update a specified reservation register. Transactional stores are purely local; they only update the reservation register and do not affect the cache or memory. At commit time, the Oklahoma Update goes through two phases (patterned after two-phase locking protocols). The *precommit phase* sorts the addresses in ascending order, checks to make sure they are all valid, and tries to acquire write privileges for any reservation that does not yet have write privilege. (Acquiring exclusive access in ascending order avoids the possibility of deadlock.) If any reservation is invalid (uninitialized, or this processor was informed by the normal cache coherence protocol that some other processor changed the value), or if write-privilege was not granted, then `commit` fails. On success, the `commit phase` atomically updates all modified shared variables by copying the `data` field into the primary cache. On failure, the modified variables are reset and the operation must be retried to complete. During commit all external ownership requests are deferred. During precommit phase, all external ownership requests for addresses lower than the currently active address are deferred until after commit.

Stone et al. do not deal with loss of cache ownership resulting from a cache line being evicted due to conflict misses and does not report any (simulated) performance figures. The Oklahoma Update does not guarantee consistency between operations that share sets of read-only locations; the burden is on the programmer to make sure that overlapping transactions always share one *written* location. On the other hand, the hardware cost in terms of instructions and cache space is lower than Transactional Memory.

The complexity of both Transactional Memory and Oklahoma Update is significant, and to date, neither has been considered for real implementation, due to their cost and complexity. The cost is not just due to attempting to support atomic operations over many locations, but also because of the structure of their solutions. Transactional Memory requires a second, fully associative, primary cache, and adds several new instructions. Oklahoma Update adds only a moderate number of new instructions, but sorts addresses and implements exponential backoff in hardware, and does not isolate the performance of transactions from each other due to the deferred ownership requests during the commit phase.

2.4.4 Constructions of higher arity operators out of Universal Primitives: CASn

Israeli and Rappaport [47] demonstrate a non-blocking implementation of CASn (n-way atomic `Compare-and-Swap`) and n-way `LL/SC` for P processors out of single `LL/SC`. They augment each word of memory with 2 fields: `value` and `proc`. `value` holds the current value of the word. `proc` specifies whether the word is currently involved in any CASn operation. Each process records its operation in a P element table (P is an upper bound on the number of processes in the system). CASn proceeds in 2 passes. First, each location (in ascending order) is checked to see whether it contains the required old value. If so, and if the CASn hasn't been aborted, and if the location is not yet locked by another CASn, then the location is locked by this process. If the location *is* already locked by another CASn, the current process helps the other owner finish, thereby freeing this location for other CASns. This two-phase locking approach is the basis for almost all other CASn implementations.

The CASn implemented by Israeli et al.[47] was the first *disjoint-access parallel* implementation. Disjoint-access parallel implementations allow two or more transactions to proceed in parallel, without interference, if they access disjoint sets of addresses. Their CASn was not wait-free, however, and had worst-case time complexity $O(nP^3)$, required at least an extra P bits for *every* word in memory, and required overlapping `LL/SC` pairs that operated atomically on words P bits wide.

Anderson and Moir [4] improved upon the helping mechanism of [47] to implement a wait-free CASn, rather than merely non-blocking. They require only realistic sized words, and impose no worst-case performance penalty for the wait-free property — the $O(nP^3)$ worst-case time is still the same. Unfortunately, they also still require a prohibitively large amount of space.

Shavit and Touitou[87] demonstrated that the recursive helping used to free locations locked by competing transactions incurred a large overhead in practice. They enhanced performance by only performing *non-redundant helping*. In such a scheme a transaction T helps only enough other transactions that a subsequent attempt at T would succeed. If the transaction finds it needs to recursively help, it aborts the transaction instead.

Finally Moir [74] proposed a more efficient non-blocking version of `CAS`n based on his earlier wait-free implementation in [4]. The efficiency gains are realized by relaxing the wait-free property (observing that wait-free algorithms can be constructed out of primitives that are merely non-blocking), by incorporating the non-redundant helping of [87], and by introducing optimizations such as allowing failing `CAS`ns to abort early. The most significant gains were made by relaxing the wait-free property — this removed an $O(P)$ loop from every read. The Moir MWCAS still requires extra space *per word* in the shared memory, however.

All of the implementations of `CAS`n exhibit ingenuity (or complexity, depending upon your point of view) and high space overhead. Only Moir's has worst-case time better than $O(nP)$.

2.5 Non-blocking algorithms/Data-Structures

The most effective use of non-blocking synchronization has been the direct implementation of data--structure-specific algorithms. Michael and Scott [69] reviewed a number of such implementations for several common data structures. They concluded that where direct non-blocking implementations exist, they generally out-perform locking implementations in all cases. Locking implementations out-perform *universal* non-blocking constructions when processes do not experience long delays such as page-faults, context switches, or failures. Universal constructions out-perform locks in the presence of long delays and, of course, the non-blocking constructions enjoy the advantages not related to performance, such as fault tolerance and freedom from deadlock.

Many specific non-blocking algorithms have been published. I concentrate on a representative sample.

2.5.1 Stacks and Queues

Treiber [84] proposed a straightforward, non-blocking, linked-list implementation of LIFO stacks. It performs as well or better than all published list-based stack implementations. There is a description of the algorithm in Section D.2.1, and a detailed discussion of its performance in Section 4.6.3. The stack `top` pointer is double-width and includes a version number to avoid race conditions in the `pop` operation. Every modification of `top` also atomically increments the version number. (Chesson [28] designed an optimized version of list-based stacks that function correctly with no version numbers, assuming that a single reader (`Pop`) exists.)

As part of the Synthesis Kernel [64], Massalin et al. also implemented non-blocking stacks — the more powerful `CAS2` instruction is capable of supporting non-blocking *array-based* stacks.

```
Push(elem)
{
retry:
  old_SP = SP;
  new_SP = old_SP-1;
  old_val = *new_SP;
  if (CAS2(old_SP,old_val,
           new_SP,elem,
           &SP, new_SP)
      == FAIL)
    { goto retry; }
}
```

```
entry *Pop()
{
retry:
  old_SP = SP;
  new_SP = old_SP+1;
  elem = *old_SP;
  if (CAS(old_SP,new_SP,&SP)
      == FAIL)
    { goto retry; }
  return(elem);
}
```

Figure 2.5: Implementation of a LIFO stack used in Synthesis.

Synthesis' array-based stacks were implemented using the code in Figure 2.5.

Synthesis used `CAS2` for `Push`, but tried to be clever and use only unary `CAS` for `Pop` (`CAS` is measurably cheaper than `CAS2`). Unfortunately, there is a race condition in the Synthesis code. Consider a process performing a `Pop`. If the process goes blocked after reading `elem` from `*old_SP`, but before performing the `CAS` in `Pop`, and another process performs K `Pops` followed by K `Pushs`, then the wrong `elem` will be returned when the original `Pop` resumes. `elem` will be returned twice, and the current value of `*old_SP` will never be returned. (If there is only a single distinguished process that always does the `Pops`, then this race condition cannot occur).

This problem is not insurmountable. I present a new, correct, implementation in Section D.2.2, using `DCAS` for both `Push` and `Pop`.

Michael and Scott [70] present an implementation of a list-based FIFO queue. Their approach uses a double-width `CAS` (not `DCAS`) to include a version number in each pointer. They relax the requirement that the tail pointer in the structure always point to the last element of the queue, but require all routines to update `tail` before using it.

This implementation is non-blocking, allows enqueues and dequeues to proceed concurrently, and out-performs all locking and non-blocking FIFO queue implementations in the literature. (My `DCAS` based algorithm in Section D.2.3 moderately outperforms it, though, by avoiding the use of version numbers and by avoiding the need to repair the `tail` pointer.)

Chesson [28] and Muir [77] independently came up with a slightly different algorithm. They avoid version numbers, but are vulnerable to process failure at a critical juncture. The result is an algorithm that performs comparably to Michael and Scott, but is not fault-tolerant. Massalin et al. [63] use `DCAS` to implement array-based circular FIFO queues, avoiding allocation costs. They present

4 algorithms, optimized for (1) single producer/single consumer, (2) single producer/multiple consumers, (3) multiple producers/single consumer, (4) fully general queues.

2.5.2 List-based priority queues

Priority queues are critical data structures for operating systems and many applications. Prior to our work on the Cache Kernel [34], no fully general, efficient, non-blocking priority queue was available.

One sequential implementation of a priority queue is a sorted linked list. However, non-blocking deletion of nodes in the interior of the list is problematic. Assume we want to delete node N_1, whose predecessor is N_0. Deletion must set $N_0 \rightarrow$`next` to point to $N_1 \rightarrow$`next`. However, there is always the possibility that N_0 will itself be deleted from the list. Herlihy[41] proposed a general technique for dealing with such problems. Reference counts are kept for each pointer referencing an entry in a list. Any attempt at deletion of an object with a non-zero reference count results in failure, and the deletion must be retried. (The reference count and pointer can be updated by `DCAS`).

Massalin and Pu [64] used `CAS2` to implement link-based priority run queues in Synthesis. To avoid the overhead of managing reference counts, they simply marked nodes for deletion but left them in the list. Subsequent passes only delete marked nodes if they are *safe*. A node is safe if it is not marked for deletion, and it is reachable from the head of the list. Deletion at the head is trivially safe. Deletion in the middle is safe if the deleter is holding an unmarked node and pointing to a marked node. However, we need some guarantee that the unmarked node will not be deleted during the operation (after you decide to delete but before you delete the next node). Herlihy's reference count is one method of ensuring the safety of the node, but is expensive. Massalin exploited a property of Synthesis run queues — namely that only one thread can visit a node at a given time. In such a system, a single binary marker is sufficient (it is also simpler than a counter, since it need not be read before update — it must be unmarked to allow entrance). A marked node is busy, and the traverser simply skips over it. Once you ensure the safety of a node, you can delete using `CAS2` by making sure that the next pointers of both the safe predecessor and the deleted node do not change during the operation.

The reference count approach is inefficient, since it requires `CAS` or `DCAS` for each element in the list. The Massalin and Pu optimization is useful but still expensive, and is only applicable in limited cases, such as the scheduler's run queues in Synthesis.

Valois [100] designed another implementation of a list-based non-blocking priority queue. He used only `CAS`(arguing that `DCAS` was not generally available), and supported concurrent updates (both deletions and insertions). In contrast, Herlihy's general technique only allows one operation to succeed at a time.

Valois required every "normal" entry in the list to be flanked by auxiliary nodes consisting only of a `next` pointer (they contained no `value`). Insertions insert both a normal node and an auxiliary node, between an auxiliary node and a normal node, maintaining this invariant. Note that adjacent pairs, or even chains, of auxiliary nodes are permitted. Deletion occurs in two steps. First, the normal node is deleted. Second, chains of consecutive auxiliary nodes are trimmed to single nodes.

The auxiliary nodes are required for correct behavior using only `CAS`. All operations on the list are performed using a cursor. The cursor contains three pointers. `pre_aux` points at the preceding auxiliary node. `pre_cell` points at a preceding "normal" node. `target` points at the node of interest. (Reference counts to each of these nodes are incremented before installing these pointers into the cursor. Reference counts to a node, N, are also incremented if any other node has a pointer to N). If the state of any of these three nodes changes, an operation is aborted and retried. If the state has not changed, then no conflict occurred. Auxiliary nodes with non-zero reference counts are never deleted — therefore, transitively, no auxiliary nodes later in the list can be deleted either. Normal nodes that are deleted contain back-pointers to the `pre_cell`, guaranteeing that a chain can be followed to a valid node inside the list (perhaps all the way to the head). Once pointing to a node in the list, `next` pointers can be followed to find a currently valid predecessor to any other node still in the list.

Unfortunately, this algorithm is very expensive. Further, a process experiencing a long delay or failure can cause memory to grow without bound (since no auxiliary nodes can be deleted once an auxiliary node has a non-zero reference count). Even worse, these auxiliary nodes remain *in* the list, causing the cost of list traversal to increase as time goes by. Finally, the algorithm is very complex. The published work has typos, and later researchers [68] discovered race conditions. The algorithm is hard to describe and understand.

2.5.3 Other data-structures

Non-blocking implementations of sets [58] and wait-free implementations of Union-Find [8] have been known for some time. More recently, Shavit [88] published a non-blocking algorithm for a *diffracting tree* (an algorithm for a concurrent counter), and a wait-free implementations of Quicksort [78]. The heap-based priority queue implementation of Israeli and Rappaport [46] is particularly interesting because it depends on an $O(1)$ binary atomic primitive (`LL/SC2`), because it implements a priority queue, and because it illustrates a clever technique for designing non-blocking algorithms for specific data structures. (The technique is described in Section D.8.1 in Appendix D).

2.6 Applications of NBS

2.6.1 Lock-Free Operating Systems

To date, two purely non-blocking operating systems have been designed and implemented: Synthesis V.1 and the Cache Kernel. Each chose non-blocking synchronization from different motives, and used slightly different approaches, but both had one thing in common: they were implemented on the same processor. This is not coincidental. The 680x0 family of micro-processors are the only processors to support `DCAS` functionality in hardware.

2.6.1.1 Synthesis

All synchronization in Massalin and Pu's lock-free (non-blocking) Synthesis V.1 multi-processor kernel [63, 64, 62] was non-blocking. The main motivation for the use of non-blocking synchronization in the Synthesis kernel was reducing synchronization overhead and reducing latency for critical tasks.

Synthesis ran on dual-68030 workstations, and was thus able to take advantage of the `CAS2` instruction. The designers of Synthesis considered using Herlihy's universal construction to implement non-blocking implementations of needed data structures. However, the relatively high average case CPU cost and memory overhead (even when there was no contention) made it impractical for a low-overhead system such as Synthesis. Fortunately, the presence of `CAS2` made it possible to implement specially designed non-blocking objects directly for Synthesis. Synthesis designers began with a small set of non-blocking data-structures ("quajects"): shared counters, stacks, queues, and lists. They tried to implement all shared data objects using variations on one of these types.

Massalin and Pu report that every object of interest needed by Synthesis was directly implementable using `CAS2`. The resulting system achieved very high performance. Thus, they concluded that a non-blocking multi-processor OS was both practical and efficient.

2.6.1.2 Cache Kernel

The Cache Kernel [24] was also a purely non-blocking system. The Cache Kernel chose non-blocking synchronization from slightly different motives [34]:

- The Cache Kernel was “signal centric”. The basic form of IPC was through signals, and real work was done in signal handlers. NBS was needed to avoid deadlock, and to avoid the overhead of disabling and enabling signals. NBS allowed signal handlers to safely access shared data structures.

- The Cache Kernel supported real-time user threads, and kernel events, such as page faults, were handled at the library level. Therefore, long delays introduced by lock-holding processes were unacceptable.

- The Cache Kernel implemented most operating system mechanisms at the class library level. NBS allows the class library level to be tolerant of user threads being terminated (fail-stopped) in the middle of performing some system library function such as scheduling or handling a page fault.

In contrast with Synthesis, which used a pre-packaged set of non-blocking “quajects”, the Cache Kernel designed data structures as needed, and then tried to apply one of several general techniques to make the data structures non-blocking. The techniques are described in Appendix D. The resulting non-blocking objects (such as our linked-list priority queues) are general, where possible. Rather than exploit data-structure-specific properties, the Cache Kernel designers depended on properties that might generally be found in well-designed systems. This approach was motivated by the desire to use the same non-blocking objects in user class libraries, as well as in the kernel.

In common with Synthesis, the Cache Kernel synchronized every shared object in a non-blocking fashion. All algorithms were direct implementations that performed well — we never had to resort to expensive, universal constructions.

2.6.2 Real-time

NBS seems to be gaining acceptance in the real-time community in contrast with the more cautious appraisal in the general purpose operating systems community. This is partly because the advantages of non-blocking algorithms are more valuable in real-time systems, but also because non-blocking algorithms are easier to implement in hard real-time systems. The ease of implementation arises from two causes. First, primitives are more “powerful” in real-time systems; that is, weaker primitives are universal, and universal primitives can be used to construct more efficient algorithms. Second, real-time systems have strong properties that can be exploited by non-blocking algorithms.

For example, Ramamurthy et al. [81] show that by exploiting strong properties present in real-time operating systems, weaker primitives than `CAS` or `LL/SC` can be universal. In particular, they show that `reads` and `writes` are universal for hard real-time applications on uniprocessors. `Reads` and `writes` are usually cheaper than strong synchronization primitives, so the result of [81] potentially allows cheaper non-blocking implementations of data structures.

Their work was extended by Anderson et al. [6], exploiting the observation that on hard real-time systems with no preemption, *any* operation executed by a higher priority process is atomic with respect to lower priority processes on the same processor. In [6] they prove both that `CCAS` is implementable in software using only `CAS` given the strict priorities of a hard real-time system, and that `CCAS` supports an optimal implementation of `CAS`$n$on real-time systems.

2.7 Universal Transformations

2.7.1 Herlihy's original protocol

Despite the proof that `CAS` and `LL/SC` are universal, relatively few data structures have known, efficient, non-blocking implementations. A major impediment to the use of NBS is the ease of reasoning about the algorithms, or their "conceptual complexity". "A practical methodology should permit a programmer to design, say, a correct lock-free priority queue, without ending up with a publishable result"[39]. The "practical methodology" proposed by Herlihy and others to overcome the conceptual complexity of non-blocking algorithms is the use of *universal construction*s or *universal transformation*s (e.g. [41, 14, 54, 4, 3, 87]). A universal construction is a mechanical translation protocol that takes as input a sequential specification or algorithm (a specification of the desired behavior in the absence of concurrency), transforms it, and outputs a provably equivalent non-blocking or wait-free concurrent algorithm. Since sequential algorithms are well understood and (relatively) easy to reason about, the conceptual burden on the programmer is lightened.

Herlihy [39] proposed the first general technique to make any algorithm or data-structure non-blocking. Each object is referenced through an extra level of indirection, $*D$. Before applying an operation on `orig` $= *D$, the entire data structure is copied and the modification is applied to the copy. After the modification is complete, the program updates $*D$ using `CAS`, only if $*D ==$ `orig`. The cost of copying the entire object renders this technique obviously impractical for large objects.

Herlihy proposed a new transformation, the *Large Object Protocol*. This protocol takes a data-structure made up of blocks connected by pointers. Only blocks which are modified, or contain

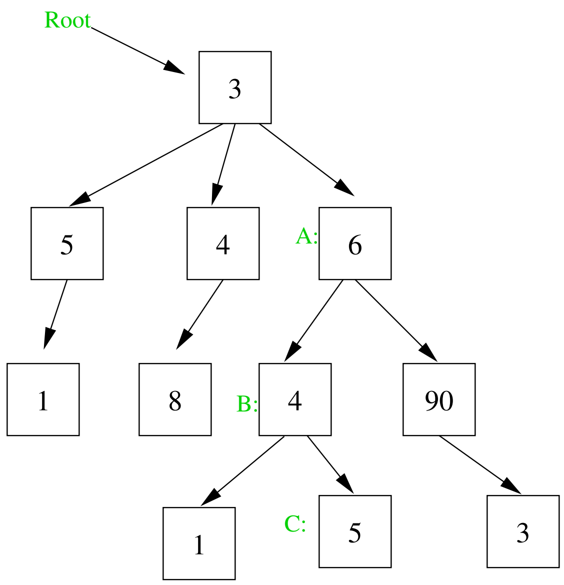

Figure 2.6: Herlihy's Large Object Protocol, a tree before modification.

pointers to modified blocks, need be copied. A parallel structure is constructed with pointers to both modified and unmodified blocks. When the root is updated by `CAS`, the new structure contains new copies of each modified block.

This approach still has several problems. First, the copying costs are still significant, since the entire path of blocks from the root to each modification must be copied. Second, without some form of contention reduction, contending processes degrade performance. Third, the burden of decomposing the object into blocks falls on the programmer. Herlihy accepts this, asserting "Whenever possible, correctness should be the responsibility of the system, and performance the responsibility of the programmer." A poor decomposition simply results in bad performance, and not in an incorrect program. However, it is not simply that the burden is on the programmer — the fourth problem is that for some data-structures *no* decomposition performs well. For example, it is hard to see *any* reasonable decomposition for a priority queue implemented as a linked list. Any attempt to insert or delete an entry near the end of the list will result in the entire list being copied. A final issue with the Large Object Protocol is the complexity and expense of the storage management of the old blocks.

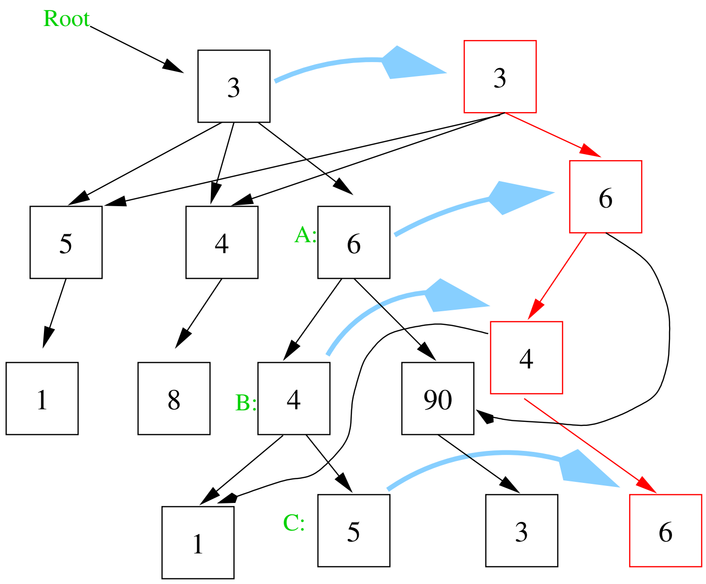

Figure 2.7: Herlihy's Large Object Protocol, tree after modification, but before atomic update.

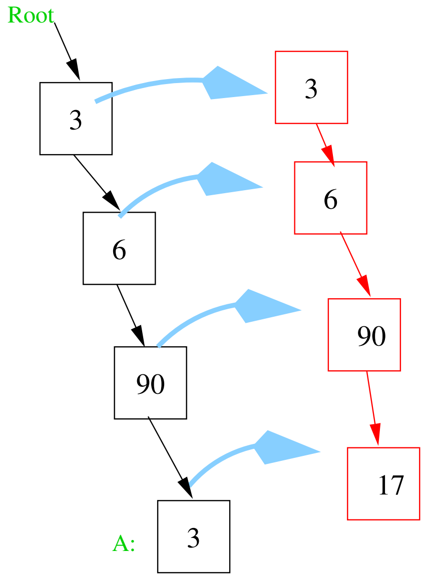

Figure 2.8: Herlihy's Large Object Protocol, list with last element modified, before atomic update.

2.7.2 Transformations based on universal primitives

Subsequent approaches are more sophisticated than simply copying the data structure. They reduce the amount of copying, increase the level of parallelism, and lessen contention that arises due to multiple processes hammering on the same data-structure. Nevertheless, they still exhibit poor average case performance compared to spin-locks (when there's no preemption or delays) and compared to custom non-blocking implementations in *all* cases. They have not been successful as a "practical methodology" — their high cost compared to spinlocks has rendered NBS implementations based on universal constructions impractical.

Many of these subsequent approaches are modifications of Herlihy's general copying methodology. These proposals range from Herlihy's original approach of copying [39] a single block covering the entire data structure, through breaking the data structure into several smaller blocks[39, 4, 74], to individual words[14, 47, 87, 1]. Correspondingly, the number of locations updated atomically range from a single "root"[39], through a small number of roots[4, 74], to using `CAS`n to atomically update each of n individual words[14, 47, 87, 1]. Even approaches that copy blocks of more than one word use `CAS`n whenever there are multiple roots. Section 2.4.4 described implementations of `CAS`n, often introduced as part of a universal construction.

`CAS`n works on static transactions — transactions where all of the locations and values are known in advance. Most universal constructions use a technique similar to, or derived from, Barnes [14], who was the first to support dynamic transactions by the *caching method*. The caching method implements dynamic transactions by running the transaction on shadow copies, recording all reads and writes, and then re-running the reads and writes as a static transaction using `CAS`n to replace all the original values with the new values atomically.

Many universal constructions resolve conflicts between transactions trying to update the same location by "helping" or "co-operating", a technique first introduced by Herlihy[41] to support wait-free transactions. Each transaction registers the operation it intends to perform. If it stalls, other transactions can complete its operation. As noted in their implementation of `CAS`n, Shavit and Touitou [87] demonstrated that performance of any helping algorithm is hurt by long chains of redundant helping, and can be improved by avoiding any recursive helping. Afek et al. [1] showed that performance can be further improved by explicitly inspecting the graph of waiters and constructing a helping schedule to minimize contention and avoiding unnecessary aborts.

Inspired by Herlihy and Moss's Transactional Memory, Shavit et al. [87] coined the term *Software Transactional Memory* (STM). Since their work, the term has been used almost interchangeably with universal construction, with a subtle connotational difference: STM algorithms are viewed as being executed dynamically, while universal constructions imply some amount of preprocessing. The two implementations are functionally equivalent.

The best of the current proposed STM implementations is by Moir [74]. Unwittingly, he reinvented an old operating systems idea of using "shadow pages" to implement atomic transactions.

The Juniper/XDFS file system [71, 95] allowed any process to group together an arbitrary collection of modifications as a single atomic action. Juniper made the updates atomic by simply changing the pointers to the modified data pages in the B-tree that mapped from file addresses to disk addresses. (In Juniper, the modifications were first stored in the log, and the atomic action was committed by atomically transferring the pointers to the pages from the log to the relevant files). According to Lampson [56], this technique was used first in the Cal system. The Cal system dates from Sturgis' Phd thesis [94] in 1974, so this rediscovered idea is almost a quarter-century old.

The basic outline of Moir's proposal is very similar. He implements an STM algorithm using local copies. He divides memory into blocks, references each block indirectly through a block table, operates on local copies of each block, and at commit time tries to use a version of `CAS`n (he uses the terminology `MWCAS`, for "Multi-Word CAS") to store pointers to all the modified blocks atomically back into the block table assuming no one has modified the originals in the meantime.

Blocks in Moir's approach are equivalent to pages, and the block table is a page table.

Moir's approach differs from the classical approach in his use of copying, (although efficient implementations of Copy On Write may eliminate the cost of copying). A more substantial difference is in the pointer switching operation. Early implementations of atomic transactions were optimistic, but not non-blocking, because the commit was done under control of a lock. Moir uses a non-blocking CASn so the entire transaction is non-blocking. Unfortunately, the implementation of CASn involves costs proportional to the size of the shared memory (fortunately, in this case it is "only" a small multiple of the size of the page tables).

2.7.3 Operating System Support

The universal constructions discussed in the previous section depend mainly on universal primitives. An alternative approach is to depend on OS support.

Johnson et al. [52] extended the basic optimistic implementation technique of restartable atomic sequences to support long and complex atomic operations, called *interruptible critical sections* (ICS). They include several enhancements including making performance predictable enough for use in real-time systems and moving the bulk of the sequence-specific work out of the kernel and into user code. Like restartable atomic sequences, the work on interruptible critical sections is only applicable on uniprocessors. Incidentally, although this work was not widely distributed in the NBS community, [52] appears to be the first published work on non-redundant helping (without naming it such); credit is usually given to Shavit and Touitou [87].

Allemany and Felten [2] extend Bershad's work on non-blocking primitives [17] to improve the performance of Herlihy's protocol. Their SOLO protocol uses OS-managed advisory locks to reduce contention on shared, non-blocking, data structures. The OS releases the lock when the lock-owner experiences a long delay. The program manages the advisory locks by incrementing a counter of active threads on entrance to a critical section, and decrementing on exit. Processes must wait until the count of active threads is below some threshold (generally 1) before being allowed to proceed. The OS decrements the counter while an active thread is switched out — delayed processes do not excessively delay other processes, because the decrement by the OS allows other processes to proceed.

Allemany and Felten also propose an alternative protocol, "SOLO with logging", that allows the current owner to operate directly on the shared object. Each transaction maintains a log with enough information to *undo* the modifications when the owner is preempted. The currently active process copies the data structure from a stalled owner, uses the log to undo all changes on the copy, and then atomically installs the copy in place of the original. (When the stalled owner eventually proceeds it modifies a now-uninstalled copy).

The SOLO protocol incurs the cost of copying. The "SOLO with logging" protocol is not strictly non-blocking — if each transaction is preempted just before completing, the new owner may spend time undoing the work of the first transaction, until it, too, is preempted just before completing.

Lamarca [54] proposed a modification to the SOLO protocols. Rather than having contending processes copy the object and undo the modifications, Lamarca proposes that the new process "help" the stalled owner finish. To avoid the cost of logging each change, Lamarca's SOLO-cooperative protocol uses the techniques of process migration to transport the stalled owner's computation to the waiter's process. When the owner's transaction is complete, the computation is migrated back. Such a scheme depends upon OS support for process migration and is expensive relative to a so-called "long delay". It seems unreasonable to introduce process migration to recover from a page fault, when it seems that the cost of process migration must surely involve at least one page fault. While the *cost* might be justified if the stalled process has actually failed, it is difficult to actually migrate a failed process (especially if the failure was due to OS failure on that node!).

The universal constructions discussed in this section support fully dynamic transactions. They have low overhead compared with constructions based on universal primitives. However, they are dependent on OS support, are not fully fault-tolerant, and don't support multi-object updates. Further drawbacks of schemes which depend upon OS and scheduler support are discussed in Section B.2.

2.8 Conclusion

There are published non-blocking algorithms for simple data structures that perform better than equivalent locking implementations even under the common case of no contention. Unfortunately these often are, literally, publishable results (i.e. complex enough to justify publishing a paper). Equally unfortunately, several critical data structures have no known efficient non-blocking implementations using only `CAS` or `LL/SC`. `CAS` and `LL/SC` are the only currently available non-blocking primitives on production hardware.

Non-blocking synchronization has been successful in real-time systems. This work depends, though, on properties of hard real-time systems that do not generally hold true in non-real-time systems. Further, these properties cannot be easily adopted in non-real-time systems, and therefore the work is not generalizable.

There is a large body of literature on universal transformations. It is easy to use universal transformations to create a non-blocking implementation of any data structure. Unfortunately, the average performance of such implementations is significantly worse than an equivalent locking implementation, rendering universal transformations impractical.

Previous work on non-blocking algorithms *seems* to pose a quandary for the programmer: either give up performance or give up comprehensibility. There is evidence, however, that this tradeoff is neither fundamental nor unavoidable.

Both the Cache Kernel and Synthesis multiprocessor operating system kernels exclusively used non-blocking synchronization for all their coordination needs. In both systems the implementers never needed to resort to universal transformations — all shared data structures were amenable to direct non-blocking implementations. Further, the direct implementations of non-blocking data-structures performed competitively (or better than) equivalent locking implementations *and* were not overly complex to understand, yet provided all the advantages of non-blocking synchronization.

One common element between the Synthesis and the Cache Kernel work was the availability of hardware `DCAS`. In addition to the Cache Kernel and Synthesis work reported, several papers proposed relatively efficient direct implementations of data structures which also depended upon `DCAS` functionality. No constant time implementation of `DCAS` exists based only on unary primitives. Unfortunately, `DCAS` is no longer supported on modern processors.

There have been a range of proposals for augmented universal primitives in both software and hardware. The minimalist proposals focus on support for particular universal constructions (e.g. `CCAS` and `C-Lock`), but do not support the range of direct implementations made possible by `DCAS`. The maximalist proposals aim to support non-blocking transactions almost entirely in hardware — but have been too complex to be treated seriously by hardware manufacturers.

Chapter 3

Universal Constructions

3.1 Introduction

Few data structures have straightforward and efficient non-blocking implementations using only unary CAS and currently available techniques. By "efficient" I mean that in the common case of no contention the cost is comparable to using locks, and that performance scales well under high contention. Thus far, any non-blocking implementation of a data structure has been considered a publishable result. Several important data structures (such as priority queues) still have no known *practical* non-blocking implementations.

The conceptual complexity of non-blocking algorithms can be overcome by using *universal translation* protocols (e.g. [39, 2, 99, 14, 54, 52, 4, 3, 87, 74, 1, 49, 50]) to convert mechanically from a well-understood sequential specification to a provably equivalent non-blocking algorithm[1]. Such protocols exist and can provide non-blocking implementations of *any* data-structure. Unfortunately, as noted by many (e.g. [39, 54, 87, 69]), the *average* performance of algorithms generated by applying these universal protocols is unacceptable compared to equivalent locking implementations.

Binary universal primitives, or universal primitives with arity 2, are universal primitives that operate on two *independent* locations. Two examples of binary primitives are DCAS and LLP/SCP (described in Chapter 5). In this chapter I show that binary universal primitives can ease the apparent tradeoff between conceptual complexity and performance. Given an $O(1)$ binary, universal,

[1]Universal translation protocols are also called "universal constructions" and "universal transformations". More recently, the term "software transactional memory" has been used to refer to run-time systems that support begin-transaction and commit-transaction operations, and treat all reads and writes nested within the begin and commit as a single, non-blocking, atomic transaction.

synchronization operator:

1. I prove that CASn on P processors can be implemented with worst case throughput of $O(1/n)$ (that is, in the worst case, some CASn completes successfully in $O(n)$ time regardless of the number of processes), and $O(np)$ space[2] (where $p < P$ is the number of processes *actively* contending at a given time).

2. I demonstrate two implementations of dynamic Software Transactional Memory [87, 74, 1, 43] (*STM*) that *dynamically* performs R reads and W writes, with $O(W)$ overhead (as opposed to $O((W + R) \log (W + R))$ [74]),

3. I show a fault-tolerant contention-reduction protocol that requires no special-purpose operating system support or program counter modification, and

4. I show how to modify these algorithms to support atomic multi-object updates (increasing the time and space complexity by $O(1)$ per object), be *disjoint-access-parallel* [47] and/or *wait-free*.

In each of these cases, a common structure emerges: unary primitives (e.g. CAS1 or LL/SC) are inefficient in either space or time or are not fault-tolerant — binary primitives are necessary to achieve each goal. Additionally, $O(1)$ primitives of arity M, $M \geq 3$, only provide marginal gains: for example, they may reduce the cost of the universal protocols by a constant factor, but cannot reduce the complexity of CASn below $O(n)$ — binary primitives are therefore sufficient to achieve each goal. The results of this chapter are part of the argument that binary primitives (e.g. CAS2 or LL/SC2) are the best operations to implement and provide as primitives of the underlying system. (Chapter 5 demonstrates that $O(1)$ binary primitives are realistic.)

I have found, in practice, that binary universal primitives ease the apparent tradeoff between conceptual complexity and performance in two ways. First, binary primitives support direct and efficient algorithms for many custom data structures. Thus we need depend on expensive universal techniques less frequently than with unary CAS. Second, as this chapter demonstrates, binary primitives improve the performance of universal constructions under a number of metrics, increasing the likelihood that a universal transformation will yield acceptable performance for any given application.

Direct, non-blocking, implementations of data structures are of critical importance if non-blocking algorithms are to be used in practice. In [34] and elsewhere in this thesis, I make the

[2]The version that is only effectively non-blocking needs only $O(n)$ space.

empirical argument that, in practice, for real systems, `DCAS` is powerful enough to *efficiently* implement non-blocking versions of *all* performance critical synchronized data-structures. In Appendix D I present data-structure-specific algorithms using `DCAS`. Some of these algorithms perform as well or better than any previously published synchronization algorithms (both blocking and non-blocking algorithms) for the same structures. In the average (and common) case, all perform comparably to locking implementations. Section D.1 argues briefly that the power of `DCAS` to improve these specific, simple, data-structures is at least as important as any general algorithmic advantages that `DCAS` might hold over `CAS`.

Performance improvements for a small set of common data structures, no matter how important, are not the whole story. The second advantage offered by binary primitives is the improved performance of universal constructions. Consequently, when one *must* resort to universal constructions, the performance is asymptotically optimal. That is, the asymptotic worst-case time complexity of the universal constructions instantiated with a particular algorithm is equivalent to the asymptotic worst-case time complexity of the original sequential algorithm. This is optimal because Jayanti [50] proves that an *oblivious universal construction* cannot improve over the worst-case time complexity of the original algorithm. Oblivious constructions are universal constructions that do not exploit the semantics of the particular sequential specification with which it is instantiated.

3.2 Universal Algorithms for non-blocking updates

Appendix B introduces a taxonomy that characterizes all universal constructions. There I argue (i) that algorithms that update-in-place are preferable to those that operate on local copies, (ii) that checking for transaction conflicts is preferable to depending upon active OS support, and (iii) that in the case of conflict, roll-back is a more attractive option than roll-forward.

However, by far the most common approach to universal constructions operates on local copies and uses roll-forward. The basic local copy construction allows a transaction to operate on local copies of parts of the shared data structure. Modifications are made to the local copies. At the end of the transaction all pointers to the original copies are replaced with pointers to the modified versions in a single atomic `CAS`n.

`CAS`n is an n location generalization of `CAS` and `DCAS`. Given n `locations`, n `oldvals`, and n `newvals`, `CAS`n checks whether *each* `location` contains the correct `oldval`. If so, it stores the appropriate `newval` in *every* location, and returns TRUE. If not, it has no effect and returns FALSE. The entire operation is atomic, so no intermediate state can be made visible.

First, I demonstrate a more efficient DCAS-based version of CASn. In addition to the improved time cost, my version of CASn has lower space requirements, which allows us to use smaller block sizes for equivalent space overhead, thereby significantly reducing copying costs.

Evaluating the implementation of CASn out of Double-Compare-and-Swap is interesting for another reason, given that we are interested in determining the *best* universal primitives (i.e. what arity primitives) to provide. Clearly, if we can efficiently implement CASn given a particular primitive CASk, then we do not need to provide a primitive implementation of CASn for $n > k$.

An alternative to the copying approach to universal constructions is to perform the updates directly on memory without resorting to local copies, and have contending processes *help* finish your update. This is equivalent to STM with roll-forward.

All published non-copying implementations of STM (with either roll-forward or roll-back), rely on active operating system support. The general approach is to "protect" a data-structure by using a lock, and to depend upon the operating system to prevent or recover from long delays by a lock holder. The precise definition of "protect" varies between proposals, from simply reducing contention to depending upon the lock to protect the integrity of the data structure.

My second algorithm is a DCAS-based implementation of STM with roll-forward. DCAS enables a purely application level approach (portable, assuming DCAS functionality is supported) that provides the same or better contention reduction and fault-tolerance properties as such schemes, yet it does not depend on the operating system support. The DCAS based approach is fault-tolerant across operating system failures on individual nodes, as long as the memory system survives. Software failures are reported to be ten times more likely than hardware failure[33], so this distinction is relevant.

The final algorithm I present is STM with roll-back. As a practical matter, roll-back is preferable to roll-forward, but such algorithms cannot be strictly non-blocking. Here too, as with the roll-forward algorithm, DCAS enables a purely application level approach.

All three algorithms described in this chapter share much code. I present them as parameterized specializations of a single body of code. For clarity of exposition, I first present preliminary versions (with limited functionality) of each protocol and then, gradually, add in features one at a time. I show how the protocols naturally support fault-tolerant contention-reduction and disjoint-access-parallel implementations of multi-object transactions.

```
  Log *trans_alloc(void *helper(),
                   void *hint)
  {
    log = newLog();
    log->helperCount = 1;
    log->helper      = helper;
    log->hint        = hint;
    return(log);
  }

  void trans_init(Log *log)
  {
    log->state.undo   = FALSE;
    log->state.finish = FALSE;
    log->state.idx    = 0;
0:  log->id = <my_machine_id,my_pid>
0:  log->orig_id = log->id;
  }

  int trans_begin(Log **laddr,Log *log)
  {
    if (log == NULL) { return(FALSE); }
1:  int *cnt = &(log->helperCount);
    do {
      if (*laddr != log)
        { return(FALSE);}
    } while (!DCAS(cnt,      laddr,
                   *cnt,     log,
                   (*cnt)+1, log));
    return(TRUE);
  }
```

```
  void trans_cleanup(Log *log)
  {
    Word state = log->state;
    int i, fin = state.finish;
    int idx    = state.idx;
    int undo   = state.undo;
    if (undo) {
      for (i=log->state.idx-1;i>=0;i--) {
        loc = log->locs[i];
        old = log->oldvals[i];
        DCAS(&(log->state), loc,
2:           <i,  fin,1>,   *loc,
             <i-1,fin,1>,   old);
      }
      idx = -1;
    }
3:  DCAS(log->domain, &(log->state),
         log,         <idx,fin,undo>,
         NULL,        <-1, fin,undo>);
  }

  /* Return log to local cache,
   * iff helperCount == 0 */
  void trans_conclude(log)
  {
    atomic_decr(&(log->helperCount));
    if (log->helperCount == 0)
4:    { deleteLog(log); }
  }
```

Figure 3.1: Basic code used to manage logs.

3.2.1 The basic non-blocking protocol using `DCAS`

3.2.1.1 Implementing `CAS`n from `Double-Compare-and-Swap`

A lock does not physically prevent another thread from accessing a data structure — we conventionally associate a particular lock with a particular data structure and agree not to reference the data structure without first acquiring that lock. We can similarly define an *owner* over a *domain*. We conventionally agree not to perform a `CAS`n over words in a given domain unless we are the current owner of that domain. A *domain* can be any collection of shared memory locations — a domain can be a block of memory addresses, a specific object, all data structures of a given type, the entire shared memory, or any other association with as fine a granularity as you care to choose. The only requirement is that the software knows how to map (either explicitly or implicitly) from a memory location to its unique domain (and therefore that domains be non-overlapping)[3].

Figures 3.1 through 3.3 display the basic algorithm used to implement `CAS`n. `CAS`n is non-blocking, and atomic with respect to other `CAS`n's over a given domain. I begin by considering a single domain, but I address multiple domains in Section 3.2.3.

The approach is straightforward. At the start of the `CAS`n we copy the arguments into a privately owned log in shared memory. A particular activation of `CAS`n becomes the current owner of the domain by atomically testing whether the domain has no owner, and if so, storing a pointer to the log in the domain. The arguments `(locs, oldvals, newvals)` are each ordered arrays of equal size, representing n `CAS`'s. The ith `CAS` is `CAS(locs[i], oldvals[i], newvals[i]);` As `CAS`n performs each individual `CAS` it records the state of the computation in the log. The state can be completely described by an index into the array of arguments, and a single bit specifying whether all the `CAS`'s have been successful so far. If an individual `CAS` fails because `*loc != oldval`, then by the definition of `CAS`n, `CAS`n must return a value of FALSE, and make sure that no locations are changed. `CAS`n marks `state.undo` in the log, and proceeds, in reverse, undoing the previous `CAS`'s. The flag marking failure and the index that tells what `CAS` the `CAS`n is up to are stored in a single word. Thus, each time `CAS`n performs a `CAS`, it can simultaneously check and update the state using `DCAS`. Now, if a second `CAS`n starts while a first is still in progress, the second detects the first log in the domain, and the second can proceed (after helping finish the first `CAS`n), rather than just waiting. Once the second `CAS`n helps, the first detects interference when its

[3]Domains must be non-overlapping at any given instant in time. However, this does not preclude a descriptor moving from one domain to another over time. This is similar to the well understood structure in blocking synchronization where a different lock protects a particular page frame while it is on a free list, another lock while it is used as a file buffer, and yet another lock when it is used as part of the address space of a process.

next DCAS fails because the state word was modified. Logs are never reused until all participating processes release it, so the state word unconditionally detects failure (the log cannot be recycled by another transaction, so the state word can never contain a pointer to an object stored at the same location as this log).

```
   int trans_open(Log **domain, Log *log)
   { /* Keep trying to acquire domain. */
     while (!(log->state.undo ||
              log->state.finish ||
              trans_start(domain, log))) {
       Log *owner = *domain;
1:     if (trans_begin(domain, owner)) {
         void *helper() = owner->helper;
         void *hint     = owner->hint;
         if (helper == NULL) { /* Abort */
           trans_abort(owner, FALSE);
           trans_cleanup(owner);
         } else { /* Help */
           *helper(domain, owner, hint);
         }
         trans_conclude(owner);
       }
     }
     return(*domain == log);
   }

   int trans_start(Log **domain, Log *log)
   { idx = log->state.idx;
     log->domain = domain;
2:   return((*domain == NULL) &&
            DCAS(&(log->state), domain,
                 <idx,0,0>,     NULL,
                 <idx,0,0>,     log));
   }
```

```
   int trans_commit(Log *log, int force)
   {
     Word state = log->state;
     if (!state.finish &&
         (force || !state.undo) &&
3:       CAS(&(log->state),
             state,
             state|FINISH))
       { state = log->state;  }
     trans_cleanup(log);
     return(state.finish);
   }

   void trans_abort(Log *log, int byOwner)
   { int flags = (byOwner?(UNDO|FINISH)
                         :UNDO);
     do {
      state = log->state;
     } while
4:      (!state.finish &&
         !CAS(&(log->state),
              state,
              state|flags));
   }
```

Figure 3.2: Basic code supporting atomic transactions using DCAS.

Note that when CASn returns “FALSE” here, it *is* still progress, because we refer to the return value computed by the CASn procedure, and not to whether the operation succeeded or failed to complete. Given the helping, *every* operation is guaranteed to complete, in the order of acquiring ownership of the domain, even if the initiating process is delayed or destroyed.

3.2.1.2 Transformation from locks to NBS: software transactional memory

A more useful abstraction, if it is implementable, is that of transactional memory [43, 87, 74]. Transactional memory provides the illusion of a large contiguous array of memory, containing all the data to be accessed by transactions. All reads and writes to the transactional memory between the beginning and the end of the transaction are considered to be part of a single atomic transaction, and either all the writes succeed, or they all fail. Transactions are atomic, execute in parallel if they

```
   int CASn (Log **domain, int n,
             void** locs, void** oldvals,
             void** newvals)
   {
     int retval;
     Log *log = trans_alloc(NULL,CASnInternal,NULL);
     trans_init(log);
     log->count = n;
     log->locs = locs;        /* copy these into */
     log->oldvals = oldvals;  /* shared memory   */
     log->newvals = newvals;  /* if needed       */
1:   trans_open(domain, log);
     CASnInternal(domain, log, TRUE); /* Do it */
     retval = !(log->state.undo);
     trans_conclude(log);
     return(retval);
   }


   void CASnInternal(Log **laddr, Log *log, void *hint)
   {
     do {
       Word state  = log->state;
       int  i      = state.idx;
       if (state.undo) {
         trans_cleanup(log); break;
2:     } else if ((i >= 0) && (i < n)) {
3:       if (!DCAS(&(log->state), locs   [i],
                   state,        oldvals[i],
                   <i+1,0,0>,    newvals[i])) {
           if (state.idx == i) /* not helped */
4:           { trans_abort(log, TRUE); }
         }
       } else { break; } /* CASn must be complete */
     }
5:   trans_commit(log, TRUE);
   }
```

Figure 3.3: Code for preliminary $\texttt{CAS}n$ implementation using DCAS.

```
acquireLock(d);
...


d->a = foo;
...
...
d->b = bar;
...
d->c = baz;
...

releaseLock(d);
```

```
Log *log = trans_alloc(NULL,NULL);
do {
  trans_init(log);
  if (!trans_open(&(d->log), log))
     { continue; }
  ...
  if (!trans_write(log, &(d->a), foo)) { continue; }
  ...
  if (!trans_write(log, &(d->b), bar))  { continue; }
  ...
  if (!trans_write(log, &(d->c), baz))  { continue; }
  ...
} while (!trans_commit(log, FALSE));
trans_conclude(log); /* return log to local cache */
```

Figure 3.4: Transforming code updating data structure d using locks to a non-blocking update.

do not conflict, and if they do, the conflicts are detected and resolved automatically.

Programmers are free to write arbitrary code — in particular they need not collect all memory locations to save for the final `CAS`n, nor do they need to cache local copies of modified locations. Transactional memory also supports early aborts, so that transactions that conflict and may need to abort and retry, can retry *before* investing many resources in the doomed transaction.

`DCAS` allows an efficient implementation of dynamic transactional memory, which has no costs associated with unused memory locations. Thus, unlike previous implementations, our implementation of STM can use all of shared memory for transactions with no extra overhead.

With the support in place for `CAS`n, extending the implementation to support STM is straightforward. Once again we map addresses to non-overlapping *domains*. Each domain conventionally controls access to the set of addresses mapped to it. Again, I first consider a world consisting of one domain. (Later, I address a case where we use individual data structures as domains over their components. At that point the code works equally well if we chose address ranges (including having each word be its own domain) or any other mapping.) Each transaction is bracketed by `trans_init` and `trans_conclude`, which associate a given log with this transaction. Within a transaction, each time you access a new domain, `trans_open` associates the domain with your transaction. The location of each write, and the previous value, are recorded in the log, so that the write can be undone if we need to abort or retry. At the end of each attempt, `trans_commit` is called by the owner. If successful, the transaction is commited and we exit. If unsuccessful `trans_cleanup` is called, which undoes any writes done by this transaction. If the owner `abort`ed the transaction, then we exit after cleanup, otherwise we retry.

If transaction T_1 tries to acquire ownership of a domain which is already owned by another transaction T_2, then T_1 can either help T_2 (in the case of STM with roll-forward) or undo T_2 (in the case of roll-back) and proceed itself. When T_2 tries to proceed it detects the state change and breaks out to the `commit`. In CASn T_1 is able to *help* T_2, and finish the transaction before acquiring ownership. We can now see that CASn was simply a special case of a transaction with a helper function. We improved performance in two ways. First, by folding `trans-write` into the helper function (`CASnInternal`). Second, by realizing that `oldvals` *already* held the old values for modified locations so we did not need to update the log at all. These optimizations are not possible for STM in general, where we have no idea of old (or new) values in advance.

The code for `trans_write` is shown in Figure 3.5 and the transformation from conventional code using locks to STM is illustrated in Figure 3.4. At (3.5:1) `DCAS` is used only to ensure that no new writes occur once a transaction is marked `undo`. This allows each transaction to detect an abort with no race conditions. There is no need for the OS to stop a transaction that has been aborted — the `DCAS` protecting each write ensures that no illegal writes occur.

```
  int trans_write(Log *log, Word *loc, Word val)
  {
    int index = log->state.idx;
    Word OldVal = *loc;
    log->locs[index] = loc;
    log->oldvals[index] = OldVal;
1:  return(DCAS(&(log->state), loc,
                <index,0,0>,   OldVal,
                <index+1,0,0>  val));
  }
```

Figure 3.5: Routine implementing software transactional memory.

Theorem 18 in Appendix C.1 proves that the implementation of CASn in Figure 3.3 is non-blocking. It also proves that the STM with roll-forward implementation presented so far, is non-blocking, assuming that the original sequential specification always executes in a finite number of steps.

The basic outline of the proof demonstrates that once a log is installed in a domain its transaction is guaranteed to complete in a finite number of steps. `state.idx` can only be modified a finite number of times; each loop is guaranteed to terminate after a finite number of iterations (usually 1); and most functions are called at most once by each process for each transaction.

The only subtle parts of the proof consist in showing (a) that logs cannot be reallocated while any references to them exist (Lemma 7), and that (b) a transaction can assume that a log is still installed in a domain as long as `log->state.idx` is non-negative, without reading `*domain` (Lemma 8). This allows us to use `DCAS` to simultaneously update a memory location, increment the index in the log, *and* be guaranteed that the transaction has not been aborted.

3.2.1.3 Description of the code

The algorithm in Figures 3.1 through 3.3 depends on auxiliary routines to manage the logs and acquire ownership of domains. These routines are also used by the STM algorithms, so they are more general than strictly needed by CASn.

Every transaction is preceded by a call to `trans_alloc`, which allocates a log for this transaction, and ends with `trans_conclude`, which manages deallocation of the log.

`trans_alloc` is called once for each transaction, regardless of how many attempts it takes to complete the transaction. `trans_alloc` allocates a log from a per-processor pool with a reference count of 1, stores a helper function and a hint (usually a pointer to extra state needed by the helper function) and returns the new log to the caller.

`trans_conclude` decrements the reference count. If the reference count reaches 0, then `trans_conclude` returns the log to the per-processor pool. It is called by any process that concludes using the log.

`trans_init` is called *each* time a process (re)tries to begin a transaction. `trans_init` resets the state variables (`finish`, `undo`, and `idx`). For CASn, or any algorithm that employs helping, `trans_init` is only called once. In such algorithms as soon as the first attempt is complete, either the originator has been successful, or some other process has "helped" the originator's transaction to complete. There is no second attempt.

`trans_begin` begins operating on `log` for some transaction. The operation might be "helping" the owner complete, or it might abort the transaction, or inspect the `log` and simply wait. In any of these cases, we must guarantee that `log` is not re-used for another transaction until the caller of `trans_begin` finishes. `trans_begin` atomically increments the reference count on the `log` that is currently stored as the owner of `domain`. `trans_begin` guarantees that at the moment the reference count is incremented, `log` was the current owner of domain. Every successful `trans_begin` must be matched by a call to `trans_conclude` on the same `log` because `trans_begin` increments the reference count on `log`. If `log` was uninstalled, then `trans_begin` returns FALSE without incrementing the reference count.

`trans_cleanup` checks whether `undo` is set. If so, it atomically restores all locations to their original values. In any case, it uninstalls `log` from `domain` but does *not* decrement the reference count. (Although the `log` is no longer attached to the domain, callers may still need to look at the result of an operation — so `log` cannot yet be reused.)

`trans_start` atomically sets `log` as owner of `domain` if and only if `domain` has no current owner and `log` is associated with a transaction that has not yet begun. It returns TRUE for success and FALSE for failure.

`trans_open` loops trying to install `log` as owner of `domain` by calling `trans_start`. It continues looping until it succeeds or `log->state` reports that this transaction completed or was aborted. `trans_open` returns TRUE if it succeeds, and FALSE if the operation/transaction associated with `log` finishes or aborts. If `trans_open` finds that `domain` is currently owned by a different log, then (bracketed by calls to `trans_begin` and `trans_conclude`) it either calls a helper function to finish the transaction for the owner, or, if no helper function exists, it aborts the owner. The goal is to free `domain` from any owner, so that `trans_open` may proceed on behalf of its caller.

`trans_abort` terminates a transaction associated with `log` regardless of whether it has completed or not. When called by the original owner of the transaction then the transaction is not retried. If called by a contending process, `undo` is set, but not `finish`, so the transaction starts all over again.

`trans_commit` returns TRUE if the transaction has committed successfully. If the transaction has not been previously aborted or finished (i.e `undo` has not been set), `trans_commit` first tries to set `finish`. If called with `force == TRUE`, then `commit` tries to set `finish` regardless of whether `undo` was set or not. `force` is used by CASn because we overload `undo` for CASn to also report a legitimate (i.e. successful) return value of "FALSE".

`CASn` calls `trans_init` passing `CASnInternal` as the helper function, so any process encountering this domain can complete the pending transaction. `CASn` copies its arguments into the log returned by `trans_init`. It then calls `trans_open` on the domain. `trans_open` does not return until this log is registered as the owner of this domain (which may involve helping other transactions finish). It then calls `CASnInternal` to do the work, and (after calling `trans_conclude`) returns whether the CASn was successful or not.

`CASnInternal` is the helper function which does the actual work. It loops, extracting `index` from `log->state`, checks to make sure `index` is within bounds, and performs a single `CAS` with the appropriate arguments. If any `CAS` fails, then `CASnInternal` sets `state->undo`. If `CASnInternal` detects `state->undo`, then it calls `trans_cleanup` and breaks from the loop. The last thing it does is call `trans_commit` with `force == TRUE`. It returns the value of `trans_commit`.

3.2.2 Refinements to the preliminary versions: contention reduction

It is well known [17, 39, 2, 54] that the performance of naive non-blocking algorithms can be hurt due to "useless parallelism". First, if p processes simultaneously try to operate on a single data structure, $p-1$ of the processes will waste processor cycles and accomplish nothing — all modifications will be undone at the end when each transaction detects that some other process succeeded. Second, the work is not only wasted, but the cache and bus interference due to the $p-1$ failing processes will also increase the execution time of the one process that ultimately succeeds. This performance degradation due to contention has been confirmed by measurement [39, 2].

A good way to reduce contention is to treat ownership of a domain by a transaction as an advisory lock (as in [17, 2, 54]) and only allow other processes to acquire ownership (by undoing or helping) if the original owner is already subject to a long delay (preemption, failure, etc.). Advisory locks reduce contention for both the CASn and STM with roll-forward algorithms. For STM with roll-back advisory locks serve another, additional, purpose. They not only reduce contention, but are necessary to make STM with roll-back effectively non-blocking.

In the initial algorithm I presented, STM with roll-back is *not* non-blocking. The non-blocking proof of the other algorithms depended upon the existence of helper functions. We can always carefully write a helper function for *any* given algorithm or data structure. However, the point of STM was to avoid requiring programmers to write complicated algorithms and, instead, to automatically convert arbitrary sequential code to an equivalent non-blocking algorithm. If we are to update in place, and not depend upon OS assistance, then the only way to avoid forcing programmers to write helper functions is by using roll-back.

In Section B.3 I point out that roll-back can *never* be strictly non-blocking, since two competing transactions can continually abort each other, forcing partially completed transactions to be undone, and then no progress is made. I defined the weaker property of effectively non-blocking for algorithms that used roll-back. However, without the contention reduction code, STM with roll-back is not effectively non-blocking either. It is not strictly non-blocking even under a weakened adversary, and neither is it non-blocking with probability 1 in all cases.

The contention reduction code has the property that once a transaction acquires ownership of a domain it is guaranteed to finish without being aborted, unless it suffers a long delay and is preempted. The guarantee of no aborts while running ensures progress under the weaker adversary. The abort of, or transfer of ownership from, preempted processes during contention ensures that the

```
   int trans_open(Log **domain, Log *log)
   { /* Keep trying to acquire domain. */
     ID myId = <my_machine_id,my_pid>;
     Log *owner = (Log *)NULL;
     while (!(log->state.undo ||
              log->state.finish ||
              trans_start(domain, log) ||
1:            !(owner = *domain))) {
       ID id = owner->id;
       Machine machine = id.machine_id;
2:     if ((machine != my_machine_id) &&
3:         (currentProcess(machine)==id.pid)) {
4:        { backoffIfNeeded(); }
       } else {
5:       if (DCAS(domain, &(owner->id),
                  owner,  id,
                  owner,  myId) &&
             trans_begin(domain, owner)) {
           void *helper() = owner->helper;
           void *hint     = owner->hint;
6:         if (*domain == owner) {
             if (helper == NULL) { /* Abort */
               trans_abort(owner, FALSE);
               trans_cleanup(owner);
             } else { /* Help */
               *helper(domain, owner, hint);
             }
           }
           trans_conclude(owner);
         }
       }
     }
     return(*domain == log);
   }
```

Figure 3.6: Enhanced `trans_open` to support contention reduction.

algorithm is interference-free. The combination is an effectively non-blocking algorithm.

3.2.2.1 Description of the modifications

The implementation consists of a small change to `trans_open` and `trans_begin`, and a simple (but lengthy) change to `trans_conclude`, as shown in Figures 3.6 and 3.7. Two other minor independent changes also help, as described in the annotations on lines 3.1:2 and 3.3:3.

When `trans_open` encounters a domain which already has an owner, it checks whether that owner is the currently running process on the owner's machine[4]. If it *is* the current process, then the new transaction waits and backs off, because it is reasonable to assume that the original owner is making progress. If the owner is *not* currently running, then it must be suffering a "long delay", and the new transaction takes over ownership, and proceeds. For STM, or any universal update with roll-back, "proceeding" means aborting the first transaction, forcing a retry. For CASn or any universal update with roll-forward, "proceeding" means the new transaction "helps" the first transaction finish.

`trans_begin` now checks whether the caller's process ID is listed as the current process ID of `log`. If not, then the caller has been preempted between acquiring ownership of the log and executing the body of `trans_begin`. Some other process has already taken over. It is safe to check the `log->ID` even though the increment hasn't occured yet because of TSM. The increment of the reference count is guarded, through `DCAS`, by `log->ID == myID`, so we are guaranteed that the reference count is only incremented by a process that is the current owner of the log.

Finally, `trans_conclude` is changed to give priority to the current owner of the log over waiters when it attempts to decrement the reference count.

The changes to `trans_begin` and `trans_conclude` ensure that the worst case throughput to complete transactions is independent of the number of contending processes. This is achieved by limiting access to the reference count on the log. `atomic_decr` must be executed once for each process that increments `helperCount`. One might imagine this would be $O(p)$, because in the worst case, all p actively contending processes might try to decrement the reference count at the same time. However, `trans_begin` is called in `trans_open` only if the current owner has been descheduled already and another process has not yet taken over ownership. The *transaction* is only delayed once for each preemption because only one process can succeed at taking over ownership.

[4] We can directly use the OS supported approach of Allemany and Felten instead, where the OS on the owner's node is required to release the lock if the owner is preempted. However, as I describe in Section 4.6.4, this reduces fault tolerance, introduces complexity, has worse performance, and only works on systems which provide the needed OS hooks.

The other processes just give up (since the (new) owner is not NULL, and it *is* the current process on its own machine). Therefore precisely one process succeeds at the increment in $O(1)$, and $p-1$ processes fail to increment in $O(1)$. The behavior of the decrements is slightly more complicated, and is described more fully in the proof in Appendix C.2.

`trans_begin` is called at most once for each preemption. Preemptions occur (roughly) independent of the number of contending processes, because preemption is a local decision made by the local scheduler, either due to quantum expiration or I/O or the process yielding the processor to wait for some other computation. If we assume that a process executes at least one statement after each preemption (else *no* algorithm is non-blocking), then the number of preemptions is bounded. There are a fixed number of statements executed by this transaction (we are only concerned with statements by current owners), and preemptions do not add loops. So the number of preemptions is very roughly bounded by $O(r+w)$.

The changes to `trans_open` reduce contention by the non-owning processes. Waiters spin on a local cache line that is only invalidated when there is a change of state that one of the waiters must act on. Thus, the local spinning produces minimal bus or memory traffic. The check for NULL at (3.2:2) ensures that `trans_start` can fail without invalidating the owner's cache line. The various tests in Figure 3.6 in annotations 2 through 4 are all read-only. Only after the owner is preempted do the waiters attempt any modification and engage in the protocol for acquiring (stealing, actually) ownership.

`backoffIfNeeded` at (3.6:4) chooses a uniformly distributed wait-time from an exponentially growing interval. The test of current process at (3.6:3) in conjunction with the advisory locking assures that this algorithm is not unfair to longer running computations. Without this code, transactions that run longer than a quantum run the risk of starvation due to many shorter running transactions.

Theorem 24 in Appendix C.2 proves that with the contention reduction code in place, the implementation of `CASn` in Figure 3.3 has worst-case time $O(n)$.

The proof consists of showing that `CASnInternal` succeeds in $O(n)$ steps, and that the increment and decrement of the reference count each take $O(1)$. The fact that atomic increment and decrement complete in $O(1)$ is non-obvious, and depends upon our use of `DCAS`. Increment only succeeds for the process currently owning the log; therefore, failing on increment implies that it is safe to immediately return without looping. A process does not have the luxury of giving up on a failing decrement (or else there will be storage leaks in the management of the logs). However, the current owner is given priority. All other waiters defer — they must wait for the owner in any case.

```
   int trans_begin(Log **laddr, Log *log)
   {
     if (log == NULL) { return(FALSE); }
     int *cnt = &(log->helperCount);
     do {
1:     if (log->id != myId){ return(FALSE);}
     } while (!DCAS(cnt,      &(log->id),
                   *cnt,      myId,
                   (*cnt)+1, myId));
     return(TRUE);
   }

   /* Return log to local cache,
    * iff helperCount == 0 */
   void trans_conclude(log)
   {
     int *cnt = &(log->helperCount);
*:   state = log->state;
2:   if (!(DCAS(cnt,      &(log->id),
*:              *cnt,      myId,
*:              (*cnt)-1, NULL))) {
       /* If we entered here it means we were
        * delayed at some point and no longer
        * own the log. */
*:     do {
*:      id = log->id;
*:      state = log->state;
3:      if (!id ||
*:          currentProcess(id.machine_id) == id.pid)
*:        { id = NULL; }
*:     } while (!(DCAS(cnt,     &(log->id),
*:                     *cnt,    id,
*:                     (*cnt)-1,id) ||
*:                (!(state.finish | state.undo) &&
4:                 DCAS(cnt,      &(log->state),
*:                      *cnt,     state,
*:                      (*cnt)-1, state))));
*:   }
     if (log->helperCount == 0)
       { deleteLog(log); }
   }
```

Figure 3.7: Enhanced `trans_conclude` and `trans_begin` to support contention reduction.

The owner exits in $O(1)$ steps. One waiter (we don't care which!) can exit immediately afterwards and can then immediately acquire ownership of the domain and proceed. There may be a process that is delayed $O(p)$ steps while decrementing the reference count, but the delays only occur while another process is free to make progress.

3.2.2.2 STM with roll-back is effectively non-blocking

With contention reduction in place, even a strong adversary who is able to inspect the state of the dynamic computation, force preemptions after each instruction, and schedule processes can only hinder progress probabilistically[5]. Given a maximum bound N on the number of processes (as is true in all real systems), once we reach the point that all N processes are actively contending on this domain, no new processes begin contending. The exponential backoff means that, after N processes are already contending, each time we fail the odds of there being a competing process ready to run are halved. In order for the adversary to successfully prevent a transaction TX_0 from completing, at least one other waiting process must awaken from a backoff during the execution of TX_0. For a given backoff interval B (the minimum interval of all N processes) the probability P of at least 1 out of the $N-1$ processes waking up during the execution time X (including the time stalled due to preemption) of TX_0, is $O(1-\left(1-\frac{X}{B}\right)^{N-1})$. As B gets larger this is approximately $O((N-1)\frac{X}{B})$. Doubling B then halves the probability of a process being available during preemption to abort TX_0.

Intuitively, this means that even in the face of infinite failures/preemptions there is only an infinitesmal probability of the algorithm running infinitely long with no progress. (With only a finite number of preemptions the algorithm is strictly non-blocking). In the face of infinitely recurring preemptions our algorithm will fail to make progress with probability P in $O(1/P)$ steps — meaning if we are willing to wait twice as long, the probability of no progress will halve.

3.2.2.3 How do we know whether a process is currently running?

The stated algorithm assumes the existence of an oracle that tells us whether a given process (the process currently owning a log, and hence owning the corresponding domains) is currently running. In practice, how would such an oracle be implemented?

Allemany and Felten [2] propose using active operating system support. When a given process

[5] The adversary cannot schedule a process if it is not runnable. Nor can it delay all runnable processes on the machine (or all algorithms trivially fail to be non-blocking).

becomes the owner of a data structure it acquires a lock on that data structure. It registers both the lock and its own process ID with the operating system. If the operating system ever preempts the process, the OS simultaneously releases the lock. Thus, waiters on the lock can know that if a lock is held the process is running. If a data structure is free, the owning process is not currently running.

This approach is adequate for our needs. However, it is not ideal for several reasons. First, if the operating system on the owner's node fails, the lock is never freed. Second, the operating system must treat any asynchronous handler as a context switch, or else not allow synchronization between main line code and a signal handler.

An alternative is to have the waiter rather than the owner make the determination. The owner simply stores its process ID and its machine ID in the log. On every process switch, the OS writes `currentProcess` to an array indexed by `machineId`.

A waiter can now determine whether the owner is currentProcess by a two stage test. First, the waiter compares the stored machine ID with the current machine ID. If they are equal, the waiter proceeds because it has clearly preempted the owner. If they are running on separate machines, the waiter simply compares `processID` to `currentProcess[machineID]`. (I assume that machine failures are detected by timeouts, and `currentProcess[deadMachine]` would be cleared appropriately.)

A third approach is possible. Assume a system that has loosely synchronized clocks. Each process writes its start time into the log when it acquires ownership and clears the field when it relinquishes ownership. In the event that the owner is delayed, processes can detect that the timestamp is older than T_{update}, and assume the process is no longer current. If the timestamp is more recent, the waiter backs off rather than acquires ownership. This avoids some (but possibly not all) cases of aborting running transactions. It also only probabilistically detects the long delays due to preemptions. This improves the probability of making progress, but is not sufficient to make the algorithm effectively non-blocking, because there is some possibility of making no progress even if there are only a finite number of preemptions due to misinterpreting a long running transaction as a stalled transaction and continually aborting it.

Section 4.6.4 discusses the measured performance of these three alternatives.

3.2.2.4 Why use both exponential backoff and currentProcess?

We use two *different* mechanisms to reduce contention: the `currentProcess` oracle and exponential backoff. The `currentProcess` oracle ensures that the algorithm is strictly non-blocking in the case of a finite number of preemptions. Exponential backoff allows the system to make progress (probabilistically) even if every statement gets preempted (more practically, it allows transactions that are longer than a scheduling quantum to make progress even under heavy contention.) Both are required to make STM with roll-back effectively non-blocking. Neither is necessary to make $\mathtt{CAS}n$ or STM with roll-forward strictly non-blocking.

The combination of the two techniques can also aid performance and fairness. The contention-reduction algorithm depends on `currentProcess` to avoid unfairness to longer running transactions. (In conventional systems that use only exponential backoff to reduce contention the odds of preempting a long-running transaction are higher than the odds of preempting a short one.) It also improves average throughput (`currentProcess` avoids waste where we might abort a transaction that is still making progress). The contention-reduction algorithm depends on backoff to reduce memory contention and synchronization contention. Using only `currentProcess` all competing processes will spin on the `owner` field, and possibly introduce contention over `helperCount`.

3.2.2.5 Summary of STM with contention-reduction

The STM algorithm, as augmented with the contention reduction code, implements an $O(n)$ $\mathtt{CAS}n$, allows STM to execute with $O(W)$ overhead, and ensures that STM with roll-back is effectively non-blocking.

However, the discussion so far has ignored transactions over multiple objects. The code, as written, does not immediately support multi-object updates. If we treat the entire shared memory as a single domain then we can support atomic updates that modify any combination of data structures. However, this comes at the cost of creating a bottleneck — only one update can occur *anywhere* in memory at a given time. Updates to two totally unrelated data structures conflict and only one can proceed. If we treat memory as a set of disjoint domains then updates to two or more disjoint domains proceed in parallel.

However, we have not shown how to implement atomic transactions that access *multiple* domains, some of which *may* conflict. We would like to support multi-object updates — *atomic* transactions that access disjoint sets of objects should proceed in parallel without interference. Atomically moving an entry from one list to another is an example of a multi-object update.

```
   int trans_open(Log **domain, Log *log)
   { /* Keep trying to acquire domain. */
     ID myId = <my_machine_id,my_pid>;
     Log *owner = (Log *)NULL;
     while (!(log->state.undo ||
              log->state.finish ||
              trans_start(domain, log) ||
              !(owner = *domain))) {
       ID id = owner->id;
       Machine machine = id.machine_id;
*:     int o_p = owner->precedence;
*:     int p = log->precedence;
1:     if (o_p == 0) { setPrecedence(log); }
       if ((machine != my_machine_id) &&
           (currentProcess(machine)==id.pid) &&
*:         ((o_p == 0) || (p <= o_p))) {
         backoffIfNeeded();
       } else {
         if (DCAS(domain, &(owner->id),
                  owner,  id,
                  owner,  myId) &&
             trans_begin(domain, owner)) {
           void *helper() = owner->helper;
           void *hint     = owner->hint;
           if (*domain == owner) {
*:           if ((o_p < p) ||  /* one != 0 */
2:               (myId != log->orig_id)) {
3:             trans_abort(owner, FALSE);
*:           }
             if (helper == NULL) { /* Abort */
               trans_abort(owner, FALSE);
               trans_cleanup(owner);
             } else { /* Help */
               *helper(domain, owner, hint);
             }
           }
           trans_conclude(owner);
         }
       }
     }
     return(*domain == log);
   }
```

```
   int trans_start(Log **domain,
                   Log  *log)
*:{ if (*domain == log)
      { return(TRUE); }
*:  int idx = log->dstate.idx;
4:  log->domains[idx+1] = domain;
*:  CAS(&(log->dstate),
        <idx,0,0>,
        <idx+1,0,0>);
    idx = log->state.idx;
    return
      ((*domain == NULL) &&
       DCAS(&(log->state), domain,
            <idx,0,0>,     NULL,
            <idx,0,0>,     log));
   }
```

Figure 3.8: Routines implementing software transactional memory. The lines with "*"s or numbers are new additions supporting multi-objects. (This code is continued in the following figure).

```
  Log *trans_init(void *helper(),
                  void *hint)
  {
    log->state.undo = FALSE;
    log->state.finish = FALSE;
    log->state.idx = 0;
    log->id = <my_machine_id,my_pid>
    log->orig_id = log->id;
*:  log->dstate = 0;
    return(log);
  }
```

```
   trans_cleanup(Log *log)
   {
     Word state = log->state;
     int i, fin = state.finish;
     int idx = state.idx, undo = state.undo;
     if (undo) {
       for (i=log->state.idx-1;i>=0;i--) {
         loc = log->locs[i];
         old = log->oldvals[i];
         DCAS(&(log->state), loc,
              <i,  fin,1>,   *loc,
              <i-1,fin,1>,   old);
       }
       idx = -1;
     }
     i = log->dstate.idx;
*:   DCAS(&(log->state), &(log->dstate),
*:        <idx, fin, undo>,  <i, 0, 0>,
*:        <-1,  fin, undo>,  <i, 0, 1>);
*:   for (i=log->dstate.idx;i>=0;i--) {
5:     DCAS(log->domains[i], &(log->dstate),
*:          log,              <i,0,1>,
*:          NULL,             <i-1,0,1>);
*:   }
   }
```

Figure 3.9: Multi-object enhancements (continued) to routines implementing software transactional memory.

3.2.3 Multi-object (multi-domain) updates

The transaction mechanism can be extended to support atomic multi-object transactions — the code in Figures 3.8 through 3.10 implement multi-object updates. Once again, the approach is straightforward. We associate a single log with *all* the domains accessed by a transaction. Each log is extended to have an array in which we store “owned” domains. In order to complete successfully, a transaction must retain ownership over any domain it acquired until the end of the entire transaction, at which point it may relinquish all owned domains. If ownership of a single domain is lost during a transaction, the entire transaction must be restarted. (Strictly speaking, you need only restart from the point it acquired ownership of that domain. Practically, though, it is painful to reset the PC (tricks with exception handlers may be plausible, but are not explored in this thesis.)).

Briefly, atomicity of multi-object transactions is ensured by having a single log shared between all objects in the transaction. Transactions over disjoint domains do not directly interact, let alone interfere with each other, unless a transaction tries to access a domain already owned by another transaction. When a potential circularity arises, each transaction is assigned an integer precedence number, which is used to arbitrate between multiple transactions competing for a domain. When

```
  void CASnInternal(Log **laddr, Log *log, void *hint)
  {
    do {
      Word state  = log->state;
      int  i      = state.idx;
      if (state.undo) {
        trans_cleanup(log); break;
      } else if ((i >= 0) && (i < n)) {
1:      if (!(trans_open(addr2domain(locs[i]),
*:                       log) &&
              DCAS(&(log->state), locs   [i],
                   state,         oldvals[i],
                   <i+1,0,0>,     newvals[i]))) {
          if (state.idx == i) /* not helped */
            { trans_abort(log, TRUE); }
        }
      } else { break; }
    }
    trans_commit(log, TRUE);
  }
```

Figure 3.10: Code changes for multi-object `CAS`n.

interference occurs on *any* domain D, the preemptor has access to the owner's entire original transaction through the owner's log stored in D. The preemptor can use the log to either help or abort the owner. If the log's transaction is complete (either committed or aborted), other transactions can unilaterally remove the log from each domain it owns.

Appendix C.3 shows in more detail that multi-object updates are still non-blocking, are deadlock and livelock-free, and are linearizable.

The key observation used in Section C.3 is that the explicit use of precedence in `trans_open` in Figure 3.8 allows us to avoid cycles and guarantee progress (even with the policy of aborting some transactions that conflict on a domain). Additionally, as a simple performance optimization, I avoid needless aborts by waiting, rather than aborting, if circular dependencies are impossible. In my STM algorithm, domains are data structures, and therefore long chains of conflicting transactions are highly unlikely – unless real dependencies exist between the data structures, in which case it will be difficult to extract parallelism. Applying sophisticated optimizations that explicitly inspect the contention graph, such as Afek's [1], are possible, but are unlikely to result in useful benefits. However, I have not measured this tradeoff in any real systems.

3.2.4 Some notes on asymptotic performance

As noted, the `CAS`n operation is non-blocking since progress (return of some `CAS`n, whether it modifies values or just returns FALSE) is guaranteed to be made (independent of the number, p, of processes) in at most every $2n$ steps. This is easiest to see in Figure 3.3. Once a transaction has begun, the current process proceeds without interference unless it is locally preempted. The $O(1)$ leader election occurs in parallel with installing the new owner in the log (note that the new owner helps finish the original `CAS`n, it does not originate a new transaction — changes to `my_id` do *not* cause an abort). The number of local preemptions is itself bounded (under the assumption that any reasonable scheduler allows the application code to execute at least one non-OS operation between preemptions). The worst case can occur only if we assume that the local scheduler preempts the current process after executing only one local step, and there is always a process waiting to run. Even so, the number of transfers of ownership is itself at most $2n$. (Bounding the number of preemptions does not similarly help algorithms which have loops that depend upon the number of processes).

For the same reason, sequential work (total number of steps when only one process is running) is $O(n)$. Similarly, the worst case time to perform r `CAS`n's is $O(rn)$, independent of p, and the worst-case work (sum of total steps for all processes) for p processes performing a sequence of r `CAS`n's is $O(npr)$, because each of the p processes can simultaneously attempt to acquire ownership of a stalled transaction, although only one succeeds.

As written, `CAS`n requires $O(np)$ space, where p is the number of processes *actively* involved in simultaneous `CAS`n's, because each process must keep all n arguments available to potential helpers. (The space per transaction is still $O(n)$). In implementations that are only effectively non-blocking, we can also use only $O(n)$ space for an arbitrary number of parallel transactions, because we can defer storing the arguments until *after* a transaction acquires ownership of the first domain. This deferral is not possible for strictly non-blocking implementations. The arguments must be accessible before acquiring ownership, else if every transaction is preempted just after acquiring ownership of the first domain but before copying its arguments, then no progress is made, as no helping is possible. The only additional space overhead is the `domain`, which is constant overhead and independent of the number, M, of words of memory, or the number, p, of actively contending processes.

Without preemption and cleanup, the STM algorithm's synchronization cost is $O(W)$, for W writes. Note that two processes accessing different domains/data-structures do not contend, and therefore can execute concurrently. Further, as noted in the text, the algorithm supports dynamic access to these multi-objects. If two transactions access disjoint sets of objects, there is no contention and both proceed in parallel. If they conflict, the dynamic assignment of bounded precedence numbers guarantees that the transaction with the highest precedence always makes progress. By deferring assignment of precedence until conflict, and treating transactions with no precedence specially, we are able to simply delay the lower precedence transaction rather than aborting it, unless there is a possibility of circular conflicts.

Multi-object transactions complicate worst-case performance slightly. They improve concurrency — note that by introducing domains of average size K, we have potentially reduced conflicts between transactions by a factor of K. (If the entire memory were a single domain, *all* transactions would conflict. By using domains of average size K and supporting multi-objects, transactions that access disjoint blocks of size K can proceed in parallel without interfering with each other.) However, multi-object transactions can hurt worst-case time. If p processes now transitively conflict, the last word of each CASn can conflict with the first word of another. If this relationship is completely circular, then we do not make progress until we have gone through all n steps of every other active transaction, aborting $p-2$ transactions, and only completing 2. Appendix C.3 describes the approach used to limit recursive helping. Fortunately, the code that limits recursive helping to one level, also limits the worst case behavior of CASn. Assume the worst case — which is when each word is in a different domain[6]. If we limit recursive helping to one level then each of the n words can trigger at most one helper. If each helper will not recurse, they will each update at most n words (assuming all transactions update roughly n words), for a total cost of $O(n^2)$. [87, 74] report, empirically, that a similar approach to non-redundant helping improved the performance of their algorithms, but do not describe the performance improvement in any detail.

3.2.5 Extra enhancements to the basic algorithms

The final versions of the basic algorithms presented in the preceding sections are non-blocking, support multi-objects, and provide a contention reduction mechanism that stabilizes performance under high load. There is some interest in the literature in enhancing non-blocking algorithms to have some additional, stricter, properties. Two common properties are "true disjoint-access-parallelism" [47] and wait-freedom. Both these properties have questionable utility when carried to the extreme. However, if required, the algorithms presented here can be extended to support both

[6] A slightly more accurate description of this case notes that if there are M/K domains in a memory of size M, then the n words can be in at most M/K domains. Similarly, if the transaction addresses only d domains, and $d < M/K$, then the n words can be in at most d domains (d is almost always only 1 or 2). Finally, if there are only p transactions actively contending at a given time, then this transaction clearly can help at most p others (p is usually small, because contention is low). Therefore, a better estimate of the cost is $O(n \min(n, d, p, M/K))$, where d and p are very small.

disjoint-access-parallelism and wait-freedom.

3.2.5.1 Disjoint-Access Parallelism

Disjoint-access-parallel algorithms allow two operations that access disjoint locations to proceed in parallel without interference. The multi-object algorithms presented here are disjoint-access-parallel to the granularity of domains — operations that access disjoint domains can proceed in parallel.

We can use the granularity of the domains to trade-off parallelism against overhead. By reducing the size of domains we increase the potential parallelism and the space and time overhead. By enlarging the average domain size, we reduce the parallelism and correspondingly reduce the overhead. If we define each word in memory to be a separate domain, then the algorithm is disjoint-access-parallel in the conventional sense of the term. At the other extreme, where all of memory is a single domain, no operations proceed in parallel.

Limiting the interference between transactions is a laudable goal, but it is not always preferable to try to increase parallelism, even if there were no overhead: increasing potential parallelism does not necessarily increase actual concurrency. There are already other limits to concurrency. Writes to the same cache lines are serialized regardless of the application because the writing processor must own the cache line at the time of writing. Contention reduction protocols serialize access to larger data structures.

There are also memory contention costs incurred by parallelism beyond serialization. There is a substantial cost to a cache miss (communication cost, plus the cost of stalling the pipeline and executing cache-miss-handling code). This cost (call it X instruction times) can be hundreds or thousands of times the cost of a single instruction. Two processes executing in parallel only gain by parallel execution if the parallelism on average causes fewer than $1/X$ cache misses per instruction. For fine granularity transactions, X is apt to be greater than the average run-time of a single transaction. Therefore, avoiding memory contention with the *currently* running transaction on the competing processor is not sufficient. To justify the parallelism we would need to avoid contending with future transactions running on the other processor. This implies some knowledge of future transactions. If transactions run on processors oblivious to the transaction's memory reference pattern, then there is no way that we can make any predictions about future contention, and therefore no way to evaluate whether parallelism is justified.

Fortunately, programmers and system designers are already roughly aware of the cost of memory contention, and most algorithms are already constructed in a way that enhances locality. In particular, programmers try to avoid hotspots — they detect memory locations that are sources of memory contention and recode algorithms to reduce or eliminate them. The programmer often has a good notion of locality that is larger than a single cache line. The programmer designs the algorithm to run with a certain set of cache lines remaining at one processor for long enough to amortize the cost of the cache-miss and transfer. It is not clear that every transaction will access all of these cache lines, but over a period of time a set of transactions on this processor will access the set of cache lines with high probability, while transactions running on other processors will (with high probability) not access any of them.

These observations have two implications. First, there is no benefit to paying a cost for disjoint-access parallelism *within* a set of cache lines that is inherently serialized. Any effort spent on allowing parallel access to a *subset* of these cache lines is wasted. Second, this set of cache lines is algorithm specific. If we *a priori* divide up memory into units of granularity for disjoint-access parallelism, we can, at best, only probabilistically match the algorithm's unit of concurrency.

`domains` allow the programmer to specify the unit of concurrency. Programmers using contention reduction techniques presumably take memory contention into account when deciding upon the appropriate granularity. Decomposition of memory into domains is tightly coupled with the decision to decompose memory into separate units for the purpose of contention reduction.

Although I argue for relatively large domains in practice (i.e. each data-structure or collection is its own domain), I should emphasize that, assuming support for operations that are atomic over multiple domains (i.e. multi-object atomic operations), then the finer the granularity of domains, the more concurrency is supported by our algorithms. In particular, assuming the concurrency is justified, if one chooses single word domains then the algorithms presented here *are* strictly disjoint-access-parallel.

The important thing to note is that `CAS1` algorithms cannot flexibly take advantage of reduced concurrency: they either pay in space per word, or by copying chunks of size K. While increasing K reduces the space cost for unused locations in memory, it increases the cost in time (due to copying larger blocks). `CAS2` (`DCAS`) allows increasing the domain size without any copying (thus reducing the space overhead).

3.2.5.2 Wait freedom

The versions of `CAS`n and STM presented here are not wait-free: there is no guarantee that a given process will ever exit the `while` loop in `trans_open`.

In practical terms this is not terrible. There are several reasons it is better to have the OS or scheduler deal with starvation and fairness than to address these problems by using wait-free algorithms. How do we know that all participants should have equal share? We do not — it is the job of the system scheduler to match system resources to the user-specified priority for each task. Consider a process with a very low relative process priority. We do not want such a process to get a turn if higher priority processes are waiting. If the low priority process is not getting a fair share, then the scheduler eventually raises its priority. It is a mistake to reimplement this same policy in every data-structure (it is even worse to implement a *different* policy!). The key property we want is non-interference, so that synchronization does not interfere with the scheduler. Non-blocking algorithms already provide the non-interference properties we need.

However, suppose we wanted to implement wait-freedom directly in our algorithms rather than depending upon the underlying system to avoid starvation. The wait-free algorithm of Moir in [74] can be adapted to our algorithm. His algorithm requires a "conditionally wait-free" `CAS`n — that it can be terminated if the algorithm has been "helped" or aborted at a higher level. Our implementation of `CAS`n is already conditionally wait-free. The client must set `log->state.undo` or `log->state.finish` to `TRUE` when it wants the transaction to terminate (which can be achieved by calling `trans_abort`). `trans_open` checks for this each time through the loop.

Following Moir, wait-freedom is implemented by means of two arrays. An `Announce` array is p elements long and contains the operation and arguments for each of p processes. When process P_i wishes to perform an operation it posts the operation and argument in `Announce[i]`. If it detects a conflict with a process P_j that already owns a domain D, P_i requests help from P_j by inserting a request for help into `Help[i,j]`. `Help` is a 2-dimensional $p \times p$ element array. Each P_k slowly cycles through the kth column of `Help`. It inspects one entry before each transaction attempt of its own. After p transactions P_k is guaranteed to have checked whether *all* other processes need help. If it finds a transaction that needs help, it helps that other process before attempting its own transaction. The actual transaction attempt follows the protocol described in this chapter, rather than the copying approach of Moir.

3.3 Related Work

This chapter presents efficient non-blocking, multi-object, implementations of dynamic transactional-memory, implemented in software with no dependence on *special* operating system support. Almost all previous work in this area lacks at least one of these properties. The one exception is Moir's work [74] which is more expensive than ours in space and time (a consequence of operating on local copies rather than updating in place). My approach differs in two ways from previous work. First, I use a O(1) binary synchronization primitive. Second, I allow the programmer to control concurrency in the input to the non-blocking transformation.

Herlihy [39] described the first universal transformation from arbitrary sequential code to a non-blocking version. He atomically replaced a pointer to an object with a pointer to a modified copy only if the original was unchanged during the modification. Like our algorithms, this is also asymptotically optimal — the cost of the local update is equivalent to the sequential cost of the algorithm, and the cost of a single copy is constant (that is, it is independent of the number of competing processes). This asymptotic performance comes at the cost of a very high constant factor: the entire data structure must be copied. It also has a significant worst-case space requirement: it may need enough space to hold p copies of the entire data structure. An equally significant problem is that the algorithm does not support multi-object updates (except by treating all possible concurrently accessed objects as one big single object).

Most earlier algorithms are designed for static transactions; to support dynamic transactions they use the caching approach originally proposed by Barnes [14]. Our algorithm requires no caching, but directly executes a step of the universal construction with each tentative write.

Anderson et al. [4] and Israeli et al. [47] use CASn to implement disjoint-access-parallel multi-object updates, by essentially using Herlihy's approach and copying each object (or parts of each object using the large-object-protocol), updating these local copies, and using CASn on the roots of each object to make the update atomic. They support only static transactions, are expensive in space and time (the CASn has costs that are a function of the number of processes (In [47] the cost is actually proportional to P^3, where P is the maximum number of processes in the system) and requires multiple words to implement each word of the special memory in which you can store pointers to the roots of objects). In contrast, my algorithms support dynamic transactions, have cost independent of the number of processes, and have very low space overhead.

These efforts depended only on universal primitives. Significant performance improvements can be realized by using special operating system support to detect long delays. Examples of such approaches are Allemany and Felten's SOLO+Logging protocol [2] and Lamarca's SOLO-cooperative protocol (both described more fully in Section 2.7.3). Both Lamarca and Allemany report significant improvements due to contention reduction by using advisory locks . However, both depend on special OS features (e.g. scheduler activations and process migration) that might not always be present, are potentially unsafe, and that do not scale well. Section B.2 describes the problems with such OS dependence. The algorithms are not always robust if the process holding the lock is terminated (or if the operating system is restarted on the node upon which the lock-owner is running). Further, this OS based approach might not be practical inside the kernel, nor economical in a system which synchronizes with a large number of asynchronous handlers (signals must be treated like context switches). Finally, neither scheme supported disjoint-access-parallel updates (even at the granularity of objects) of multiple objects.

The algorithms presented in this thesis achieve the same (or better) results without dependence on special OS support. Rather than depend upon the OS to clear the advisory lock, the advisory lock is freed by the waiters, using exponential backoff, assuming only the ability to determine (heuristically, if an oracle is unavailable) the current-process on a given machine. Direct-updates in place can be safely done by performing each write in parallel with `DCAS` checking the current state of the transaction. In comparison, the strictly non-blocking approach (SOLO) of Allemany and Felten requires as much copying as Herlihy's approach — OS support merely reduces contention. In the effectively non-blocking approach (SOLO with Logging), an undo log must be constructed, with cost comparable to our logging cost.

The work by Moir is the most promising work to date. He provides an implementation of `CAS`n over at most P contending processes. His `CAS`n works over a memory of M words, but requires a lock word of size $\log P$ for *each* word in memory. (A further restriction is that the operands to `CAS`n cannot use the full word of memory, because some space must be reserved for a tag). To support transactions, he divides memory into M/S S word blocks (pages). Like Barnes' approach, he copies each block accessed to a local copy (up to a maximum of T blocks), and operates on the local copy (so reads and writes cost $O(\log(T))$). At the end of the transaction the pointers to the blocks in the shared memory (shared page table) are atomically replaced by pointers to the local modified copy using the `CAS`n operation defined out of `CAS1`. The cost of Moir's STM (that is, the cost of STM over the basic cost of the transaction without any synchronization) is roughly $O((R + W)\log(T))$, compared to $O(W)$ for our STM algorithm. The space requirement is $O(\frac{M}{S} + TSp)$ compared to $O(Wp)$ for our algorithm. Any attempt to reduce the space cost of Moir's algorithm by reducing S (to get $TS \leq W$), will increase the size of block table. The size of the table of block pointers is M/S words, so reducing S increases the size of this table. (Also, T probably increases for smaller S, because it is likely that the expected number of writes hitting the same block will decrease.) Further, there is an additional space cost of $O(\frac{M}{S}\log P)$ for Moir's algorithm, for a total of $O(\frac{M}{S}(1 + \log P) + TSp)$. This cost arises because the simulated `CAS`n must operate on M/S words, and requires a `proc` field per word.

The space advantage of my algorithm over Moir's is compelling. The significant difference in constant factor in run-time is, too. There is also the (minor) advantage that my algorithm will work where copying the data is not an option (e.g. data-structures inside an OS that are tied to a specific location because of device constraints). However, these advantages are more indicative of the fact that I assume an $O(1)$ `CAS2` while Moir assumes only an $O(1)$ `CAS1`.

It is fair to ask whether the advantage of my algorithm over Moir's is due to a better algorithm or to better primitives. We can compare the algorithms more directly by looking at both in systems with identical underlying primitives.

In systems that do support $O(1)$ `CAS2`, we can replace Moir's MWCAS with a more efficient implementation built out of `CAS2`. The space requirement is no longer proportional to the size of the entire transactional memory. However, even so, Moir's algorithm is not competitive. It is optimized for the case of an inefficient `CAS`n, minimizing its use, and thus does not gain much when $O(1)$ `DCAS` provides a better `CAS`n. The original analysis holds, with Moir's requiring significantly more space and time due to copying blocks.

On systems that do not support an $O(1)$ `CAS2` operation, we can consider implementing our STM algorithm using Moir's implementation of `CAS`n to provide an inefficient `CAS2`. We require $O(M \log(P))$ space for this emulation, since we need to apply `CAS2` to the entire transactional memory, not just M/S locations. This overhead can be larger than Moir's when $(TSp + (1 + \log(P))M/S) < (M \log(P) + Wp)$. Since $\log(P)$ bits probably fit in 1 word, and since W may be close to T, we can say, roughly, that our `CAS1` algorithm uses more space than Moir's when $TSp < M$. In practice, experience shows that in real systems contention on shared objects is low, hence p would not need to be very large. For small p, our algorithm uses more space than Moir's. The space cost of `DCAS` emulation eats up the savings in avoiding copying blocks. `DCAS` is expensive in Moir's emulation. We use $O(W)$ `DCAS`'s vs. one `CAS`n in Moir's algorithm. The n in the `CAS`n is $O(T)$, but, in practice, T is almost always smaller than W.

In conclusion, it seems as if the advantages of my STM algorithm over Moir's are due to *primitives* rather than *algorithms*: each algorithm is tuned to a different model of the underlying system. The performance advantages of my algorithm over Moir's is evidence that `CAS2` enables practical NBS while `CAS1` still has prohibitive costs.

One substantial distinction between the algorithm I present and previous algorithms lies in the input to the transformation.

The idea of universal constructions was originally raised to simplify the design and implementation of non-blocking algorithms. As such, the input to the transformation need only be code that programmers understand and can reason about. However, all universal constructions proposed in the literature are limited to input of sequential specifications, presumably because these are more amenable to proof.

Programmers are experienced in dividing programs into concurrent and non-concurrent sections through the use of critical sections or locked objects. Introducing concurrency into transactions may improve performance. If the underlying implementation merely uses the mutual exclusion to denote (lack of) concurrency, and doesn't suffer from deadlock or priority inversion, then the right starting point for universal constructions seems to be algorithms that denote concurrency, not merely a sequential specification.

Note that my transformation accepts a strict superset of inputs acceptable to conventional universal transformations, and is thus legitimately comparable. If one chose, one could restrict inputs to purely sequential specifications.

The multi-object universal construction I proposed in this chapter is an example of a universal construction that allows concurrency. Programmers can write programs that can fork and proceed in parallel as long as they don't simultaneously access a single shared domain. In the (guaranteed) absence of deadlock and priority-inversion, the input is "well understood" in the same sense that a sequential specification is well understood. This allows programmers to exploit concurrency in non-blocking implementations of complex objects. In keeping with Herlihy's dictum, the programmer is responsible for performance, but the system maintains correctness.

Jayanti [50] showed that oblivious universal constructions get at best serial performance. (P processes take $O(P)$ to complete, and can gain no advantage from concurrency.) To exploit concurrency for a particular class of algorithms, Chandra, Jayanti, and Tan [22] proposed a specific semi-universal transformation for *closed objects*[7] that had polylog time worst-case performance (i.e. it performed better than $O(P)$). The construction combined pending operations in a tree-like fashion; the tree was spread over several processors. Composition of independent operations can proceed in parallel on different branches of the tree.

However, it seems unreasonable to require new transformations for each class of algorithms. This restriction holds for all previous universal constructions, since they are limited to sequential specifications as input. Thus *each* class of algorithms wishing to exploit concurrency must have its own semi-universal transformation. In contrast, in my construction, the concurrent tree used

[7]An object is "closed" if functional composition is closed. This allows operations to be combined without knowing the state of the object. For example, the two operations `+5` and `+7` on an object D, can be combined to a single `+12` operation with no knowledge of the current value of D.

by Chandra et al. to combine closed operations can be expressed as the *input* to our universal construction. One must still reason about the correctness of the input, but the non-blocking property is guaranteed by my construction. No new semi-universal transformations need be written for any other algorithms, either.

3.4 Conclusions

This chapter shows how the availability of hardware $O(1)$ `DCAS` can significantly improve universal constructions of non-blocking objects compared to the best available constructions on systems only supporting unary `CAS`. In particular, I identify weaknesses with `CAS`-only algorithms with respect to fault-tolerance, worst-case running time, and space overhead. I demonstrate that `DCAS`-based algorithms can overcome these problems.

Non-blocking algorithms require some indication of transaction state per unit of concurrency. For `CAS`-only algorithms, the block size (unit of concurrency) must either be a single machine word or else a copying technique must be used. Thus, there is either space overhead per word of shared memory, or else unmodified words must be copied (and space is needed to store the copies of each modified block). In contrast, in this chapter I prove that for a universal construction based on `DCAS` a *single* state word can be checked in parallel with *many* independent one word checks. Thus, space overhead can be $O(d)$, where d is the number of concurrently accessed objects (typically 1 per transaction) rather than $O(M)$, where M is the size of shared memory.

Algorithms depending only on unary `CAS` may depend upon OS support to protect the integrity of a data structure. This avoids the space and time overhead incurred by either copying or using extra space per word of memory. I present an algorithm using `DCAS` that tests a single state word in parallel which each modification. The overhead is comparable to the OS-based algorithms, but `DCAS` algorithms do not need to either copy objects or depend upon OS support. This makes `DCAS` algorithms more portable, robust in the face of OS failure on a node, and applicable even in cases where synchronization is needed between interrupt/signal handlers and main-line code.

`DCAS` also improves worst-case time compared to `CAS`-only algorithms. In this chapter I show that by coupling leader election with `CAS`, `DCAS` can determine which contending process proceeds in $O(1)$ and can make progress independent of the number, p, of contending processes. In the worst-case, `CAS`-only algorithms may have to loop $O(p)$ times in the critical path.

The best universal constructions implemented only from `CAS` are either inefficient in time or space, or depend upon fragile special-purpose OS support. Similar constructions, implemented from $O(1)$ `DCAS` and supporting the same properties, need not depend on special OS support, can survive OS failure, have substantially lower space requirements, and have worst-case time overhead that is on the order of the number of words updated. Direct implementations are always at least as efficient as algorithms derived from universal constructions — in systems that provide only `CAS`, universal constructions are needed more frequently than in systems which provide `DCAS`.

Chapter 4

The Synergy Between Good System Structure and NBS

4.1 Introduction

NBS has clear advantages over blocking synchronization. However, NBS has not been widely deployed to date because it *seems* to present the programmer with a choice of either poor performance or incredible complexity.

Somewhat surprisingly, we did not have this experience in designing the Cache Kernel [24, 34], a multiprocessor operating system kernel. The designers of the *Synthesis V.1* operating system [64] did not experience such problems, either. *Every* data structure of interest in these systems was synchronized using NBS with good performance. Further, the code was not overly complicated. Why?

There is synergism between non-blocking synchronization and good operating system structure. NBS is often only practical by exploiting properties present in well-designed systems. Conversely, blocking synchronization (as opposed to non-blocking synchronization) can constrain system structure and adversely impact modularity, reliability, and scalability of a system. A central contention of this dissertation is that non-blocking synchronization is the best synchronization mechanism for scalable, reliable systems.

I have identified four properties of the Cache Kernel that are sufficient to design and implement simple and efficient non-blocking data structures. By isolating these particular properties, I make it possible for *any* system to gain similar benefits from NBS by ensuring that the system possesses these properties. In systems that exploit these properties most shared data structures are amenable to direct non-blocking implementations, rather than requiring universal transformations. The algorithms perform better than the best previously published non-blocking algorithms. The algorithms are also "simpler" by standard metrics such as lines of code, cyclomatic complexity [65], and comparison complexity. More importantly, I demonstrate that non-blocking algorithms that depend upon these properties are simpler because there are issues that they need not address which (a) must be addressed in non-blocking algorithms that cannot depend on these properties, and (b) addressing these issues adds complexity and cost to code.

The Cache Kernel exploited three main techniques to design and implement efficient non-blocking algorithms. These techniques were already being used, independent of NBS, for modularity, performance and reliability in the Cache Kernel. The main techniques we depended upon were *type-stable memory management* (TSM), *contention-minimizing data-structures* (CMDS), and minimizing the window of inconsistency. The last two properties are likely to be found in most well designed systems. They are not peculiar to the Cache Kernel. TSM is not widely deployed, although stronger properties (which provide equivalent benefits to NBS) can be found in other real systems. TSM can provide many benefits for non-blocking algorithms as well as systems in general.

We also exploited the availability of atomic `DCAS` (`Double-Compare-and-Swap`). `DCAS` is defined in Figure 1.3. That is, `DCAS` atomically updates locations `addr1` and `addr2` to values `new1` and `new2` respectively if `addr1` holds value `old1` and `addr2` holds `old2` when the operation is invoked. This chapter assumes the existence of efficient `DCAS`. However, as noted, hardware support for `DCAS` is only available on the 680x0 family of processors. Chapter 5 presents a hardware implementation of `DCAS` and compares that to software and OS-based implementations.

The rest of this introduction discusses the coupling between synchronization and system structure. The next section (4.2) describes type-stable memory management, which facilitates implementing non-blocking synchronization as well as providing several independent benefits to the software structure. Section 4.3 describes the contention-minimizing data structures which have benefits in performance and reliability for lock-based as well as non-blocking synchronization. Section 4.4 describes our approach to minimizing the window of inconsistency and the systems benefits of doing so. Section 4.5 describes the non-blocking synchronization implementation in further detail with comparison to a blocking implementation. The non-blocking data structure used inside one particular operating system, the Cache Kernel, is described in Section 4.5.3. Section 4.6 describes the performance of our implementation using simulation to show its behavior under high contention. Section 4.7 describes how our effort relates to previous and current work in this area.

4.1.1 Synchronization and system structure

There is a coupling between *system structure* and the synchronization mechanisms used by the system. The term “system structure” is used loosely in many contexts. In this dissertation, system structure refers to the organization and timing of software modules, threads of control, and communication paths in the system. (This encompasses the decomposition of the system into components, the organization and mapping of the system’s software components to processes, processors, and address spaces, and the implementation of inter-component communication and synchronization).

Synchronization and overall system structure interact in three basic ways.

1. *The synchronization mechanism imposes constraints on the system design.* The cost of synchronization can determine the granularity of system modularity and concurrency. The manner in which the synchronization mechanism solves problems can affect the choice of data structures. The semantics and limitations of the synchronization mechanism can constrain the structure and semantics of the system as a whole.

   For example:

   - expensive synchronization costs force a system to use coarse-grained concurrency, reducing available parallelism.
   - Blocking synchronization can force a system to refrain from performing any useful work in signal handlers, possibly increasing latency.
   - Situations that are difficult to synchronize (e.g. multi-object updates using NBS) can cause a system redesign or a re-specification.

2. *Specific features of the system may be exploited in order to avoid, simplify, or improve the performance of the synchronization.* For example, systems with low contention (or a small number of processors) may not need exponential backoff or queuelocks. Simple spin-locks might suffice. Real-time systems which enforce strict priorities and take no page-faults can depend on the scheduler’s priority system to make certain operations atomic with almost no extra cost (e.g. [6, 7]).

3. Finally, *one must evaluate a synchronization algorithm in the context of the entire system.*

   In particular, it is the *incremental* cost of a synchronization method in a particular system that is relevant, and not its absolute cost. Consider an algorithm which, in isolation, is deemed too expensive because it requires logging all modifications to memory. If, in the system of

> interest, logging is already required in any case for recovery, then the incremental cost of logging is zero, and we would not consider the need for logging to be a black mark against this algorithm.
>
> Conversely, if a method of synchronization lacks some property, it is only relevant if the property is needed in systems we are concerned about. For example, if a synchronization algorithm does not allow concurrent insertions into a priority queue, we may consider this a flaw. However, in a system where concurrent insertions are not needed or used, such a restriction is not a drawback.

In particular, NBS helps system structure in several ways. First, NBS provides run-time isolation between synchronization and scheduling: this allows a separation between policy (scheduling) and mechanism (blocking, wakeup, and synchronization). Second, NBS provides some design-time isolation between independent components of the system: modules can be composed without the need of exposing internal locking strategies. Third, NBS removes constraints on system decomposition and modularity: synchronized routines can be called anywhere, increasing the amount of shared code and increasing flexibility (e.g. adding functionality to asynchronous handlers).

NBS can exploit the structure of well-designed systems if the programmer can depend upon the properties and techniques described in the next several sections. Each section describes one property, explains how it aids NBS algorithms, and explains why it is beneficial to systems in general. If a property provides benefits to systems independent of NBS, then higher implementation costs are justifiable and there is increased likelihood that systems possess that property.

4.2 Type-Stable Memory Management (TSM)

Type-stable memory management (TSM) refers to the management of memory allocation and reclamation so that an allocated portion of memory, a *descriptor*, does not change type. TSM is a guarantee that an object O of type T remains type T as long as a pointer to O exists. It does *not* guarantee that O will not be freed. TSM is a fancy name for an extension of an old idea: the notion of static allocation. For example, the process descriptors in many operating systems are statically allocated at system initialization and are thus type-stable for the lifetime of the system execution. A system supports type stable memory management for type T by making sure that in spite of allocation and reclamation, objects of type T do not change type within some time bound t_{stable} of last use. (Type stability follows trivially in some systems if $t_{stable} == \infty$: this is just static allocation.)

Our notion of TSM incorporates three basic extensions to a conventional implementation of allocation.

- First, as in static allocation, a descriptor remains a valid instance of the type even when it is not active, i.e. on the free list.

- Second, TSM allows multiple memory allocation pools for the same type. Support for multiple pools was originally a performance optimization. For example, there can be a pool of thread descriptors per cluster of processors on a large-scale multiprocessor to minimize contention between clusters. NBS additionally uses multiple allocation pools to enforce a slightly stricter notion of "type", as described in Section 4.2.1.1.

- Finally, in contrast to static allocation (but in common with garbage collection), the type of a portion of memory *can* change over time, but only as long as it is type-stable over some time t_{stable}. More specifically, a descriptor has to be inactive for at least t_{stable} before it can be reallocated as a different type[1]. However, for simplicity, we assume an infinite t_{stable} for this discussion.

Many (if not most) work on NBS in the literature simply *assumes* TSM (c.f. Herlihy [40, 41]). The algorithms, as presented, are incorrect unless the system supports type-stability properties at least as strong as TSM.

4.2.1 NBS benefits from type-stable memory management

TSM simplifies the implementation of non-blocking synchronization algorithms. A pointer of type `T1 *` to a descriptor can never point to a descriptor of another type. Type changes can only result from the target area of memory being freed and reallocated as type `T2`; with TSM this cannot happen because the descriptor of type `T1` is type-stable. Without TSM a pointer may point to a descriptor that has been deleted and reallocated as a different type. This type error can cause a random bit-field to be interpreted as a pointer, and cause the search to infinitely loop, perform incorrectly, raise an exception due to unaligned access, or read a device register. TSM is a simpler and more efficient

[1] An example of a TSM implementation is a collection of descriptors that are stored in a set of page frames which are allocated and released over time. When more descriptors are required, additional page frames can be allocated from the general pool and when the number of descriptors falls, the descriptors may be consolidated into a smaller number of pages and the excessive page frames returned to the general pool. However, the release of page frames to the general pool must be delayed sufficiently to ensure the type-stability property. This delay provides a useful hysteresis to the movement of pages between this descriptor collection and the general page pool.

way of ensuring this type safety than other techniques we are aware of that prevent reallocation (such as automatic garbage collection mechanisms or reference counts), or that detect potential reallocation (such as per-list-element version numbers) or that make reads "safe" (such as using DCAS to check the list's version number concurrently with each pointer dereference). In addition to the direct computational overhead added by those approaches, they convert all reads into writes, and do not support concurrent read-only operations.

Consider, for example, the code shown in Figure 4.1 to do a non-blocking deletion from a linked list[2].

```
/* Delete elt */
do {
  backoffIfNeeded();
  version = list->version;

  for(p=list->head;(p->next!=elt);p=p->next){
    if (p == NULL) {  /* Not found */
      if (version != list->version)
        { break; }    /* Changed, retry */
      return (NULL);  /* Really not found */
    }
  }
} while(version != list->version ||
        !DCAS(&(list->version), &(p->next),
              version,          elt,
              version+1,        elt->next))
```

Figure 4.1: Deletion from the middle of list, protected by DCAS and version number.

The delete operation searches down a linked list of descriptors to find the desired element or detect the end of the list. If the element is found, the element is atomically deleted from the list by the DCAS operation. The DCAS succeeds only if the list has not been modified since the delete operation started, as determined by the `version` field.

The code only needs to check for conflicts once it reaches the desired element or the end of the list. The descriptors are TSM so each link pointer `p` is guaranteed to point to a descriptor of this type. Without TSM some protection is needed to avoid the errors mentioned above. Either each dereference must use a complicated DCAS protocol, or each pointer must include a version number that is managed on each modification and checked on each dereference, or storage management must be tightly coupled to this algorithm to guarantee that no descriptors are freed and reallocated

[2]The list is initialized with a dummy node at the head, thus deletion of the first element works correctly.

while any process is inspecting the list. Any of these options significantly increases complexity and decreases performance.

There are some non-blocking data structures not aided by type-stability. If all pointers are static (from the header of the object, for example), then the object is type-stable even if the system isn't TSM, in the sense that the pointers are always "safe" to dereference. Alternatively, if the entire state of the data-structure is encoded in k words and the system supports an atomic $\texttt{CAS}n$, $n \geq k$, then type-stability is not needed. In practice, such data structures are extremely rare.

4.2.1.1 The meaning of "type" in type-stability

Type-stability is sufficient to provide read-safety, but some care must be taken to understand when two objects have different types with respect to TSM. Sometimes simply ensuring that an object's type remain stable may not be sufficient to aid NBS.

For example, consider a descriptor, D, containing a `next` field. List termination must be consistently denoted in every list which may contain D. If D is sometimes stored in a collection where list termination is denoted by a `null` pointer, and sometimes stored in a list where termination is denoted by a pointer to some other distinguished object (e.g. the header of the list, or the first element), then pointers to D are *not* sufficiently type-stable to aid NBS. An algorithm that was redirected from a list with NULL termination to a list terminated by a pointer to the head of the list may result in an infinite loop. One must either conventionally agree on termination conventions for *all* collections in which D is stored, or segregate descriptors' allocation pools based on usage semantics in addition to type.

The latter approach is straightforward. Two collections, both containing objects of type D, require separate allocation pools if an algorithm used in one collection would raise a type error or fail to halt if a pointer from one collection happened to point to an element currently in the other collection. In essence, our notion of an object's "type" for TSM may need to be specialized slightly beyond the basic definition of class in C++ — we may need to have multiple managers for a given type, corresponding to client's usage.

There are stronger stability properties than TSM — consider systems with automatic *garbage collection*. A GC guarantees that an object is never freed or reallocated as long as a pointer to it exists. This not only guarantees type-safety for the lifetime of the pointer, but also guarantees the stability of object identity. Such stronger properties may provide occasional optimizations for non-blocking algorithms (Section 4.6.3 shows how list-based stacks can be implemented in garbage collected systems using only unary `CAS`, while in systems with no stability properties either `DCAS` or per-pointer version numbers are required.) However, TSM is sufficient to allow algorithms to depend upon reads being safe. TSM is (by definition) the weakest property that keeps algorithms safe from type errors.

4.2.2 Further benefits of TSM

Besides the benefits to non-blocking synchronization, TSM has several important advantages in the construction of modular, reliable, high-performance operating systems. First, TSM is efficient because a type-specific memory allocator can normally allocate an instance of the type faster than a general-purpose allocator can. For example, allocation of a new thread from a free list of (fixed-size) thread descriptors is a simple dequeue operation whereas a general-purpose allocator like `malloc` may have to do a search and subdivision of its memory resources. The class-specific `new` and `delete` operators of C++ support a clean source code representation of TSM. This allocation can be made even more efficient with many types because a free (or inactive) descriptor is already an instance of this type, and so may require less initialization on allocation than a random portion of memory.

Second, TSM aids reliability because it is easier to audit the memory allocation, locating all the descriptors of a given type and ensuring that pointers that are supposed to point to descriptors of a given type actually do so. With fixed-size descriptors, TSM also avoids fragmentation of memory that arises with general-purpose allocators. Fragmentation can cause failure as well as poor performance. Relatedly, TSM makes it easier to regulate the impact of one type of descriptor on the overall system resources. For example, with a collection of descriptors that are allocated dynamically using the page frame approach described above, the number of pages dedicated to this type can be controlled to avoid exhausting the memory available for other uses, both from overallocation and from fragmentation of memory.

TSM also minimizes the complexity of implementing the caching model [24] of descriptors in the operating system kernel. In this approach, the number of descriptors of a given type is limited but an allocation never fails. Instead, as in a memory cache, a descriptor is made available by its *dirty* data being written back to the higher-level system management and then reused to satisfy the new allocation request. This mechanism relies on limiting the number of descriptors, being able to locate an allocated descriptor to reclaim, and being able to determine the dependencies on these descriptors. TSM simplifies the code in each of these cases.

4.2.3 Implementing Type Stable Memory Management (TSM)

TSM also allows a modular implementation. From an object-oriented programming standpoint, there can be a base class descriptor manager class that is specialized to each type of descriptor. For example, there is a `CacheKernelObjMan` class in our operating system kernel that provides the basic TSM allocation mechanism, which is specialized by C++ derivation to implement `Thread`, `AddressSpace`, `Kernel` and `MemMap` types as well as several other types.

It is realistic to assume system support for TSM for two reasons.

First, there are already systems with type-stable memory management. Any system that uses type specific static allocation is type-stable. Objects of type T are allocated from, and returned to, a type-specific free list: t_{stable} is infinite in these systems. Operating system kernels which use static allocation, or applications which use program termination to free memory, are also examples of such systems. Another class of examples is the type-stability of objects in a garbage collected system. Garbage collection imposes stricter stability semantics on objects than in type-stable memory — an object not only never changes type while a pointer points to it, but the object also maintains its identity (the object cannot be freed and reallocated).

Second, the cost of implementing type-stability is low. It is possible to implement type-stability on a per-type basis, so the total burden need not be large. Unlike garbage collection, type stability can be implemented locally (pages can be returned to the general pool without checking tables of remote pointers), and the cost is a function of the number of frame activations, rather than the number of references (Appendix E describes implementations of TSM in more detail.)

4.3 Data Structures that Minimize Contention

In addition to TSM, the second feature we exploited to provide for efficient non-blocking synchronization was contention minimizing data structures (CMDS). Data structures in the Cache Kernel were designed and implemented to minimize both logical and physical contention. By *logical contention*, we mean contention for access to data structures that need to be controlled to maintain the consistency and semantics of these data structures. By *physical contention*, we mean the contention for access to shared memory that needs to be controlled to maintain the consistency and semantics of the memory system[3].

[3]Physical contention is separate from logical contention because one can have logical contention without physical contention as well as vice versa, or so called *false sharing*. For example, if two shared variables reside in the same cache line unit, then there can be physical contention without logical contention. Two processors may attempt to update the

4.3.1 NBS benefits from contention minimizing data structures

Minimizing logical contention with non-blocking synchronization minimizes the overhead of conflicting operations failing and being retried. It also avoids the complication of complex backoff mechanisms as part of the retry. Finally, and perhaps most importantly, systems designed to minimize contention naturally lead to systems that have relatively few types of shared data structures. In our experience, these data structures are typically amenable to an efficient direct non-blocking implementations.

Minimizing logical contention benefits both blocking and non-blocking synchronization. However, the cost of logical contention can be much higher for NBS. For non-blocking algorithms (in the absence of contention reduction algorithms) conflicting operations normally proceed in parallel (exhibiting what is called "useless parallelism"). If P processes proceed in parallel, $P-1$ of the processors waste their CPU cycles on conflicting operations that are destined to fail. Worse, the memory contention and bus traffic slow down even the one process that is destined to succeed. In contrast, with blocking synchronization, if P processes proceed in parallel, $P-1$ of the processors spin harmlessly (wasting only their own cycles) on a local cached copy of the lock. Thus, eliminating logical contention confers larger gains on NBS than on blocking algorithms.

To avoid problems with useless parallelism, NBS algorithms implement contention reduction techniques (such as exponential backoff or advisory locks or "helping") around each operation. In the common case of no contention, the cost and complexity of setting up such algorithms have measurable effect on throughput (c.f. examples later in this chapter). In systems designed to minimize worst-case logical contention, it is possible to completely avoid the contention reduction techniques.

While TSM improves the non-blocking implementation of a given data structure, CMDS aids NBS by reducing the set of relevant data structure types. More particularly, in a system designed with shared objects that have low contention and small scale, the set of shared objects requiring synchronization are relatively simple. We have found that these shared objects are amenable to direct NBS implementations using `DCAS`. Massalin et al. [64] reports similar conclusions.

4.3.2 Further benefits of CMDS

The *spatial locality* of data access achieved by these techniques provides significant benefit for synchronization, whether non-blocking or conventional locks. This spatial locality also minimizes the consistency overhead when the system is running across multiple processors, with each processor

variables simultaneously, with each processor updating a separate variable.

caching portions of this shared data. In general, our experience independent of NBS (e.g. [25]) suggests that it is better to (re)structure the data structures to reduce contention rather than attempt to improve the behavior of synchronization techniques under high contention. Low-contention algorithms are simpler and thus easier to get right, and faster as long as contention is actually low.

Contention minimization is widespread, even on systems not supporting NBS. Gupta et al. [98] report low contention on OS data structures and Michael et al. [67] report low contention on shared application data structures.

4.3.3 Implementing contention minimizing data structures (CMDS)

Most of our techniques for contention minimization are well-known. One method of contention minimization is replicating the functionality of a shared data structures for each processor, rather than using a single shared object. In particular, there are per-processor ready and delay queues in the Cache Kernel, so contention on these structures is limited to signal/interrupt handlers and management operations to load balance, etc. being executed by a separate processor.

Some data structures cannot be easily made per-processor without complete replication. Even so, useful contention minimizing techniques apply. Per-processor "caches" for such data structures allow a significant number of references to be purely local, reducing contention on the shared object. For example, the Cache Kernel uses a per-processor signal-delivery cache to reduce access to the shared signal mapping table[24]. This per-processor "cache" approach is similar to that provided by a per-processor TLB for address translation. The TLB reduces access to the real virtual address space mapping structure, which is necessarily shared among threads in the address space.

Contention on a data structure is also reduced in some cases by structuring it as a multi-level hierarchy. For example, a large list that is searched frequently may be revised to be a hash table with a version number or lock per bucket. Then, searches and updates are localized to a single bucket portion of the list, reducing the conflict with other operations, assuming they hash to different buckets. The upper levels of the hierarchy are read-only or read-mostly: descriptors are only added at the leaves.

Physical contention is also reduced by using cache-aligned descriptors. TSM with its restricted allocation of descriptors can also reduce the number of pages referenced as part of scan and search operations, reducing the TLB miss rate, another source of physical contention. Finally, in this vein, commonly updated fields are placed contiguously and aligned to hopefully place them in the same cache line, thereby making the updates more efficient.

4.4 Minimizing the Window of Inconsistency

The Cache Kernel was also structured to minimize the window in which a data structure was inconsistent. (This is sometimes referred to as the "window of vulnerability" (e.g. [39])). This provides *temporal locality* to a critical section.

4.4.1 NBS benefits from minimizing the window of inconsistency

Techniques that minimize the window of inconsistency allow efficient non-blocking synchronization. A typical locking implementation of a data structure may involve a number of writes inside a critical section. If no care is given to reducing the window of inconsistency, these writes may be spread throughout the critical section. Designing or restructuring code to minimize the window of inconsistency has two effects. First, all modifications are co-located (usually near the end of the update). Second, updates are grouped into distinct consistent updates. For example, system accounting information is often updated alongside a modification to a data structure, inside a single critical section. The data structure is not necessarily inconsistent after the modification and before the accounting information is updated. An implementation that is careful about minimizing the window of inconsistency groups these two sets of writes separately.

Non-blocking implementations can take advantage of this constrained structure. In particular, a non-blocking update typically consists of one of the *direct-implementation* techniques described in section D.2 (e.g. a `DCAS` operation that updates the version number plus one other location, with the version number ensuring that the data structure has not been changed by another concurrent update). That is, the window of inconsistency is reduced to the execution of a single `DCAS` operation. Reducing the granularity of the consistent updates increases the likelihood that direct implementation using a single `DCAS` is possible.

Reducing the window of inconsistency is beneficial even when the consistent update consists of more than 2 locations. By grouping all modifications in one place, static, rather than dynamic, transactions can be used. Much complexity in the universal transformations described in Chapter 3 can be eliminated if the entire set of addresses being updated were known at the start of the transaction. (For example, in Section 3.2.3, address sorting can be used rather than the entire precedence mechanism presented there.)

4.4.2 Further benefits of minimizing the window of inconsistency

These techniques to reduce the window of inconsistency have other benefits as well. In particular, the reduced window of inconsistency reduces the probability of a failure, such as a thread termination, corrupting the system data structures. They also reduce the complexity of getting critical section code right because it is shorter with fewer separate control paths through it and therefore easier to test. Some of this structuring would be beneficial, if not required, for an implementation using lock-based synchronization because it reduces lock hold time, thereby further reducing contention. In fact, such approaches are already common: techniques such as optimistic locking [53] and field calls [32] are simply approaches that structure the code to minimize the window of vulnerability.

4.4.3 Reducing the window of inconsistency

Again, we use familiar techniques. The basic pattern is to read all the values, compute the new values to be written, and then write these new values all at once after verifying that the values read have not changed. Generally, a structure is inconsistent from the time of the first write to the point that the last write completes. Therefore, removing the computation from this phase minimizes the window of inconsistency. To minimize the cost of verifying that the read values have not changed, we often use a version number that covers the data structure and is updated whenever the data structure changes. The use of a version number also avoids keeping track of the actual location read as part of the operation.

The window of inconsistency is also minimized by structuring to minimize physical contention as part of data structure access.

Physical contention increases the time for a processor to perform an operation because it increases the effective memory access time.

4.4.4 Increased robustness through relaxed consistency requirements

Another way of reducing the window of inconsistency is by redefining "consistency". In our experience, we have found that system robustness is sometimes enhanced by relaxing the consistency requirements of a data structure. (At this point we have no hard data to support this subjective feeling.) For example, if we require a descriptor to always belong to exactly one collection (e.g. a list) then we must forbid the system from going through states where the descriptor temporarily belongs to no collection, or belongs to multiple collections. The alternative is to relax this requirement and allow the descriptor to belong to any subset of the legal collections. This requires us to recode all clients to deal with the slightly wider variety of states. In practice we have found this recoding to be insignificant and the result aids system robustness and recovery.

This relaxation of consistency requirements has often been motivated by initial difficulties in maintaining a consistent state while performing non-blocking updates. We have found that such difficulty is often (but not always) a clue pointing to a fragile design. Some specific examples are mentioned in Section 4.5.3 and in Section D.5.2.

4.5 Non-Blocking Synchronization Implementation

Non-blocking algorithms are relatively simple to implement and verify in systems that possess the properties described above including efficient `DCAS` support. Algorithms may depend upon read-safety and low contention for reads, writes, and memory accesses.

Non-blocking algorithms can be implemented in one of three basic ways:

Direct implementation if the entire state of an update can be encoded in two words then an operation can be made atomic by reading the initial state, performing a computation, and then performing the update in a single `DCAS`.

Universal transformations *Any* data-structure, even if too complex to yield to direct implementation, can have a non-blocking implementation. Universal transformations mechanically produce non-blocking implementations based on a sequential specification. The drawback is poor average case performance.

Special-purpose techniques Some data structures that are not amenable to direct implementation still have efficient implementations. Applying data-structure specific tricks yields algorithms that are far more efficient than universal transformations.

`DCAS`, TSM, and other properties of the Cache Kernel enable a straightforward approach to implementing universal transformations of sequential implementations of data-structures to non-blocking concurrent implementations. The approach is described in Chapter 3.

However, to date, we have not needed to employ either universal constructions or the so-called *helper* techniques in the Cache Kernel and therefore cannot comment on their actual practicality or utility. Moreover, as I note in Appendix D, it seems questionable from the standpoints of both reliability and performance to have threads from separate address spaces sharing fine-grained access to complex data structures. These data structures are also more difficult to program and to maintain and often provide marginal performance benefits in practice, particularly when synchronization overhead and memory contention is taken into account. Their asymptotic performance benefits are often not realized at the scale of typical operating system data structures.

Synchronization of more complex data structures than we have encountered can also be handled by a server module. This approach is less forbidding than a universal transformation. Each operation allocates, initializes and enqueues a "message" for a server process that serially executes the requested operations. Read-only operations can still proceed as before, relying on a version number incremented by the server process. Moreover, the server process can run at high priority, and include code to back out of an operation on a page fault and therefore not really block the operation any more than if the operation was executed directly by the requesting process. The server process can also be carefully protected against failure so the data structure is protected against fail-stop behavior of a random application thread, which may be destroyed by the application.

This approach was proposed by Pu and Massalin [64]. For example, a general-purpose memory page allocator can be synchronized in this manner, relying on TSM memory pool to minimize the access to the general allocator. (Access can be further minimized by per-processor pools.) To date, we have not needed to resort to server modules for synchronization.

Appendix D describes techniques that can be used to implement non-blocking versions of sequential implementations of data-structures. Each technique is illustrated by detailed non-blocking algorithms for specific data-structures. These algorithms depend on `DCAS`, TSM, and CMDS to implement efficient non-blocking synchronization.

The next section describes the common case of direct non-blocking implementations in somewhat more detail. (An even fuller description is available in Appendix D.2).

4.5.1 Direct Implementation of common data structures

Most performance critical shared data structures are collections of fixed-size descriptors. Several collections are queues for service. In the Cache Kernel, for example, thread descriptors are queued in the ready queue and a delay queue of their associated processor. Other collections are lookup or search structures such as a hash table with linked list buckets. For example, the Cache Kernel organizes page descriptors into a lookup structure per address space, supporting virtual-to-physical mapping for the address space.

4.5.1.1 The Base Approach: updates consisting of a single write

The non-blocking synchronization for these structures follows a common base structure. There is a version number per list or container. The DCAS primitive is used to atomically perform a write to a descriptor in a list and increment the version number, checking that the previous value of both has not been changed by a conflicting access to the list. Figure 4.1 illustrated this structure for deleting a descriptor from a list, where the single write to the descriptor was to change the link field of the predecessor descriptor. Inserting a new descriptor D (stored in the argument `entry` in the figure) entails initializing D, locating the descriptor in the linked list after which to insert D, writing the D's link field to point to the next descriptor, and then performing the DCAS to write the link field of this prior descriptor to D and to increment the version, checking both locations for contention as part of the update.

En/dequeuing a descriptor to or from a TSM free list is a degenerate case of deletion because the enqueue or dequeue always takes place from the head: the list acts as a LIFO stack. It is possible to optimize this case and use a single CAS to enqueue and a DCAS to dequeue, without a version number. However, with efficient DCAS support, it is attractive to use DCAS with a version number upon enqueue to allow the version number to count the number of allocations that take place. (As another special case, an operation requiring at most two locations for the reads and writes can be updated directly using DCAS. We have used this approach with array-based stacks and FIFO queues.)

4.5.1.2 Enhancements to the base approach: multiple writes and multiple containers

Some operations that involve multiple writes to the same descriptor can be performed by creating a duplicate of this descriptor, performing the modifications and then atomically replacing the old descriptor by the new descriptor if the list has not changed since the duplicate descriptor was created. This approach is a variant of Herlihy's general methodology [41]. However, we use DCAS to ensure atomicity with respect to the entire data structure (the scope of the version number) even though we are only copying a single descriptor[4]. As a variant of this approach, the code can duplicate just a portion of the descriptor, update it and use DCAS to insert it in place of the original while updating

[4]The basic Herlihy approach involves copying the entire data structure, modifying the copy, and then atomically replacing the old copy with the new copy using CAS, and retrying the entire copy and modifying if there is a conflict. Herlihy's Large Object protocol, duplicates only part of the data structure, but requires transitively copying (all the way to the root) all parts of the object that point to modified locations. Our approach reduces the allocation and copy cost to a single descriptor rather than almost the entire data structure, but requires DCAS.

a version number.

As a further optimization, some data structures allow a descriptor to be removed, modified and then reinserted as long as the deletion and the reinsertion are each done atomically. This optimization saves the cost of allocating and freeing a new descriptor compared to the previous approach. This approach requires that other operations can tolerate the inconsistency of this descriptor not being in the list for some period of time. If a thread fails before completing the insertion, we rely on a TSM-based audit to reclaim the partially initialized descriptor after it is unclaimed for t_{stable} time. Note that just having a search mechanism retry a search when it fails in conjunction with this approach can lead to deadlock. For example, if a signal handler that attempts to access descriptor D, retrying until successful, is called on the stack of a thread that has removed D to perform an update, the signal handler effectively deadlocks with the thread.

A descriptor that is supposed to be on multiple lists simultaneously complicates these procedures. The techniques described in Chapter 3 can easily handle descriptors on multiple lists, however those techniques impose a performance penalty that is more expensive than equivalent locking implementations in the common case of no contention. So far, we have found it feasible to program so that a descriptor can be in a subset of the lists, and inserted or deleted in each list atomically as separate operations. In particular, all the data structures that allow a descriptor to be absent from a list allow the descriptor to be inserted incrementally. This eliminates the need for relying on the more expensive universal constructions.

4.5.2 Comparison to blocking synchronization and `CAS`-based non-blocking synchronization

Much of the structuring we have described would be needed, or at least beneficial, even if the software used blocking synchronization. For instance, TSM has a strong set of benefits as well as contributing to the other techniques for minimizing contention and reducing the window of inconsistency.

We have found that the programming complexity of non-blocking synchronization is similar to conventional blocking synchronization. This differs from the experience of programmers using `CAS`-only systems. In `CAS`-only systems programmers must rely on universal constructions in order to reduce the conceptual complexity of the algorithm. These approaches generally perform badly.

Non-blocking data-structure specific algorithms have good performance, but, using `CAS`, are difficult to design and complex to understand. `DCAS` enables many non-blocking algorithms with excellent performance (see Appendix D for examples). `DCAS` plays a significant part in the complexity

Synchronization Method	Functions	Avg. Complexity Cyclomatic	Avg. Complexity Comparison	Total Complexity Cyclomatic	Total Complexity Comparison
Spin-lock	3	1.33	7.67	4	23
`DCAS`	3	2.00	11.00	6	33
`CAS`	20	2.70	8.50	54	170

Table 4.1: Some Standard Complexity Metrics for implementations of linked-list priority queues

reduction for these data-structure-specific algorithms. Using the crude metric of lines of code, a `CAS` implementation (Valois [100]) of concurrent insertion/deletion from a linked list requires 110 lines, while the corresponding `DCAS` implementation requires 38 (a non-concurrent `DCAS` implementation takes 25). The `CAS`-only implementation of a FIFO queue described in [70] requires 37 lines, our `DCAS` version only 24. Figures 4.2 and 4.1 show that the `DCAS` implementation of linked lists has complexity[5] 39 compared to 224 for the Valois implementation. My `DCAS` FIFO queue has complexity of 30, which is a slight improvement over the complexity of Michael and Scott's `CAS`-only implementation which has complexity of 38. The `DCAS` versions are correspondingly simpler to understand and to informally verify as correct. In many cases, using `DCAS`, the translation from a well-understood blocking implementation to a non-blocking one is straightforward. In the simple case described in Figure 4.1, the initial read of the version number replaces acquiring the lock and the `DCAS` replaces releasing the lock.

Synchronization Method	Functions	Total Complexity Cyclomatic	Total Complexity Comparison	Total
Spin-lock	3	6	17	23
`DCAS`	3	5	25	30
`CAS`	3	12	26	38

Table 4.2: Cyclomatic and Comparison Complexity Metrics for implementations of FIFO queues

In fact, version numbers are analogous to locks in many ways. A version number has a *scope* over some shared data structure and controls contention on that data structure just like a lock. The scope of the version number should be chosen so that the degree of concurrency is balanced by the synchronization costs. (The degree of concurrency is usually bounded by memory contention concerns in any case). Deciding the scope of a version number is similar to deciding on the granularity

[5] Complexity is measured by summing comparison complexity (also known as "predicate complexity") and cyclomatic complexity for all of the routines in initialization, insertion, and deletion. Comparison complexity and cyclomatic complexity are standard metrics used by software practitioners. I am skeptical of their value as precise quantitative measures, but believe that they are useful as a *general* indication of complexity of software.

of locking: the finer the granularity the more concurrency but the higher the costs incurred. However, a version number is only modified if the data structure is modified whereas a lock is always changed. Given the frequency of read-only operations and the costs of writeback of dirty cache lines, using read-only synchronization for read-only operations is attractive. Finally, version numbers count the number of times that a data structure is modified over time, a useful and sometimes necessary statistic.

Finally, the overall system complexity using blocking synchronization appears to be higher, given the code required to get around the problems it introduces compared to non-blocking synchronization. In particular, special coding is required for signal handlers to avoid deadlock. Special mechanisms in the thread scheduler are required to avoid the priority inversion that locks can produce. Special cleanup-code is required to achieve reliable operation when a thread can be terminated at a random time, increasing code complexity. For example, some operations may have to be implemented in a separate server process.

A primary concern with non-blocking synchronization is excessive retries because of contending operations. However, our structuring has reduced the probability of contention and the contention reduction techniques of Chapter 3 as well as the conditional load mechanism described in section 5.4.5, can be used to achieve behavior similar to lock-based synchronization.

4.5.3 Use of NBS in the Cache Kernel

Chapter 1 details the advantages that non-blocking synchronization (NBS) holds, *in general*, over blocking synchronization. There is a *particularly* strong synergy between non-blocking synchronization and the design and implementation of the Cache Kernel [24] for performance, modularity and reliability. First, signals are the *only* kernel-supported form of notification, allowing a simple, efficient kernel implementation compared to more complex kernel message primitives, such as those used in V [23]. Class libraries implement higher-level communication like RPC in terms of signals and shared memory regions [103]. Non-blocking synchronization allows efficient library implementation without the overhead of disabling and enabling signals as part of access and without needing to carefully restrict the code executed by signal handlers.

Second, we simplified the kernel and allowed specialization of these facilities using the C++ inheritance mechanism by implementing most operating system mechanisms at the class library level, particularly the object-oriented RPC system [103]. Non-blocking synchronization allows the class library level to be tolerant of user threads being terminated (fail-stopped) in the middle of performing some system library function such as (re)scheduling or handling a page fault.

Finally, the isolation of synchronization from scheduling and thread deletion provided by non-blocking synchronization and the modularity of separate class libraries and user-level implementation of services leads to a more modular and reliable system design than seems feasible by using conventional approaches.

This synergy between non-blocking synchronization and good system design and implementation carries forward in the more detailed aspects of the Cache Kernel implementation.

Overall, the major Cache Kernel [24] data structures are synchronized in a straightforward manner using the techniques described in this chapter. Threads are in two linked lists: the ready queue and the delay queue. Descriptor free lists are operated as stacks, making allocation and deallocation simple and inexpensive. The virtual to physical page maps are stored in a tree of depth 3 with widths of 128, 128, and 64 respectively. Although the 128 immediate descendants of the root are never deleted, sub-trees below them can be unloaded. Modifications to a map on level 3 are therefore synchronized using `DCAS` with its parent's version number to make sure that the entire subtree has not been modified (e.g. unloaded) in conflict with this update. The Cache Kernel maintains a "dependency map"[24] that records dependencies between objects, including physical to virtual mappings. It is implemented as a fixed-size hash table with linked lists in each bucket. The Cache Kernel maintains a signal mapping data structure which maps addresses to receivers. (This structure supports "memory-based messaging" [27], a technique that allows address-valued signals to be delivered to processes that register their interest in the signal.) There is a signal delivery cache per processor as an optimization for signal delivery to active threads. As noted in Section 4.3.3, this "cache" allows a significant number of signals to be delivered by a processor without accessing the shared signal mapping data structure, which cannot be made per-processor without replicating the entire structure. The signal mapping cache structure is also a direct mapped hash table with linked lists in each bucket. The majority of uses of single `CAS` are for audit and counters.

The Cache Kernel does have updates that are implemented by atomically removing a descriptor from a container, modifying it and (atomically) reinserting it. Other clients of the container tolerate the inconsistency of a descriptor being missing during the update. We considered duplicating the descriptor, modifying the copy, and atomically replacing the old descriptor, but generally opted to save the cost of allocating and freeing a new descriptor. An example of clients tolerating such an inconsistency is in the Cache Kernel signal delivery mechanism. Signal delivery relies on a list of threads to which a signal should be delivered. Threads may be removed from the list (or specially marked to be skipped) during an update. A thread fails to get the signal if it is not in the list at the time a signal is generated. However, we defined signal delivery to be best-effort because there are (other) reasons for signal drop so having signal delivery fail to a thread during an update is not a violation of the signal delivery semantics. Programming the higher-level software with best-effort signal delivery has required incorporating timeout and retry mechanisms but these are required for distributed operation in any case and do not add significant overhead [103]. These techniques, analogous to techniques used in the transport-layer in network protocols, also make the Cache Kernel more resilient to faults.

Every shared data structure has a non-blocking implementation. None depend upon universal transformations or a server module. In our code to date, the only case of queuing messages for a server module arises with device I/O. This structure avoids waiting for the device I/O to complete and is *not* motivated by synchronization issues.

4.6 Performance

I first describe the performance on the ParaDiGM experimental multiprocessor. I then describe results from simulation indicating the performance of our approach under high contention. Finally, I describe some aspects of overall system performance.

4.6.1 Experimental Implementation

The operating system kernel and class libraries run on the ParaDiGM architecture [26]. The basic configuration consists of 4-processor Motorola 68040-based multiprocessors running with 25 MHz clocks. The 68040 processor has a `DCAS` instruction, namely `CAS2`. This software also runs with no change except for a software implementation of `DCAS`, on a uniprocessor 66 MHz PowerPC 603. We have not implemented it on a multiprocessor PowerPC-based system to date.

As of 1997, kernel synchronization used `DCAS` in 27% of the critical sections and otherwise `CAS`. However, the `DCAS` uses are performance-critical, e.g. insert and deletion for key queues such as the ready queue and delay queue. The only case of blocking synchronization is on machine startup, to allow Processor 0 to complete initialization before the other processors start execution.

The overhead for non-blocking synchronization is minimal in extra instructions. For example, deletion from a priority queue imposes a synchronization overhead of 4 instructions compared to no synchronization whatsoever, including instructions to access the version number, test for `DCAS` success and retry the operation if necessary. This instruction overhead is comparable to that required for locked synchronization, given that lock access can fail thus requiring test for success and retry.

The Motorola 68040's `CAS2` [75] is slow, apparently because of inefficient handling of the on-chip cache so synchronization takes about 3.5 microseconds in processor time. In comparison, spin locks take on average 2.1 μsecs and queuelocks [9, 66] take about 3.4 μsecs. In contrast, the extended instructions we propose in Section 5.2 would provide performance comparable to any locking implementation. In particular, it requires 16 extra instructions (including the required no-ops) plus an implicit `SYNC` in an R4000-like processor (an explicit `SYNC` on the R10000). A careful implementation would allow all instructions other than the `SYNC` to execute at normal memory speed. The performance would then be comparable to the roughly 24 instruction times required by the R4000 lock/unlock sequence. Figure 4.3 compares the overhead in terms of instruction times.

Operation	Instruction Times
`DCAS` using `CAS2` on 68040	114
`DCAS` using `LLP/SCP`	26
SGI R3000 lock/unlock	70
R4000 lock/unlock	24

Table 4.3: Approximate instruction times of extra overhead to synchronize deletion from a priority queue. This overhead does not include the backoff computation. MIPS times are in cycles. 68040 times are based on worst-case number of cycles used by `CAS2` instruction.

4.6.2 Simulation-Based Evaluation

The actual contention for the kernel data structures in our current implementation is low and I do not have the ability to create high contention at this time.

To understand how our system behaves under heavy load, we have simulated insertion/deletion into a singly linked list under loads far heavier than would ever be encountered in the Cache Kernel. (Further simulation and performance tests of various implementations of stacks are described later in this section.)

Our simulation was run on the Proteus simulator [21], simulating 16 processors, a 2K-byte cache with 2 8-byte lines per set, a shared bus, and using the Goodman cache-coherence protocol. All times are reported in cycles from start of test until the last processor finishes executing. Memory latency is modeled at 10 times the cost of a cache reference. The cost of a `DCAS` is modeled at 17 extra cycles above the costs of the necessary memory references. The additional cost of a `CAS` over an unsynchronized instruction referencing shared memory is 9 cycles.

Four algorithms were simulated:

1. `DCAS`/*Cload*: Our `DCAS` algorithm with contention controlled by advisory locking, as implemented on Paradigm and described in Section 5.4.5. The `Cload` instruction is a load instruction that succeeds only if the location being loaded does not have an *advisory lock* set on it, setting the advisory lock when it does succeed.

2. `DCAS`/*A&F*: `DCAS` algorithm with contention controlled by OS intervention as proposed by Allemany and Felten [2] and described in Section 2.7.3.

3. `CAS`: An implementation using only `CAS` and supporting a much higher degree of concurrency based on a technique by Valois [100] [6].

4. *SpinLock*: Spin-lock with exponential back-off as a base case.

Each test performed a total of 10,000 insertions and deletions, divided evenly between all processes. We varied the number of processors from 1 to 16 and the number of processes per processor from 1 to 3. We also controlled the rate of access to the list by each process by doing local "work" between the insertion and deletion. The work varied from 20 to 2000 cycles.

We were careful to implement contention reduction algorithms for all algorithms. First, contention-reduction is implemented in practice on any real system with high contention. Second, we wanted to isolate the algorithmic costs themselves. The contention reduction techniques we used only allowed a single process access to the data structure at a time, allowing other processes access only if the first were stalled. The only exception was Valois', which counts increased concurrency as a feature. For Valois' algorithm we used exponential backoff to reduce contention after a failed `CAS`.

These simulations indicate that the Cache Kernel `DCAS` algorithms perform as well or better than `CAS` or spin locks.

Figure 4.2 shows the performance with 1 process per processor, and minimal work between updates. The basic cost of 10,000 updates is shown at $P = 1$, where all accesses are serialized and there is no synchronization contention or bus contention. At $P = 1$, cache contention due to collisions is small, the hit rate in the cache was over 99% in all algorithms. At more than one processor, even assuming no synchronization contention and no bus contention, completion time is significantly larger because the objects must migrate from the cache of one processor to another.

[6] It was necessary to derive our own version of the algorithm, as the algorithm presented in [100] is not strictly correct. This is the natural result of the complicated contortions necessary when using only `CAS`. In contrast, the `DCAS` algorithm is relatively straightforward.

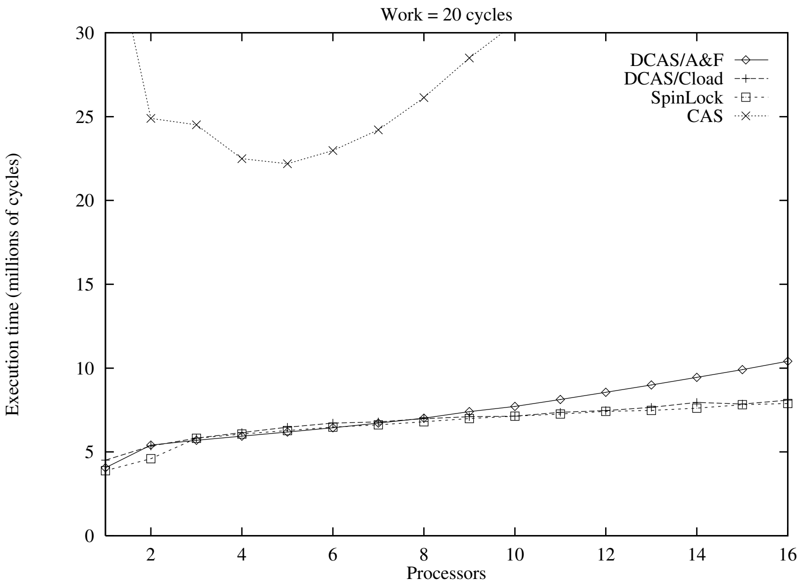

Figure 4.2: Performance of several synchronization algorithms with local work = 20 and the number of processes per processor = 1

In our tests, bus contention remained low due to the contention-reduction protocols I used. When processes/processor = 1 no processes are preempted. In this case the difference between the non-concurrent algorithms is simply the (small) bus contention and the fixed overhead, because we are not modeling page faults[7]. All degrade comparably, although DCAS/A&F suffers slightly from bus-contention on the count of active threads. The Valois algorithm using CAS exploits concurrency as the number of processors increase but the overhead is large relative to the simpler algorithms. The bus and memory contention are so much greater that the concurrency does not gain enough to offset the loss due to overhead. Further, synchronization contention causes the deletion of auxiliary nodes to fail, so the number of nodes traversed increases with a larger number of processes[8]. Our DCAS algorithm performs substantially better than CAS, even granting that the Valois CAS algorithm allows more concurrency.

Figure 4.3 displays the results from reducing the rate of access and interleaving list accesses in parallel with the local work. Insertion/delete pairs appear to take 400 cycles with no cache inter-

[7]Page faults improve the performance of our DCAS algorithms compared to both locks and CAS. My algorithm outperforms the others even without page-faults, thus modeling page-faults is superfluous.

[8]The Valois simulation in Michael and Scott [69] reports better asymptotic behavior than we do. The difference appears because the authors are only simulating a FIFO queue. In the FIFO queue algorithm — where insertion always occurs at the tail and deletion at the head — auxiliary nodes are not traversed in general and thus don't affect completion time. In fully general lists auxiliary nodes increase the execution time and memory traffic.

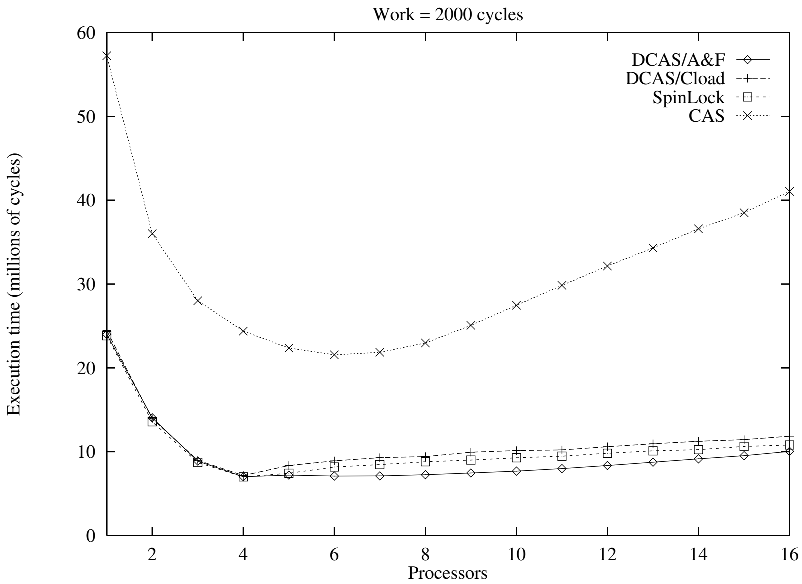

Figure 4.3: Performance of several synchronization algorithms with local work = 2000 and the number of processes per processor = 1

ference so adding 2000 cycles of "local work" lets even the non-concurrent algorithms use about 4 or 5 processors concurrently to do useful work in parallel. Beyond that number of processors, the accesses to the list are serialized, and completion time is dominated by the time to do 10,000 insertion/deletion pairs. `DCAS` with either form of contention control performs comparably to spin-locks in the case of no delays and performance is significantly better than the `CAS`-only algorithm.

Figure 4.4 shows the results when 3 processes run on each processor. In this scenario, processes can be preempted — possibly while holding a lock. As is expected, spin-locks are non-competitive once delays are introduced. In contrast, the non-blocking algorithms are only slightly affected by the preemption compared to the non-preempting runs using the same number of processors. The completion time of `CAS` is still larger than for `DCAS`-based algorithms. While the average time of the `CAS`-based algorithm was mostly unaffected, the variance increased significantly in the case of preemption. Table 4.4 compares the variance between these two experiments.

The increase in variance is not due to delays introduced by one process waiting for another (this is a non-blocking algorithm). Rather, the increase is due to reference counts held by preempted processes occasionally delaying the deletion of nodes — when a process resumes after a delay, it can spend time releasing hundreds of nodes to the free list. Further, when a preempted process holds a reference count to a node, the increased length of the chain of auxiliary nodes increases the time of *all* list accesses, slowing down the processes that are still running.

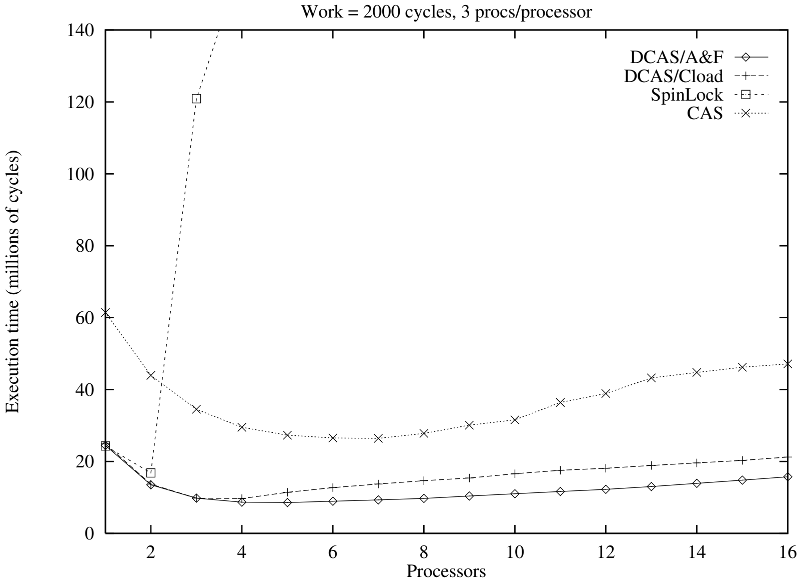

Figure 4.4: Performance of several synchronization algorithms with local work = 2000 and the number of processes per processor = 3

Overall, `DCAS` performs comparably to, or better than, spin locks and `CAS` algorithms. Moreover, the code is considerably simpler than the `CAS` algorithm of Valois.

In these simulations, the number of processors accessing a single data structure is far higher than would occur under real loads and the rate of access to the shared data structure is far higher than one would expect on a real system. As previously noted, contention levels such as these are indicative of a poorly designed system and would have caused us to redesign this data structure. However, they do indicate that our techniques handle stress well.

This experiment, which measured completion time, is different than the experiments later in this chapter and in Chapter 5, where we measure throughput. When we allow multiprogramming and preemption, the Valois algorithm performs very badly. If a process is preempted while traversing the list, the reference counts prevent any auxiliary nodes from being deleted. (In a simple experiment where I terminated a process while it held a reference count, the list continued to grow without bound). This increases the cost to do insertions and deletions in the middle of the list. In the throughput experiments, throughput dropped over time (the longer I ran a test, the lower the average throughput became for Valois), and several processes experienced starvation. In the completion time experiments reported here, as processors completed their alloted insertions and deletions they

N Processors	1 Process/Processor	3 Processes/Processor
1	574.0E9	500.1E9
2	117.4E9	279.3E9
3	101.3E9	145.3E9
4	68.0E9	80.8E9
5	67.6E9	11013.2E9
6	69.9E9	80.8E9
7	85.1E9	3775.6E9
8	82.7E9	2099.6E9
9	102.0E9	11171.8E9
10	97.1E9	114.4E9
11	107.5E9	159.9E9
12	141.6E9	40541.5E9
13	145.6E9	76209.6E9
14	157.7E9	256.5E9
15	179.7E9	307.9E9
16	205.8E9	14461.2E9

Table 4.4: Variance of Valois algorithm, for work=2000, comparing 1 process per processor (no preemption) vs. three processes per processor (preemption).

stopped contending, allowing auxiliary nodes to be deleted, and gradually allowing the entire task to finish.

4.6.3 Measurements of optimized data structures

`DCAS` supports non-blocking implementations of common data structures that out-perform all locking and `CAS`-based algorithms. I report on representative measurements of two simulated implementations of stacks using `DCAS` (one list-based and one array-based) to support this claim. The `DCAS` stack algorithms are described in the Appendix in sections D.2.1 and D.2.2. I compared them against several stack implementations:

- A linked-list implementation of stacks, protected by a spin-lock.
- An array-based implementation of stacks, protected by a spin-lock
- A `CAS`-based algorithm by Treiber[84], which is non-blocking and represents the stack as a linked-list. (My implementation is based on the version of Treiber's algorithm used by Michael and Scott [69]),

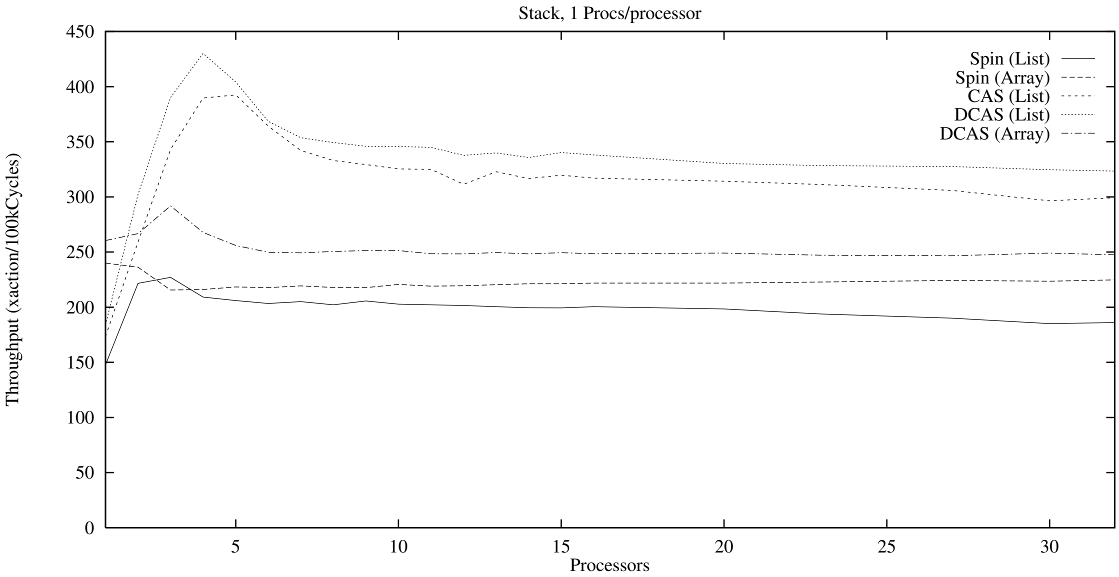

Figure 4.5: Performance of stacks with 1 process per processor

The test environment was designed to be as friendly as possible to the spin-lock implementations. Non-blocking algorithms have significantly better worst-case performance than locking algorithms in the face of long delays. We concentrated instead on the common case when the lock holding process does not experience long delays. Thus, we did not simulate page faults, and had only one process involved in the test running on each processor. (Some tests utilized other background processes, but none ever used more than one process per processor directly accessing the stacks.)

Each process performed a random number $r, 1 \leq r \leq 5$, of `push`es, then looped 10 times to simulate a simple computation, and then `pop`ped off r values. I used the Proteus simulator to simulate a 32 processor multiprocessor, using the same parameters as described in Section 4.6.2. All stack implementations used identical exponential backoff algorithms to implement contention reduction.

The list-based arrays allocated stack-entries from a local free pool; entries migrated from processor to processor because our test allowed each process to pop off entries pushed by other processes. The free pool itself was implemented as a stack.

Array-based stacks used only fixed storage, and therefore did not need to access the free pool.

The computation of the random number, the backoff algorithm, the 10-cycle loop between the `push`es and `pop`s, and (where applicable) the allocation and deallocation of stack entries were all purely local. Therefore, as the number of processors increased this part of the computation could be overlapped. Individual `push`es and `pop`s can only succeed if the stack were in identical state when the operation tried to complete. Generally this would imply that the process had exclusive access to the stack for the entire duration of the operation. Therefore, we would expect to see concurrency increase with the number of processors until the time spent by p processors performing pushes and pops equaled the time spent for 1 processor doing all the overlappable computations. The data in Figure 4.5 is consistent with such a model. All the algorithms (except for array-based stacks using spin-locks, which has no local allocation cost) show increased throughput with the initial increase in the number of processors, eventually settling down as the time is dominated by the sequential access to the stack itself.

One might also expect array-based stacks to perform better than list-based stacks. They do not require allocation and deallocation of stack entries and so incur roughly half the overhead. Measurements show that they do out-perform list-based stacks for both non-blocking and blocking implementations for a small number of processors. Surprisingly, this did not continue to hold true for non-blocking implementations as the number of processors increased. Cache-miss costs begin to dominate the performance of the non-blocking implementations for a reasonable number of processors. In my tests, the array-based stacks typically caused cache-misses on references to both the stack pointer and the top element in the array. These misses occured for both `push` and `pop` operations. List-based stacks caused fewer cache misses. First, they allocate the stack entry from a private per-processor free-pool. All `push` and `pop` operations wrote a new value into `stack->top`. This write ensured that `stack->top` was not in the cache of the next processor that acquired the stack. However, in the `push` operation, the write of `entry->next` almost always hits in the cache because no other process has touched it since it was last `pop`ped. In the `pop` operation, the write of `top->next` to NULL misses in the cache if the immediately preceding operation was a `push` by a different processor. However, if the previous `push` was local, no miss is incurred. If the previous operation was a `pop`, then it did not touch the current stack entry which therefore has a chance being present in the cache.

This pattern does not repeat itself for the lock-based stack implementations. There, the synchronization costs of the lock-based lists continued to be more significant than the cost of the cache misses. Consequently, the array-based stacks continued to outperform the list-based stacks even as the number of processors increased.

Both the `CAS` and `DCAS` non-blocking implementations out-perform the equivalent locking implementations in all circumstances. For common algorithms (locking and Treiber) [69] reported

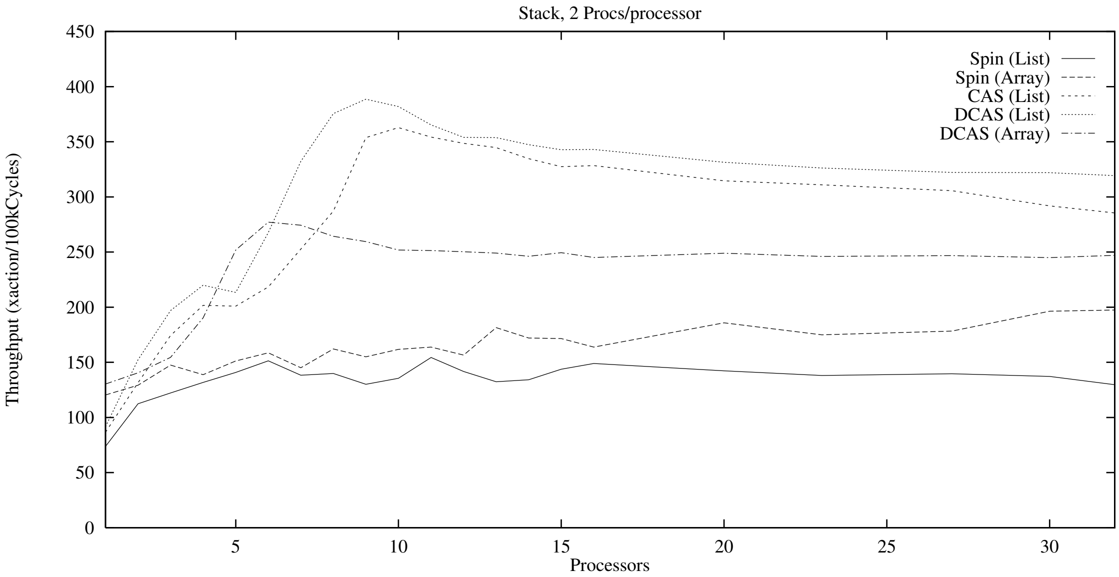

Figure 4.6: Performance of stacks with 1 process per processor, and one background computation

results consistent with ours. That research did not, however, look at algorithms which used DCAS.

The DCAS implementation of list-based stacks out-performs the CAS implementation because DCAS enables a more efficient implementation of pop. When X and Y are the top two elements of the stack, CAS must protect against a contending process popping X, then pushing A, followed by a push of X. A naive implementation of pop would notice that X was the top of the stack. It would incorrectly assume that X was *still* the top of the stack, as opposed to correctly notice that X was the top of the stack *again*. A simple pop at this point would effectively pop both X and A, leaving Y as the new top of stack. Treiber encodes version numbers in the pointers between each stack element to detect this problem. My DCAS implementation only needs to straightforwardly check both `stack->top` and `top->next` before popping, requiring no extra space and no version number computations. It should be noted that this performance advantage is less significant than the advantage of DCAS linked lists mentioned in the previous section. There, multiple CAS's were needed to complete a single update, and therefore performance degraded as the number of processes increased. In contrast, the performance gap in the CAS stack implementation is not exacerbated by a larger number of processes (because the CAS update is still done as a single atomic operation on a double-word)). For stacks implemented as linked lists, DCAS only out-performs CAS by roughly a constant amount as the number of processes increase.

The case of array-based stacks differs markedly. If the number of contending processes is small,

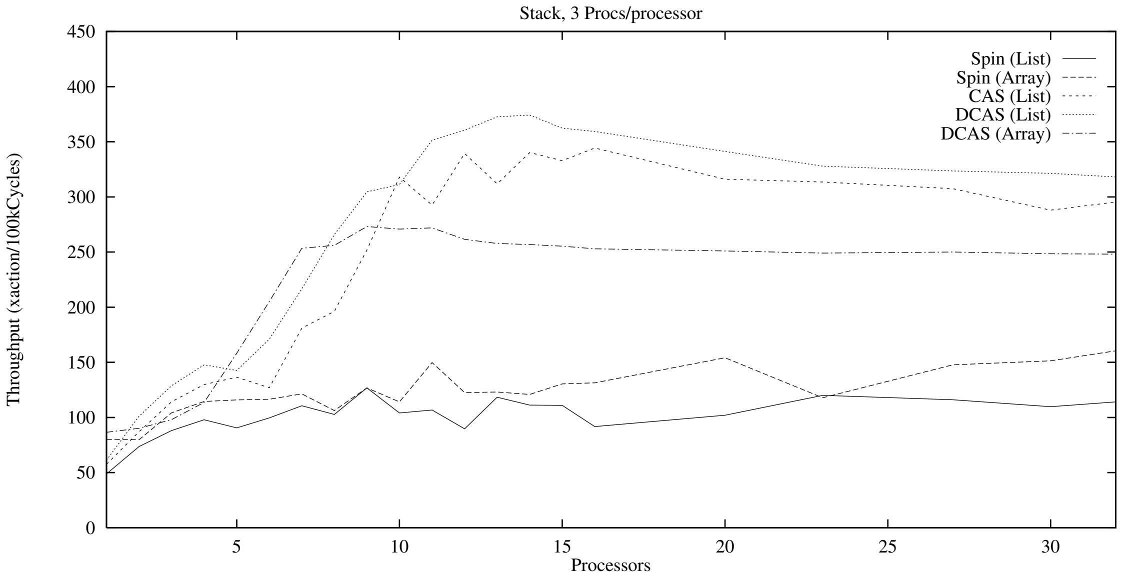

Figure 4.7: Performance of stacks with 1 process per processor, and 2 background computations

then array-based stacks out-perform list-based stacks. In systems where we know contention for a stack is typically low, (which is the common case), we would prefer to use an array-based stack. Unfortunately, no efficient `CAS`-based implementation of stacks exists at all — so on machines without `DCAS` support, non-blocking array-based stacks are not an option. The non-blocking `DCAS` based stack out-performs spin-locks in both average and worst-case time.

4.6.4 Contention Reduction

The results in Section 4.6.2 are one indication of how hardware advisory locking performs compared to operating system support in the style of Allemany and Felten. In the normal case, the lock-holder experiences no delays and the waiters detect immediately when the advisory lock is released. However, when a process is preempted, the waiters *cannot* immediately detect that. When the waiter has backed off beyond a certain maximum threshold, it uses a normal `Load` rather than a `Cload` and no longer waits for the lock-holder. Beyond a moderately large number of processes, the occasional occurrence of this (bounded) delay enables `DCAS`/A&F to outperform the cache-based advisory locking. However, the expected behavior of the Cache Kernel is for the waiters to be on the same processor as the lock-holder (either signal handlers or local context switch). In this case, the advisory lock does not prevent the waiter from making progress. Therefore, there is no advantage to the operating system notification and the lower overhead of advisory locking makes it preferable.

On the other hand, if extremely high contention (relative to the maximum backoff threshold) may occur, the `Cload` approach is unattractive since it does not degrade gracefully. `Cload` delays but does not prevent contention under very high load, because the `Cload` algorithm misinterprets the lengthy wait for a locked cache line as an undetected long delay. After waiting for the maximum backoff interval it uses a regular load to acquire the locked cache line. This defeats the purpose of advisory locking.

The measurements in Section 4.6.2 only indirectly measure the relative performance of the contention reduction algorithms.

I directly compare the costs of several contention reduction algorithms to understand their relative strengths. The algorithms I investigate are (a) simple exponential backoff, (b) hardware (using the `Cload` instruction in conjunction with cache-based advisory locking), (c) the SOLO protocol of Allemany and Felten [2], (d) the "current process oracle" approach from Section 3.2.2, and (e) the approach described there that replaces the oracle with a timestamp-based heuristic. As a base case, I compare the performance of these contention reduction techniques against an algorithm with *no* contention reduction: thus each experiment is run using 6 different techniques.

Note that the `DCAS`-based software contention-reduction protocols proposed in Chapter 3 has the same fault-tolerant properties as hardware advisory locking, with only somewhat higher costs in the case of no contention. It also has the same performance characteristics under high load as the Allemany and Felten approach. If the software implementation has comparable performance, then it is to be preferred over hardware solutions. Finally, the approach that generally worked best (exponential backoff with an entry-timestamp used to distinguish between long delays due to a stalled computation and long-delays due to *many* owners), was *not* strictly non-blocking.

I use each contention reduction algorithm to protect a single `DCAS` operation. Each process uses the `DCAS` to continually update the same fixed pair of memory locations. Each process alternates between a local computation (simulated by an empty loop) and updates to shared data. Each process performs an equal number of "short transactions" and "long transactions" to the shared data structure. For the results reported here, the gap between transactions is 10 iterations of a loop. The time spent in a short transaction after reading the current state and before attempting to update that state with `DCAS` is another 10 cycles. The time spent in the long transactions is 20 iterations; double the time spent in short transactions. Note that these times are short relative to the cost of contention reduction, to synchronization, and to the update.

I use an exponential backoff algorithm with hysteresis. Any process that successfully acquires

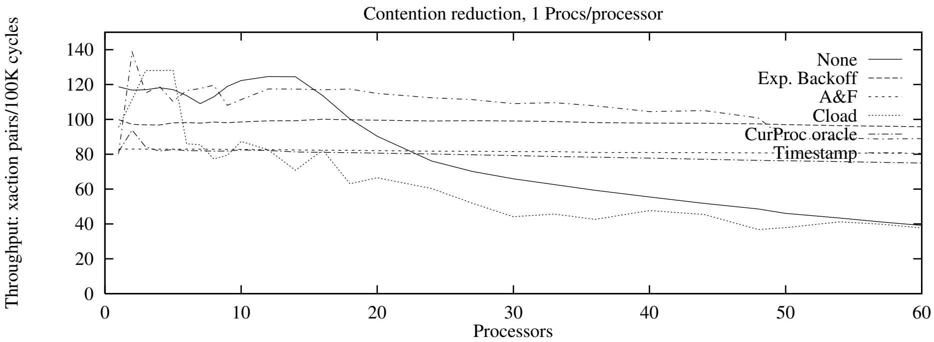

Figure 4.8: Throughput comparison of contention reduction techniques. Number of transaction pairs per 100k cycles.

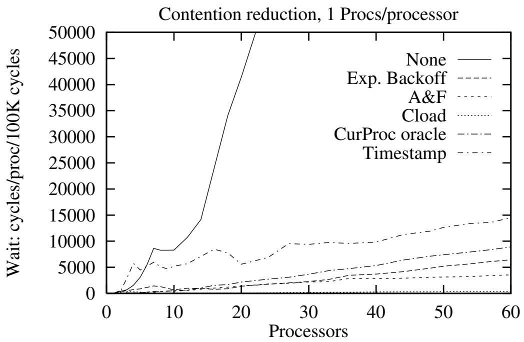

Figure 4.9: Comparing contention reduction techniques: Bus contention (avg. wait per processor per 100k cycles, for access to the bus.)

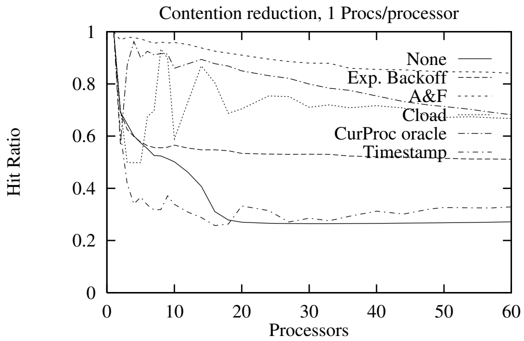

Figure 4.10: Comparison of cache hit ratios for several contention reduction techniques.

ownership of some resource after backing off and waiting, records the time of its last interval inside the resource. If another process subsequently tries to acquire ownership, and fails, the new process must use exponential backoff. The first interval it uses for backoff is one half the interval recorded in the resource (the actual wait time is chosen uniformly from the closed interval `[1..(savedBackoffInterval/2)]`).

Each of the other contention reduction techniques is also coupled with exponential backoff. This avoids a mad rush to acquire the resource when it is released by the previous owner.

The timestamp heuristic uses a simple implementation. Instead of recording a process ID with each owned resource, the process stores a timestamp to signify ownership. If no timestamp is stored in the owner field, then the resource is unowned. Contention reduction is implemented by checking whether `((currentTime - stallTime) > timestamp)`. If so, the waiter assumes the owner is experiencing a long delay, and steals ownership.

I used `savedBackoffInterval` as `stallTime`. Note that `savedBackoffInterval` is not an approximation of the average duration of ownership, but rather the product of the ownership duration times the average number of active waiters. In practice `stallTime` should not be smaller than the granularity of clock synchronization, so there may need to be a minimum value of `stallTime`.

I first compare the contention reduction algorithms using only one process per processor. Thus, there are no long delays because the processes are never preempted. The y-axis of Figure 4.8 shows the throughput of each algorithm, counting the number of successful pairs of transactions (one short, one long) per 100,000 clock cycles. I also show the impact of each algorithm on the rest of the machine. Figure 4.9 shows the average number of cycles per 100,000 bus cycles that each processor spent waiting to acquire the bus. Figure 4.10 shows the cache hit rate as the number of processors increases.

Several observations can be drawn from these graphs.

- For short transactions, it is questionable whether fancy contention reduction algorithms are worthwhile. The lower overhead of simple exponential backoff allows it to perform as well or better than the more complex approaches. (As we shall see, other issues (such as fairness) may render exponential backoff less than ideal.)

- On the other hand, implementing *no* contention reduction whatsoever seems unwise: Although throughput under low contention is not seriously degraded, bus contention increases significantly. Further, the incremental cost of a simple contention reduction scheme is low, it

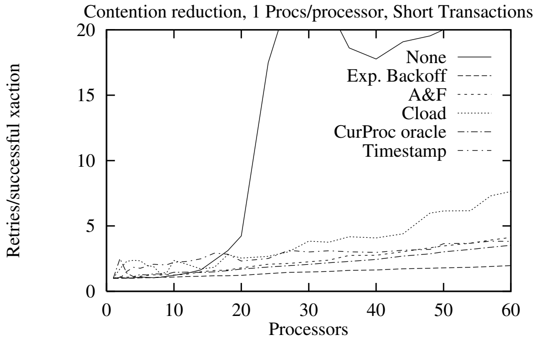

Figure 4.11: Average number of retries for each contention reduction technique. This graph represents the performance of the short transactions.

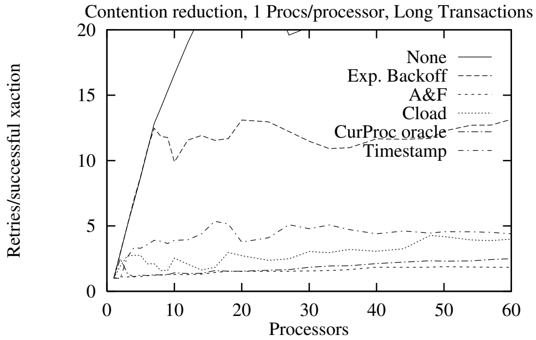

Figure 4.12: Average number of retries for each contention reduction technique. This graph represents the performance of the long transactions.

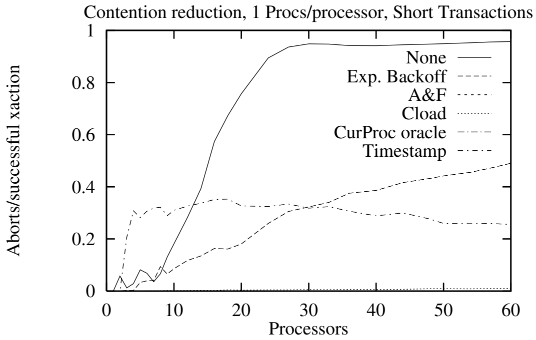

Figure 4.13: Average number of failing DCAS's (which means the work was wasted). Comparison between different contention reduction techniques. This data is for short transactions.

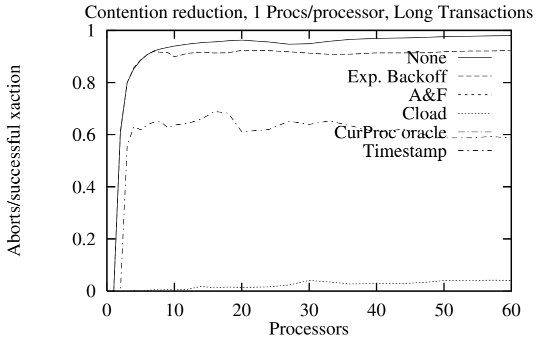

Figure 4.14: Average number of failing DCAS's (which means the work was wasted). Comparison between different contention reduction techniques. This data is for long transactions.

provides protection in the (unusual) event of high contention, and the performance degradation due to high contention can be severe.

- `Cload` (hardware advisory locking) performed very well for a very small number of processors, but eventually the long lock-wait time under contention made `Cload` perform as poorly as no contention reduction at all. Waiters detect long lock-wait times but cannot detect that the cache line was locked by many processors each for a short period of time. Hence, under high contention waiters eventually steal locked cache lines.

  This was not the only problem with `Cload`. `Cload` tended to perform inconsistently. There is some evidence to suggest that some of its erratic behavior was caused by locked cache lines being evicted from the local cache due to conflict misses (not coherency misses). Further investigation is necessary to verify whether this is the real explanation for `Cload`'s inconsistent performance.

- The timestamp heuristic performed better than I expected. The overhead was comparable to simple exponential backoff, yet timestamps were more accurate in reducing contention. In particular, when the number of processes increases, timestamps allow waiters to distinguish between different cause of long waits. A single process may be stalled while owning a data structure, or *many* processes may each own a data structure briefly. Timestamps allow waiters to avoid interfering or aborting when progress is being made.

Figures 4.11 through 4.14 display measurements which allow us to asses fairness between long and short transactions. The first two graphs show the number of retries per successful transaction. The y-axis counts the average number of times a process looped before executing a successful `DCAS`. Not all retries involved attempts; the contention reduction algorithm may suggest skipping the attempt because the data structure is already owned. Figure 4.15 gives a pseudo-code framework defining `retries` and `aborts`. The first two graphs report `retries/successes`. The second two graphs report `aborts/attempts`.

Exponential backoff penalizes the longer transactions, even though the absolute difference between short and long transactions is negligible. The average number of retries for short transactions is approximately 1. The average number of retries for long transactions is over 10! In addition to the unfairness, the number of retries represent wasted processor cycles. 30% of short transactions are aborted. 90% of long transactions are aborted. Each attempt involves bus communication and cache interference.

```
while (workToDo())
{
  do {
    skip = 1;
    retries++;
    backoffIfNeeded();
    if (reduce_contention()) {
      continue;
    }
    skip = 0;
    attempts++;
  } while ((skip==1) ||
           (updateSuccessful?()))
  successes++;
}
aborts = attempts - successes;
```

Figure 4.15: Pseudo-code explanation of “retries” and “aborts”.

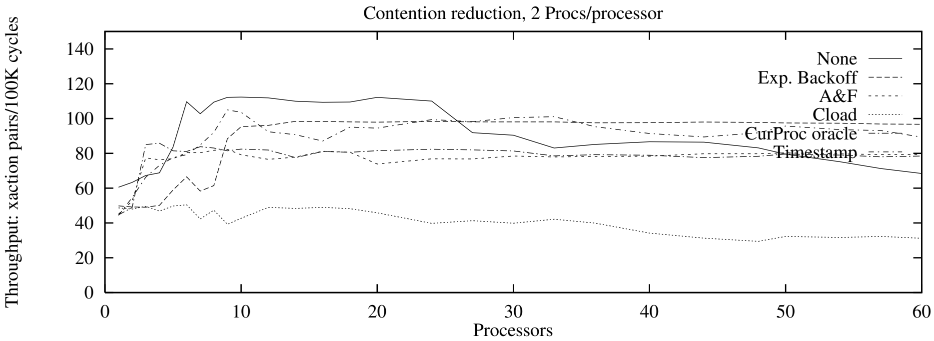

Figure 4.16: Throughput comparison of contention reduction techniques, with background process. Number of transaction pairs per 100k cycles.

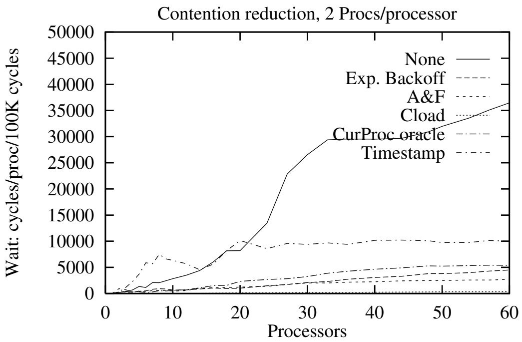

Figure 4.17: Bus contention (wait per proc. per 100k cycles) while comparing contention reduction techniques with a contending process and a background process on each processor.

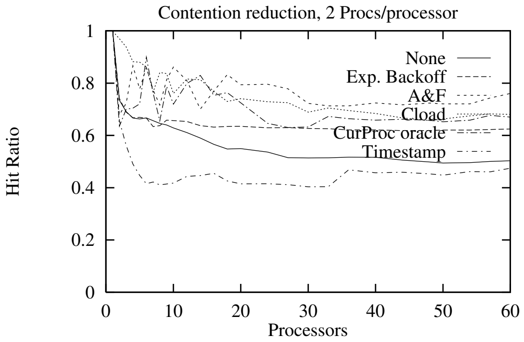

Figure 4.18: Comparison of cache hit ratios for several contention reduction techniques in the presence of background computations.

In contrast, Allemany and Felten's approach and the current process oracle have essentially *no* aborts. The advisory lock protects the data structure, so most processes attempting to access it have exclusive access to it. Aborts are only possible if a process is preempted while holding the lock. `Cload` can be aborted when the max threshold is reached, but this is relatively infrequent. Timestamps are more prone to aborts, but they are still 2 to 3 times less likely to be aborted than exponential backoff.

The number of retries is determined by the backoff algorithm, which *should* be identical for all approaches. The variation in number of retries between the different approaches is due to differences in performance due to bus contention and cache misses — the quicker each retry, the more chances to retry it had, but the cumulative wait *time* remained very similar.

Figures 4.16 through 4.29 show the results of the same tests when run with 1 or 2 background processes per processor. One would expect the behavior to be similar to the earlier graphs, with performance comparable to performance in the original graphs at points equivalent to $p/2$ and $p/3$, respectively. (The simulated algorithms are all non-blocking, therefore performance should be roughly impervious to the long delays, but *should* reflect the fact that only half (or 1/3) the processors are actively engaged in the shared computation.) This seems to be the case for most of the algorithms. Exponential backoff, Allemany and Felten, and the current process oracle seem to be unaffected by the long delays and do show the predicted flattening of performance.

However, the performance of `Cload` suffers disproportionately. Partly this is due to the factors listed above, which are exacerbated in systems where other processes are running. However these

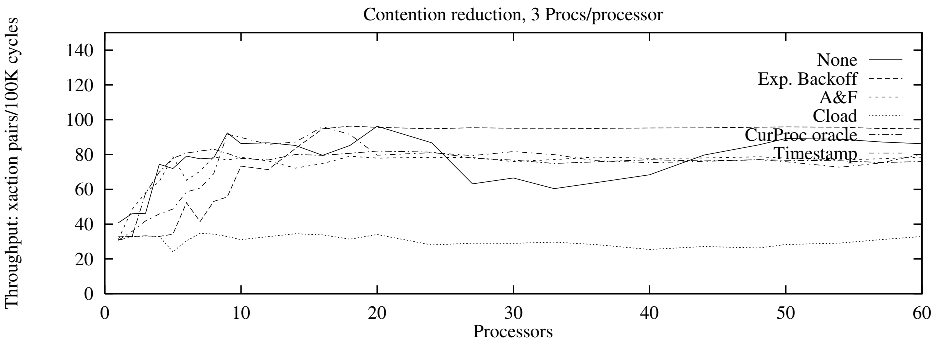

Figure 4.19: Throughput comparison of contention reduction techniques. Number of transaction pairs per 100k cycles.

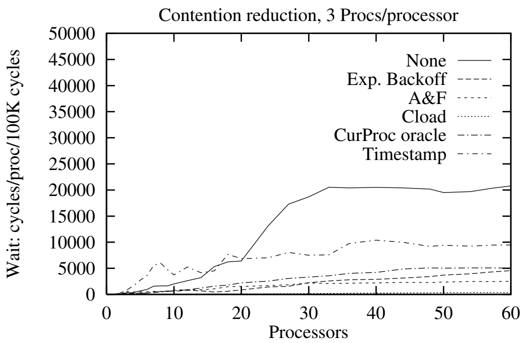

Figure 4.20: Bus contention.

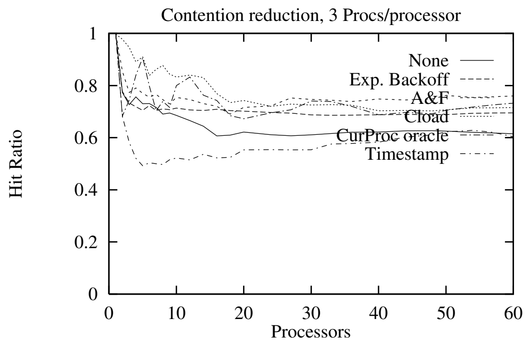

Figure 4.21: Comparison of cache hit ratios of several contention reduction techniques.

factors do not seem sufficient to fully explain the measured performance. Further investigation is needed to understand this behavior better.

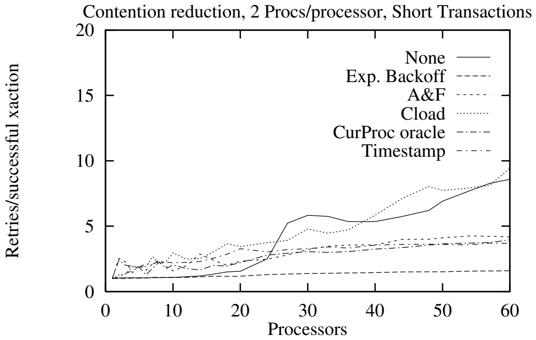

Figure 4.22: Average number of retries for each contention reduction technique. This graph represents the performance of the short transactions.

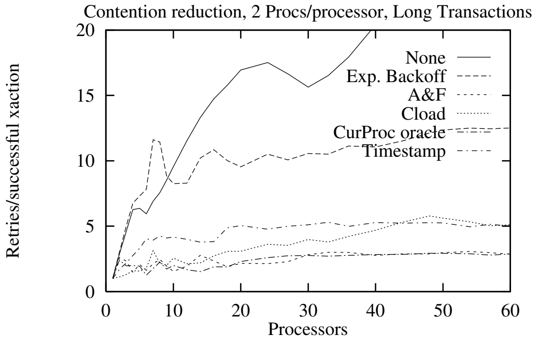

Figure 4.23: Average number of retries for each contention reduction technique. This graph represents the performance of the long transactions.

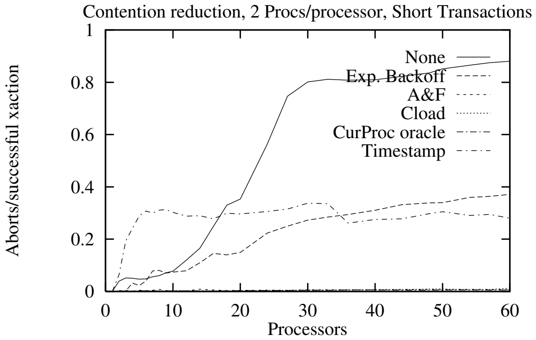

Figure 4.24: Average number of failing DCAS's. Comparison between different contention reduction techniques. This data is for short transactions.

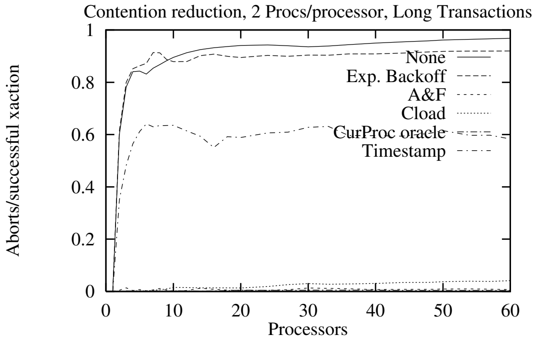

Figure 4.25: Average number of failing DCAS's. Comparison between different contention reduction techniques. This data is for long transactions.

One might also expect timestamps to behave in unpredictable fashion. Unlike Allemany and Felten and unlike the current process oracle, timestamps are just a heuristic method of detecting long delays. However, long delays are infrequent, and the heuristic is actually quite reasonable. Timestamps perform about as well in the face of background processes as they do in the original case of one process per processor. Furthermore, their overhead is comparable to exponential backoff, and significantly less than advisory locks (whether using a current process oracle or the technique of Allemany and Felten). They exhibit slight unfairness with respect to longer transactions, but this unfairness is on the order of 50% longer wait times, as compared to the factor of 10 exhibited by pure exponential backoff.

The main problem with timestamps as a method of contention reduction is that they are not strictly non-blocking. Further experiments are required to understand their behavior when transaction times have high variance. Given that long delays are rare, it seems reasonable to investigate increasing the threshold for stealing ownership to some multiple of `savedBackoffInterval` (e.g. `((currentTime - K*savedBackoffInterval) > timestamp)`).

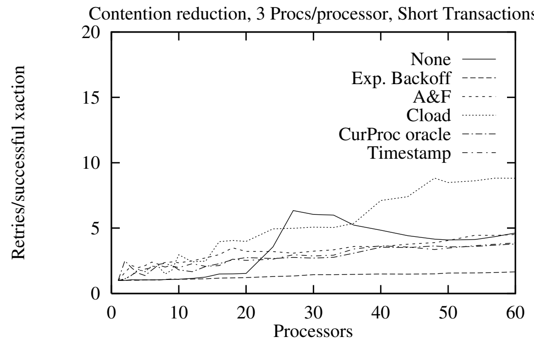

Figure 4.26: Average number of retries for each contention reduction technique. This graph represents the performance of the short transactions.

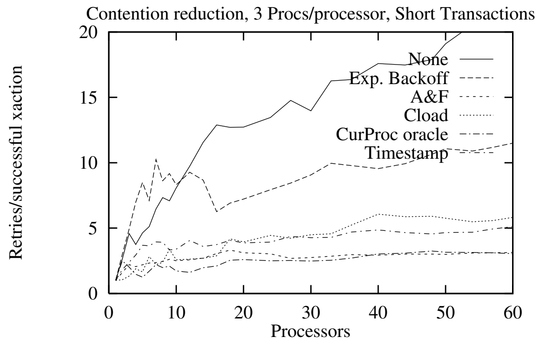

Figure 4.27: Average number of retries for each contention reduction technique. This graph represents the performance of the long transactions.

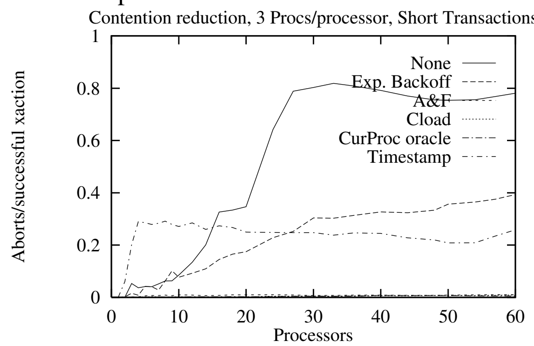

Figure 4.28: Average number of failing `DCAS`'s. Comparison between different contention reduction techniques. This data is for short transactions.

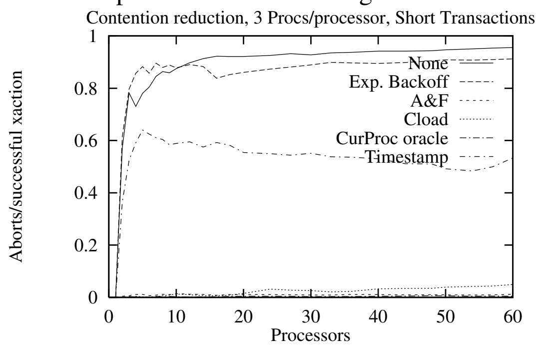

Figure 4.29: Average number of failing `DCAS`'s. Comparison between different contention reduction techniques. This data is for long transactions.

One final note seems pertinent on the relative performance of the contention reduction schemes. The effect of aborts versus retries appears to be negligible. This is somewhat surprising. One of the arguments [2] in favor of advisory locks over retry (with exponential backoff) was that the advisory locks reduced aborts. Aborts (failing `CAS` operations) were believed to cause more contention than simply retrying because a lock was held. An aborted transaction performed the transaction and failed on the commit. This increases bus traffic and increases the likelihood that another transaction will take a cache miss on accesses to shared memory. A retried transaction halts before even attempting the transaction, and so introduces less bus traffic and doesn't contend for shared memory (except perhaps for the words implementing the advisory lock).

However, in my tests aborts did not seem to have a significant adverse impact compared to retries. This is because in most tests the bus was not close to saturation, so the very slight increase in bus traffic only marginally increases bus wait times. Cache misses are more relevant. However the relevant contention is for a single cache line (else there's little increase in cache misses on remote processors). At a given level of contention for a single cache line, all contending processes are equally likely to access those memory locations, and the successful process almost always missed in the cache regardless of whether contending processes retried or aborted — the cache miss was due to the last *successful* transaction, which was usually on a different processor.

It is unclear whether aborts are ever a significant performance issue for specific real applications. It is clear that if they are a concern, it is only for a particular set of applications. For those applications, we need to be careful before reducing the number of aborts, or choosing a contention reduction algorithm that does not eliminate aborts.

The heuristic methods (exponential backoff, timestamp, and `Cload`(when the cache-line is evicted)) all experienced aborts. If a real application actually experienced a performance loss due to the aborts, then the number of aborts (relative to the number of retries) must be reduced. To reduce the number of aborts, a more conservative approach must be taken before re-attempting to perform an update. However, attempting a longer wait-period for exponential backoff unacceptably increases latency even in the case of no long delays. In contrast, the timestamp and `Cload` algorithms can increase the threshold at which they steal ownership without reducing the rate of retries — if ownership is relinquished then there is no latency penalty in the common case of no preemption (the waiters proceed as soon as they detect ownership is relinquished).

4.6.5 Overall System Performance

We do not have the ideal measurements to show the benefit of non-blocking synchronization for overall system performance. (There is no blocking version of the entire system, as that would have entailed many other design changes). However, in other work [103], system performance has been shown to benefit considerably from the ability to execute code in signal handlers as exploited extensively by the Cache Kernel object-oriented remote procedure call system. This system allows restricted procedures, namely those that do not block, to be executed directly as part of the signal handler invocation that handles a new call. With this optimization, many performance-critical RPCs can be invoked directly in the signal handler without the overhead of allocating and dispatching a separate thread to execute the RPC. Our measurements, reported in the cited paper, indicate a significant savings from this optimization, particularly for short-execution calls that are common to operating system services and simulations.

4.7 Related Work

Previous work has explored lock-free operating systems implementations and general techniques for non-blocking and wait-free concurrent data structures. Very little work has explored optimizations and simplifications to NBS algorithms by methodically exploiting properties of systems. Some work, however, has exploited algorithm-specific details to optimize specific NBS implementations.

4.7.1 Lock-Free Operating Systems

Massalin and Pu [64] describe the lock-free (non-blocking) implementation of the Synthesis V.1 multiprocessor kernel, using just `CAS` and `DCAS`, as we did. Their work supports our contention that `DCAS` is sufficient for the practical implementation of large systems using non-blocking synchronization. In particular, although they initially prototyped non-blocking implementations of particular data structures using a "universal" construction (implementing the object as a server process, and enqueuing operations and dequeuing results), they report that, ultimately, `DCAS` was sufficient to design custom non-blocking implementations of *all* their data-structures. No server processes were needed in their final release. This confirms our experience that, given `DCAS`, universal constructions are not needed for shared data structures.

However, the Synthesis work focused on using a small number of predesigned wait-free and lock-free data structures ("qua-jects") inside their operating system kernel. One reason their work has not been further emulated is their exploitation of application-specific optimizations to implement data structures. One example is their implementation of a linked list with insertion and deletion from the middle of the list: it is efficient only because the usage within the Synthesis kernel is highly constrained and a single bit suffices where a reference count is normally needed. The precise properties needed by the application to support this are unclear. In contrast, our implementation of linked lists is general, and is usable by arbitrary application code, assuming they support and that `DCAS` is available.

4.7.2 General Methodologies for Implementing Concurrent Data Objects

Barnes [14], Turek [99], Valois [100], Israeli and Rappaport [46, 47], and others provide techniques for increasing the concurrency with non-blocking synchronization. However, these concurrent updates have noticeable algorithmic cost and often increase memory contention. These costs appear to outweigh the actual benefits, because of the low rates of contention in our system. Studies such as [98, 67], which also reported a low level of contention on kernel and application data structures, respectively, suggest that low contention might be the norm in more systems than just in the Cache Kernel.

4.7.3 Exploiting system properties to support NBS

Anderson et. al. [81, 5, 7] have investigated non-blocking synchronization in real-time systems. Like our work, they simplify and optimize non-blocking algorithms by exploiting properties of real-time systems. In contrast to our work, the properties they exploit are more narrowly applicable to systems with real-time properties. Some of these properties and assumptions are unrealistic in general systems. For example, they depend upon a strict priority order — that is, a lower priority process *never* runs while a higher priority process is computing. This eliminates multi-programming across page faults. In their real-time systems (which have no page faults) such restrictions are not a problem. In general, however, such restrictions limit the utility of their work.

In a similar vein, Johnson [52] exploits the serial nature of uniprocessor systems to implement efficient non-blocking algorithms suitable for real-time and embedded systems. His *interruptible critical sections* are a form of universal construction requiring no special primitives, minimal operating system support, and relatively efficient performance. However, it is not generalizable to multiprocessor systems where multiple processes may execute concurrently.

My work tries only to utilize properties of systems that have general utility and benefits independent of non-blocking synchronization.

Blumofe [11] and Chesson [28], among others, exploit client-specific properties in order to implement non-blocking data structures. Blumofe's work takes advantage of the fact that in the work-stealing algorithm, only a single processor enqueues or dequeues at the bottom of a deque, and that other processors only `pop` from the top of the deque, never `push`. This effectively eliminates the need for most synchronization, and allows a non-blocking implementation of a deque using only `CAS`. Similarly, Chesson exploits the fact that many queues are multi-writer/single-reader to implement efficient non-blocking implementations using only single `CAS`.

While their work is interesting in overcoming the limitations of systems that support only `CAS`, they depend upon properties which are somewhat less generally applicable than the techniques upon which we depend. Further, `DCAS` and TSM enable non-blocking implementations of their data structures in the fully general case, imposing no limitations on clients.

4.8 Conclusions

This chapter showed how careful design and implementation of operating system software for efficiency, reliability and modularity makes implementing simple, efficient non-blocking synchronization far easier. In particular, I identify type-stable memory management (TSM), contention-minimizing data structuring and minimal inconsistency window structuring as important for all these reasons. These techniques are beneficial even with blocking synchronization and yet, together with efficient `DCAS`, significantly reduce the complexity and improve the performance of non-blocking synchronization.

In this chapter (along with supporting algorithms in Appendix D) I demonstrated how each of these features simplifies or improves NBS algorithms. I showed a number of techniques for implementing non-blocking synchronization using TSM, version numbers and `DCAS`. (Appendix D more fully describes these techniques, as well as others.) In contrast to single `CAS` without TSM, these techniques are simple to write, read, and understand, and perform well. Our experience suggests that good `DCAS` support is sufficient for a practical non-blocking OS and run-time system implementation, and that single `CAS` is not sufficient. In fact, lack of efficient `DCAS` support in systems is a potential impediment to using our techniques.

Fortunately, the hardware implementation I propose in the next chapter indicates that it is feasible to implement efficient `DCAS` functionality in a modern processor with minimal additional complexity and full compatibility with the load-store architecture.

Our Cache Kernel experience demonstrates that non-blocking synchronization (NBS) is practical as the sole coordination mechanism in a well-designed system, provided that programmers can safely exploit properties of that system. I have identified several important properties useful for NBS implementation. By explicitly identifying these properties, system designers can be careful not to violate them, and programmers can then safely write NBS code depending on these properties being true.

Non-blocking synchronization significantly reduces the complexity and improves the performance of software in the signal-centric design of the Cache Kernel and its associated libraries, especially with the large amount of conventional operating system functionality that is implemented at the library, rather than kernel, level.

Our experience suggests that there is a powerful synergy between non-blocking synchronization and several good structuring techniques for the design and implementation of an operating system and supporting run-time libraries.

Chapter 5

A Hardware Implementation of DCAS Functionality

5.1 Introduction

Most current hardware and operating systems do not provide adequate support to make non-blocking synchronization practical. In particular, the single Compare-and-Swap (CAS) (or, equivalently, load-linked/store-conditional (LL/SC)) provided on many current processors, although universal, is not sufficient because commonly needed data-structures, such as linked-lists, priority queues, and hash tables [34] cannot be efficiently implemented non-blocking with CAS alone[1].

Double-Compare-and-Swap (DCAS) functionality (a 2 location version of CAS, see Figure 5.1) *is* sufficient to implement non-blocking versions of all needed data-structures in real systems. Moreover, experience with two OS's, the Cache Kernel [24, 34] and the Synthesis kernel [64], further support its sufficiency in practice.

In this chapter, I argue that a simple architectural extension to support double CAS is in fact the ideal approach to supporting software synchronization, based on the benefits of non-blocking synchronization, the necessity and sufficiency of DCAS, and the cost-benefits of hardware support over software-only provision of DCAS functionality. I demonstrate that DCAS functionality can be

[1] "Efficient" is defined as in Chapter 3. Universality, in this context, means only that non-blocking algorithms for all data structures are *possible* using only CAS, not that they are practical or efficient. The practical insufficiency of universal primitives is not a new phenomenon: although a Turing machine is universal (in a slightly different sense of universal), it is clearly not a practically efficient design for a real computer.

```
int DCAS(int *addr1, int *addr2,
         int old1,   int old2,
         int new1,   int new2)
{
 <begin atomic>
 if ((*addr1 == old1) && (*addr2 == old2)) {
   *addr1 = new1;  *addr2 = new2;
   return(TRUE);
 } else {
   return(FALSE);
 }
 <end atomic>
}
```

Figure 5.1: Pseudo-code definition of `DCAS` (`Double-Compare-and-Swap`)

provided as a simple extension of current modern processor instruction sets, relying on operating system and application support for starvation avoidance. I argue that this extension is simple to implement, builds on conventional processor cache coherency mechanisms, and provides significant advantages over the best known software-only approaches. I also argue that this implementation is significantly simpler than more extensive hardware extensions for non-blocking synchronization that others have proposed. Consequently, the architectural support for `DCAS` I propose makes the benefits of non-blocking synchronization available to real software systems and applications with the minimal cost and maximal benefit.

The next section describes extensions to an instruction set supporting `Load-linked/Store--Conditional`. Section 5.3 describes the implementation of these extensions, using the MIPS R4000 processor as a specific example implementation. Section 5.4 discusses issues that affected choices among alternative designs. Section 5.5 compares this approach to a software-only implementation and Section 5.6 compares my design to other proposed hardware extensions.

5.2 Extensions to `LL/SC` Instructions

`DCAS` can be implemented on processors that support `load-linked/-store-conditional` by adding two new instructions:

1. `LLP` (load-linked-pipelined): load and link to a second address after an `LL`. This `load` is linked to the following SCP.

LLbit	A single bit of state to specify synchronization instructions. Set by `LL`, cleared by `ERET` and `Invalidate` and read by `SCP` and `SC`.
SCPlast	Register containing the vaddr of the last `SCP` instruction, or all 1's. Set by `SCP` only if `LLbit` is set. Set to all 1's during the WB stage of every instruction other than `SCP`.
LLaddr	System Control Processor (CP0) register 17. Contains the physical address read by the most recent `LL` instruction. For diagnostic purposes only.
LLaddr2	Contains the physical address read by the most recent `LLP` instruction. For diagnostic purposes only.
SCP-Buffer	Stores the contents of rt (DATA) of the last `SCP` instruction, pending a successful `SC`. Set by `SCP` only if `LLbit` is set.

Table 5.1: State used by processor and primary cache controller in implementation of `LLP/SCP`

$\text{vAddr} \leftarrow ((\text{offset}_{15})^{16}$ —— $\text{offset}_{15..0}) + \text{GPR[base]}$
(pAddr,uncached) $\leftarrow$ AddressTranslation(vAddr, DATA)
mem $\leftarrow$ LoadMemory(uncached,WORD,pAddr,vAddr,DATA)
GPR[rt] $\leftarrow$ mem
LLbit $\leftarrow$ 1
LLaddr $\leftarrow$ paddr
SyncOperation()

Figure 5.2: High level language description of `LL rt,offset(base).`

2. `SCP` (store-conditional-pipelined): Store to the specified location provided that no modifications have been made to either of the memory cells designated by *either* of the most recent LL and LLP instructions and these cache lines have not been invalidated in the cache of the processor performing the `SCP`.

If an `LLP/SCP` sequence nested within an `LL/SC` pair fails, the outer `LL/SC` pair fails too.

The R4000 "links" the `LOAD` specified by the Load Linked instruction (`LL`) to the `STORE` issued by the subsequent `SC` instruction. Each processor maintains a single link bit, `LLbit`, to conditionally issue the `STORE` in the `SC` instruction. The `LLbit` is set to 1 by the `LL` instruction, and the address is stored in `LLAddr`. If the `LLbit` is cleared to 0 before the execution of `SC`, the

SC fails and no STORE is issued.

vAddr $\leftarrow$ ((offset$_{15}$)^{16} —— offset$_{15..0}$) + GPR[base]
(pAddr,uncached) $\leftarrow$ AddressTranslation(vAddr, DATA)
mem $\leftarrow$ LoadMemory(uncached,WORD,pAddr,vAddr,DATA)
GPR[rt] $\leftarrow$ mem
LLaddr2 $\leftarrow$ paddr
SyncOperation()

Figure 5.3: High level language description of `LLP rt,offset(base)`.

The LLP's load similarly establishes a "link" with the subsequent SCP instruction, recording the address passed to LLP in LLAddr2. It does not change the LLbit. The SCP STORE is a no-op if the LLbit is clear when this instruction is executed. If the LLbit is set, the STORE is buffered until after the enclosing SC completes its STORE.

vAddr $\leftarrow$ ((offset$_{15}$)^{16} —— offset$_{15..0}$) + GPR[base]
(pAddr,uncached) $\leftarrow$ AddressTranslation(vAddr, DATA)
data $\leftarrow$ GPR[rt]
if LLbit then
SCP-Buffer $\leftarrow$ data
SCPlast $\leftarrow$ vaddr
endif

Figure 5.4: High level language description of `SCP rt,offset(base)`.

The LLbit can be cleared, causing SC to fail, in the following circumstances.

If the physical addresses translated from (T0) or (T1) are modified then the LLbit is cleared[2]. The LLbit is also cleared when an external request changes the state of the cache line. If an ERET[3] is executed (locally), then the LLbit is cleared, and the SC fails. Additionally, any instruction other than SC that follows an SCP clears the LLbit, and causes the next SC to fail (unless reset by another LL).

[2]Following the example of the R4000, we leave the following case unspecified. If T0 or T1 are *not* modified, we sidestep the question of whether SC will *necessarily* succeed. This leaves open the simplification of clearing the LLbit (described in the R4000 manual) on *any* write or cache-miss, causing a "spurious" failure.

[3]ERET, or "Exception Return", is the R4000 instruction used to return from an interrupt, exception, or error trap.

In the absence of LLP/SCP, the semantics of LL and SC remain unchanged from the current R4000 ISA definitions.

vAddr ← $((\text{offset}_{15})^{16}$ —— $\text{offset}_{15..0})$ + GPR[base]
(pAddr,uncached) ← AddressTranslation(vAddr, DATA)
data ← GPR[rt]
if LLbit then
  CACHE(Freeze,offset(base)) StoreMemory(uncached,WORD,pAddr,vAddr,DATA)
  if (SCPlast != -1) then
    StoreMemory(uncached,WORD,LLaddr2,SCPlast,SCP-Buffer)
  endif
endif
GPR[rt] ← 0^{31}——LLbit
SyncOperation()

Figure 5.5: High level language description of `SC rt,offset(base)`. When SCPlast is not set, the behavior is unmodified from the current R4000 implementation.

5.2.1 Implementing DCAS out of LLP/SCP

Double-Compare-and-Swap is then implemented by the instruction sequence shown in Figure 5.6 (using R4000 instructions in addition to the LL/SC(P) instructions). The LL and LLP instructions in lines 1 and 2 "link" the loads with the respective stores issued by the following SC and SCP instructions. Lines 3 and 4 verify that (T0) and (T1) contain V0 and V1, respectively. The SCP and SC in lines 5 and 6 are conditional. They will not issue the stores unless (T0) and (T1) have been unchanged since lines 1 and 2. This guarantees that the results of CAS in lines 3 and 4 are still valid at line 6, or else the SC fails. Further, the store issued by a successful SCP is buffered pending a successful SC. Thus, SC in line 6 writes U1 and U0 to (T1) and (T0) atomically with the comparison to V0 and V1. (Given data structures that are protected by a version number, the DCAS is actually a Compare-And-Double-Swap (CDS) — the second value cannot have changed if the version number is unchanged. In these cases a minor optimization is possible and line 4 can be deleted.)

A distinct LLP instruction is necessary because LL must set the LLbit and LLP cannot touch the LLbit. We cannot simply nest two LLs. If we were to do so, it would be impossible to distinguish the proper behavior of the second LL. For example, consider (in a system with no distinct

```
/*
 * If (T0) == V0, and (T1) == V1, then
 * atomically store U0 and U1 in T0 and T1
 */
DCAS(T0, T1, V0, V1, U0, U1)
   ;; Get contents of addresses in registers.
1  LL        T3, (T1)
2  LLP       T2, (T0)
   ;; Compare to V0 and V1. If unequal, fail.
3  BNE       T2, V0, FAIL
4  BNE       T3, V1, FAIL
   ;; If equal, and unchanged since LOAD, store
   ;;     new values
5  SCP       U0, (T0)
6  SC        U1, (T1)
   ;; Success of SC and SCP is stored in U1
7  BLEZ      U1, FAIL
  ...
FAIL:
```

Figure 5.6: `DCAS` implementation using `LL/SC` and `LLP/SCP`. Success or failure of `SC` (and thus of the `DCAS` operation) is returned in U1 or whatever general register holds the argument to `SC`. 1 denotes success, 0 failure. If the next instruction tries to read U1, the hardware interlocks (as it already does for `LL/SC`) if the result of `SC` is not already in U1.

`LLP`) the behavior of a failed `Compare-and-Swap` (`LL`, `BNE ... FAIL`) followed immediately by a succesful `Compare-and-Swap` (`LL, SC`) versus the case of a single `Double-Compare-and-Swap` in Figure 5.7. Both issue two `LL`s to the processor with no intervening `SC`s. However, in the first case both `LL`s (at lines 1 and 3) must set the `LLbit`, while in the second case the second `LL` (at line 2) should not change the state of the `LLbit`.

`LL/SC` is available on MIPS, ALPHA, and PowerPC processors, among others. Some machines (e.g. Pentium, SPARC v9, 680x0) support `CAS` directly. The 68040 supported DCAS directly by the `CAS2` instruction. It seems feasible to extend `Compare-and-Swap` to `Double-Compare--and-Swap` using the principles described in Section 5.4, as long as the tight coupling between processor and cache controller is present. In this thesis I concentrate solely on `LL/SC`.

5.3 R4000 Multi-processor Implementation

The implementation of the LLP/SCP instructions is described as an extension of the MIPS R4000 processor. This implementation could be adapted to other processors with some minor changes.

```
1  LL        T3, (T1)
2  BNE       T3, V0, FAIL
   ...
FAIL:
   ...
3  LL        T2, (T0)
4  BNE       T2, V1, FAIL2
5  SC        U1, (T0)
6  BLEZ      U1, FAIL2
  ...
FAIL2:
```

```
1  LL        T3, (T1)
2  LL        T2, (T0)
3  BNE       T2, V0, FAIL
4  BNE       T3, V1, FAIL
5  SCP       U0, (T0)
6  SC        U1, (T1)
7  BLEZ      U1, FAIL
  ...
FAIL:
```

Figure 5.7: Example showing why a distinct LLP is needed in addition to LL. These two sequences assume LLP doesn't exist. They are indistinguishable at the point the second LL is executed, but the later instructions show that the two LL's should behave very differently.

Familiarity with the R4000 ISA is assumed.

Implementing LLP/SCP has two requirements: (a) First, the SC and SCP either both succeed or both fail, and (b) if successful, the words written by the SC and the SCP appear to be simultaneously updated to any processor reading them.

5.3.1 Atomically Storing Two Values

Atomicity is ensured in two ways, depending on the tight coupling between cache controllers and processors that is becoming more common.

First, the store of the successful SCP is deferred until the second store (the SC) has also succeeded. The processor buffers the first store in a special processor register: SCP-Buffer.

Second, the processor does not relinquish *ownership* of the cache lines in question between executing the SC and SCP, until *both* updates have finished. The processor will not commit these updates unless it held ownership of the cache lines from the time of the LL(P) until the SC instruction. "Ownership" has different meanings in different cache coherency protocols. We require the processor to "own" the cache lines in the following senses:

- Other processors must request the value of the cache line from this processor in order to obtain the value.
- This processor must be notified if it has lost possession of the cache lines (causing the SC to fail).
- It must be possible to defer responding to a read or ownership request for the duration of one SC (strictly speaking, for the duration of the MEM pipeline stage, not the entire instruction),

without losing ownership of either cache line. (An alternative is to respond with the old value of the cache line, and abort the LL/SC LLP/SCP: flush our change and return. This alternative, although workable, seems more complicated.)

The atomicity is protected by the cache controller: no processor (including the processor performing the `LL/SC`) can see an intermediate result of the `SC/SCP`. The processor guarantees this by executing the following sequence of steps[4] (see Figure 5.8): It always either completely succeeds, or completely fails.

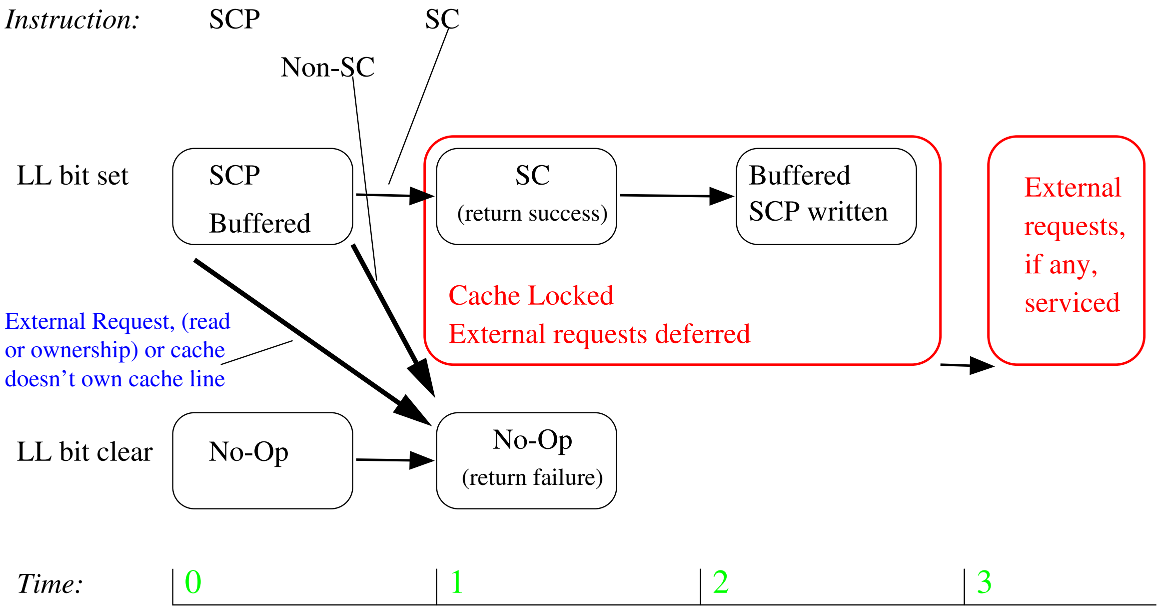

Figure 5.8: Behavior of Cache Controller during `SCP` and `SC`

1. The processor only performs the store specified by the `SCP` if the `LLBit` has not been reset. Once the processor determines that the `SCP` is executed successfully, it signals the cache that this is an `SCP` write that is to be deferred pending a successful `SC`.

   The cache then stores a copy of the entire cache line in `SCP Buffer`. (The cache must also maintain its current (pre-`SCP`) state and contents of the cache line in case the `SC` fails.) The cache-controller issues an INVALIDATE and begins to acquire ownership of the cache line. If there is a coherency conflict before the cache fully acquires ownership, the buffered "store" does not have to be nullified — clearing the `LLbit` is sufficient.

[4]For simplicity, I describe these steps as if the R4000 had an unpipelined datapath. In the few places where the extension to a pipelined datapath is not obvious, I comment briefly.

If a read or ownership request for either of the cache lines arrives now, it is serviced, and the LLbit is cleared.

2. If the next operation is not an SC the SCP has failed and the LLBit is cleared. (In practice, this occurs during the ID stage of the pipeline, so the LLBit will be cleared before the MEM stage of the SCP.)

   If the next operation is an SC and the LLbit is still set, the processor/cache-controller is committed to completing the SC/SCP. At the commit point the processor has ownership of the SCP line (else LLbit would have been cleared). The cache must no longer respond to any coherency requests, but must wait until it processes the deferred SCP buffer.

   The cache performs the store specified by the SC instruction. If the cache line is shared, then an invalidate (or an update, depending on the coherency protocol) must be sent[5].

   The processor is already committed to a successful SC. The success/failure notification mechanism of the SC instruction still behaves exactly as in the R4000 specification — it has not been changed. The processor begins loading of the processor's register with the result (success) of the SC. It begins concurrently with the write of the SC since the cache controller has already committed to performing both writes.

3. The cache line buffered in the SCP buffer is updated. An Invalidate was already sent for this line, else the SC could not complete. When this update is complete, the cache can now respond to any READ or ownership requests.

The cache controller does not respond to any read/ownership requests between the two invalidates, so it is impossible for any other processor or cache to see an inconsistent state. In other words, no processor sees any part of the transaction unless both SC and SCP succeed. (This also ensures that the reversed order of the writes does not violate total store order, since the reordering is undetectable.)

The actions above are intended to take place in the first level (primary) cache. Doing these actions in the secondary cache controller (for an off-chip secondary cache) seems too expensive — to do so requires the SCP cache line be flushed to the secondary cache on every SCP issue.

[5]This cache line invalidation is only an issue if the cache line is shared. If LL assumes exclusive ownership then this is not an issue. There are performance arguments for and against exclusive ownership; we simply follow the existing protocol on any given architecture.

If, however, there are compelling reasons, then `SCP/SC` atomicity can be moved off-chip to the secondary cache controller. In this case the processor must take additional actions. It must notify the secondary cache controller of the (successful) execution of `SCP` and `SC`. On the R4000, it can utilize the reserved bits of `SysCmd` to do this. Further, as noted above, it must flush and invalidate the primary cache line referenced by the `SCP` instruction (`LLAddr2`). This is critical, because the primary cache must not be left with an incorrect version of the `SCP` cache line if the `SC` fails. Flushing `LLAddr2` complicates handling both `LLAddr` and `LLAddr2` in the same cache line. This case must be specially handled by the processor and the primary cache controller, and not flushed out to the secondary cache.

5.3.2 Detecting Failure

Failure detection is provided by having `LL` and `LLP` each record, in an internal processor register, the cache line referred to by their respective arguments. If the processor notices a loss of ownership or an invalidation of these lines, it clears the `LLbit`. Given the cache coherence protocol, the processor chip is already informed of such an occurence because all cache coherency traffic involving these cache lines must be visible to the processor. For example, the processor must invalidate the primary cache if some other processor acquires ownership of the cache line.

If either cache line has been evicted from either cache due to collision, the architecture must ensure that the processor still sees all the coherency traffic related to that line. We discuss this issue next, (section 5.3.3), where we discuss changes to the *System Interface*.

5.3.3 Changes to the System Interface: Cache collisions

In the interval between the `LL/P` and the `SC/P` the processor is required to see all coherency traffic related to *both* cache lines referred to by the operands of `SCP` and `SC`. This normally happens automatically since the cache lines reside in the primary cache, and this coherency traffic is visible to the processor.

However, in a direct-mapped cache, one or both of the cache lines referred to by `SCP` and `SC` might be flushed from the cache due to collisions.

The R4000 must already deal with the situation when the `LL` or `SC` instruction maps to the same cache line as its operand. Even when there are separate instruction and data caches in the processor, a collision in the secondary cache will flush the entry from the primary cache because the R4000 caches preserve the (multilevel) *inclusion property* [37]. That is, the smaller faster caches must be strict subsets of the larger, slower caches.

If the cache line pointed to by LLAddr or LLAddr2 is evicted, it is a *system event* that is externally visible on the System Interface, and the *external agent* (which monitors external cache traffic on the R4000) is informed. A *link address retained* bit is set in SysCmd during the processor read, so that although the processor is replacing the flushed cache line, the processor still sees any coherence traffic that targets this cache line. If an external snoop or intervention request is directed at the cache line stored in the link address retained register, and the LLbit is *still* set, then the cache controller must return an indication that the cache line is present in the cache in the shared state, even though it is actually not.

Our design is identical to that defined in the R4000, except that two link address retained registers are required, two cache lines must be managed, and there are more circumstances where such collisions can occur. The processor must manage up to four cache lines (one for the SC, one for the SCP, and one for each of the two addresses). Any pair may now conflict.

It should be noted that deadlock does not arise because of collision, despite the fact that external cache requests are locked out during the two writes. The cache is not locked *until* the processor knows that both writes will succeed.

A different processor implementation (such as the R10000) that maintains separate primary instruction and data caches, and uses a 2-way set associative L2 cache, may eschew the use of link address retained registers. In such a processor it is still possible to eliminate pathological failure of the Double-Compare-and-Swap code in Figure 5.6 without requiring link address retained registers. Other changes are required, though. First, the processor must relax the inclusion property for the I-cache, and, second, the SC and SCP instructions must be in the same cache line[6]. Although the sequence may fail *once* due to a collision between the instruction and the data, the next time around we will succeed because the instruction is in the primary I-cache. The instruction sequence will *not* pathologically fail[7]. This approach is more feasible than requiring 3-way (or, more likely, 4-way) caches.

It is important to remember that our concern here is to eliminate failures caused by collisions between the operands themselves or between the instructions and one of the operands. Such a pathological failure will *never* succeed. We do not worry about collisions with other cache-lines causing the SC to fail — presumably the SC will succeed when it is retried. The specification of

[6] The assembler can automatically insert a NOP to ensure this alignment, if necessary.

[7] Writing arbitrary code using LL/SC is tricky — we recommend using only conventional code sequences. As such it seems reasonable to optimize the architecture for (a) particular code sequence(s).

SC and SCP are conservative, they are allowed to *spuriously fail* (For example, leaving open the possibility of an implementation that fails on *any* cache-miss between the LL and the SC).

5.4 Issues that affect the design

5.4.1 Cache-line issues and higher arity CASn

We exploit the well known[8] fact that transaction conflicts can be detected with almost no overhead on top of the coherency protocol. This is true regardless of the arity of the CASn primitive we wish to implement. However, this indirectly introduces other factors which determine the overhead of implementing a specific CASn, possibly introducing a steep cost for $n > 2$.

If we are to detect transaction conflicts through ownership conflicts, it is essential that *every* cache line we care about is currently in the cache, or at least participating in the coherency protocol. To implement CASn, for a given n, we need to monitor as many as n cache-lines[9]. Given the possibility of cache line conflicts, this either requires adding a special fully-associative "transactional cache" (equivalent to n link-address-retained registers), or else having at least an n-way cache.

For architectures, such as the R4000, which obey the inclusion property it is insufficient to only have an n-way cache on the processor. *All* the caches down to memory must be n-way, else a cache conflict in a secondary cache might evict a cache line from the primary processor cache. Our DCAS design just squeaks by – two-way caches are not uncommon. However, implementing CAS3 or higher is no longer a case of making only a small addition to the architecture. Requiring 4-way caches is a visible burden (in addition to the implementation cost of the associative cache itself, higher associativity may increase the cost of a cache hit, which may directly effect the CPU's clock cycle), and raises the bar for anyone wishing to argue in favor of hardware implementation of CASn, for $n > 2$.

Since two-way caches are relatively common, and DCAS is sufficient, it will be hard to justify implementing higher arity primitives.

[8] For example, Herlihy and Moss [43] remark that "any [cache] protocol that is capable of detecting ownership conflicts can also detect transaction conflict at no extra cost."

[9] Although we manage n lines, $n + 1$ lines have the potential of conflicting, since we must include at least one extra cache-line for the instructions.

5.4.2 Deadlock, livelock, and starvation

Given that our instruction set extension requires atomic access to two distinct cache lines, some form of deadlock avoidance is required. This must be implemented at the instruction level, as software approaches to deadlock avoidance cannot work if individual instructions deadlock. Fortunately, all that we require to allow software mechanisms to deal with deadlock is to make sure that instructions abort in potential deadlock situations, rather than hang.

It is easy to verify that our implementation never deadlocks. The only case of a processor withholding resources from another processor is during the atomic write phase following a successful `SC`. During that brief period it will not respond to external coherency requests. However, before it commits to issuing the writes it must guarantee that it already has all the resources it requires (ownership of both cache lines). Therefore it can proceed, will not wait for any other processor, and will respond to external requests within finite time.

There is, however, still a concern relating to livelock. A simple example arises if process P_1 uses `DCAS` on locations A1, A2, and process P_2 simultaneously uses `DCAS` on locations A2, A1 — both `SC`s may fail indefinitely as each successfuly `SCP` causes a cache-line invalidation in the other processor.

We restrict the semantics of `DCAS` to do only what is strictly necessary. Thus, instead of guaranteeing a successful write given a successful `SCP` (conditional on a successful `SC`, of course), we only guarantee a write following an `SC`. Until the processor reaches the `SC` instruction, the cost of acquiring exclusive ownership of the cache-line of the argument to `SCP` can be totally wasted.

The minimalist design of simply aborting at any potential deadlock sacrificed a guarantee of progress in order to achieve deadlock-freedom. Many systems guarantee that a processor will complete at least one write if it acquires exclusive access to a cache line. Such guarantees help avoid livelock and starvation by ensuring that some progress is made by the system even during high contention. However, such guarantees are neither needed nor completely effective for instructions such as `SC` or `SCP`.

First, the semantics of `SC` allow for spurious failure, so the meaning of the "guarantee" is slightly weaker than for normal writes. In the case of `Store`, the system *must* eventually complete the write in order to make progress. `SC` may fail and yet the system is still progressing (the instruction can return).

Second, in the case of `SC`, livelock and starvation-freedom are also problems at the software level — trying to implement instruction-level solutions is misguided. Even if we hypothesized a livelock-free implementation[10] of SCP/SC at the instruction level this does not prevent starvation at the system level. Livelock *only* occurs if the software loops in both processes in a roughly synchronized manner. (As noted, there is no concept of livelock at the instruction level because the processor returns failure and goes on). If process P_1 and P_2 both loop, inserting and deleting elements from a single shared list, then nothing prevents process P_1 from succeeding each time through the loop causing process P_2 to fail each time. If contention that can cause starvation *is* a possibility, then the software algorithm must address it (say, by using contention-reduction techniques such as exponential backoff or advisory locking) in any case.

Third, any solution that prevents starvation also, by definition, avoids livelock. Given that we must assume looping at the software level for a problem to exist, and given that once we assume software looping the hardware cannot prevent starvation, and given that any (software) solution that prevents starvation also prevents livelock, implementing livelock or starvation avoidance in hardware is unnecessary.

In summary, the best guarantee about progress one can make for SC is weaker than for Store. Avoiding starvation and livelock at the instruction level is *not* sufficient to avoid starvation at a higher level. Given that this *must* be addressed in software whenever it is an issue, an expensive hardware solution is unnecessary, provides little or no benefit and need not be implemented.

A guarantee of progress in the case of single SC is similarly unjustified. The difference there, however, is that the cost of providing a "guarantee" of progress is lower for CAS than for DCAS. DCAS, or SCP/SC, both issue *two* writes, while CAS or a single SC only issues a single write. Thus, for CAS, the cost of guaranteeing progress is cheaper, and does not make deadlock avoidance any harder.

5.4.3 Performance concerns

One final issue is performance. As long as the cost of DCAS is not more than twice the cost of CAS, DCAS clearly provides performance advantages. In fact, for many algorithms (a linked list, for example), there are overall performance benefits of DCAS over CAS even if DCAS costs many times CAS. The benefits are realized at a higher level, in reduced complexity of algorithms. The hardware implementation uses information from the cache controller to stop relevant memory traffic, removing the complexity of preventing other processors from seeing intermediate steps. The

[10] A simple implementation would delay acquiring the cache lines for both SCP and SC until the SC was successful, and then trying to upgrade both cache-lines to exclusive ownership. If the cache-lines are upgraded in some fixed processor-independent order (e.g. lowest address first), then the implementation is livelock free.

hardware implementation also eliminates the (smaller) cost of undoing intermediate writes in the case of preemption between the two CAS's. Finally, a hardware implementation piggy-backed on the coherency protocol pushes the synchronization down to per-cache-line ownership. This eliminates the cost of locking (other than what is already needed within the coherency protocol to ensure correct cache line ownership) and avoids the need for coarser granularity locks that are potential hot-spots in the system.

Although we have no measurements of a real hardware implementation (and no proof-of-concept hardware implementation), it is unlikely that SCP will cost more than SC. More broadly, it is unlikely that Double-Compare-and-Swap functionality will cost twice as much as Compare-and-Swap functionality. DCAS requires the same number of memory operations as two sequential CAS's, and only slightly more logic. On the other hand, some mechanism can be shared. For example, if a SYNC[11] or a pipeline flush is required, only one would be required for the SC/SCP pair. In the worst case, where an implementation can share no mechanism, the aggregate cost of DCAS should be, at worst, twice the cost of CAS. All of our simulations conservatively assume that a DCAS costs roughly twice a CAS.

An interesting question related to performance is whether LL should only acquire ownership for read, or whether it should acquire exclusive ownership immediately. The argument in favor of immediate exclusive ownership is that it overlaps the latency of acquiring exclusive ownership with the instructions that execute between the LL and SC. It also requires one fewer bus transaction than first acquiring the cache-line in shared mode and then upgrading to exclusive. The argument against exclusive ownership is that in the case that the code branches and does not perform the SC (or in cases where the SC would fail) deferring exclusive ownership saves an expensive memory operation and does not impact other processes that have a copy of this cache-line.

This is equivalent to asking whether Compare-and-Compare-And-Swap is a useful optimization over Compare-and-Swap. Compare-and-Compare-And-Swap, or Compare-and-Double-Compare-And-Swap, does a non-atomic test of each location for equality with old1 and old2 respectively, in order to avoid the expensive CAS or DCAS. This is analogous to test-and-test-and-set, commonly used to implement spin-locks.

Note, that given a hardware LL that acquires exclusive ownership, one can simulate the weaker LL by doing a normal Load instead, followed by the instructions that are to be protected by LL and SC. If the SC will be avoided, then the code avoids the LL too, thus providing the same behavior as

[11] An R4000 SYNC forces all prior reads and writes to complete before any subsequent ones start. LL and SC each implicitly perform a SYNC. The R10000, however, decouples SYNC from LL and SC.

an LL which only acquires cache lines in shared mode.

Exclusive LL can, if needed, be approximated in a system whose hardware only supports non-exclusive LL by a simple trick. The code must reserve a dummy location in the same cache line as the operand of LL. Immediately prior to issuing the LL, the code performs a write to the dummy location, acquiring exclusive ownership of the cache-line. Unfortunately, this only *approximates* exclusive LL (since exclusive ownership may be lost before the LL) and is more expensive (extra memory operations and bus transactions).

From the hardware point of view, then, the real question is whether there are compelling reasons to implement LL with exclusive ownership at all. From the software point of view, the question is whether to code and use a compare-and-compare-and-swap optimization given a hardware CAS or exclusive LL.

There are several reasons one might suspect that choosing non-exclusive LL or using Compare-and-Compare-And-Swap could significantly improve performance over exclusive LL or bare CAS.

First, it is well known that test-and-test-and-set, rather than simple test-and-set, improves the performance of spin-locks under contention, with little penalty under low contention. Second, when CAS is implemented by the OS in software protected by a spin-lock, Bershad [17] advocated the use of compare*-and-compare-and-swap. This operation differs from compare-and-compare-and-swap in that the former combines the implementation of CAS and its spin-lock. It tests whether *both* the lock is free and the data is unchanged, before acquiring the lock. This approach avoids unnecessary synchronization operations and reduces bus contention, and is known to be a significant optimization.

Our measurements show that exclusive LL or bare hardware CAS is slightly better than non-exclusive LL or compare-and-compare-and-swap. The advantage is too slight, however, to provide a compelling argument for exclusive LL in hardware, all other things being equal. (LL is typically implemented acquiring ownership only for read, since it is an extension of a normal Load.) The measurements, do, however, argue against the software optimization of compare-and-compare-and-swap, in most cases.

It is worth understanding the difference between my results and the two cases mentioned above. First, the result is mildly surprising. Second, sometimes compare-and-compare-and-swap *is* occasionally a reasonable optimization — it is important to understand *when*.

Bershad's compare*-and-compare-and-swap is needed, when CAS is implemented in software by the OS, for two reasons. First, it is precisely analogous to `test-and-test--and-set` vs. `test-and-set` to implement spin-locks. It is better to spin on a cheap local test and only attempt the `test-and-set` (accessing the non-cached copy and causing bus traffic) if the local test succeeds. The second reason is to avoid expensive lock acquisitions if they will be wasted. If the `CAS` will fail, it is harmful to acquire the lock — it is expensive for the transaction issuing the `CAS`, and also slows down other competing transactions.

If `DCAS` or `CAS` are implemented without a lock (e.g. in hardware), then doing a simple (non-exclusive) read first does not have the same benefits. Non-blocking algorithms do not typically "spin" in the same sense as spin-locks. Consider the case where the value has not changed, and the cached copy (and `oldval`) are up-to-date. A cheap local read from the cache tells the process that a `CAS` is likely to succeed — but this means that our reward is to immediately execute the `CAS` or `DCAS`. The latter are expensive memory operations. If the value has not changed, but the cache-line is *not* up to date, then the read simply costs an extra memory operation.

Now consider the case where the value *has* changed and is detected by the preliminary read. Our simple read will probably not hit in the cache, and will therefore be relatively expensive. In this case we can avoid executing the `DCAS`, but we've already paid for an expensive read. That isn't to say that it had *no* value: after all, the read is cheaper than a `DCAS` (and it does not gain exclusive ownership thus invalidating entries in other caches). This last benefit, however, is of questionable value. In practice, the odds of `DCAS` failing are highest when many processors are competing for the same data-structure. The preliminary read avoids invalidating other caches, true, but one must remember that most of these other caches had already been invalidated by the transaction which just recently completed successfully.

So, unlike the case of spin-locks, there is no clear-cut gain here by doing reads to avoid an expensive failing `CAS` or `DCAS`. For purposes of comparison, I ran my tests using both implementations of `DCAS`. The results in Figure 5.9 are for a simulated hardware implementation *without* this "optimization", i.e. `LL` gets exclusive ownership of its operand's cache line. The results in Figure 5.10 are for a hardware implementation that *includes* the "optimization" of allowing `LL` to perform shared reads. Figures 5.9 through Figure 5.20 all include measurements with, and without, `LL` acquiring exclusive ownership. The measurements support the preceding description, and shows that the difference between exclusive `LL` and shared `LL` is insignificant, and that `compare-and--compare-and-swap` is not a useful optimization. Under low contention exclusive ownership *slightly* out-performs ownership for read. Under high contention the degradation is slightly less severe when `LL` does not acquire exclusive access, however, the point at which degradation starts being noticeable occurs at a smaller number of processors.

In summary, `compare-and-(double-)compare-and-swap` is valuable only when two conditions are met. First, a cache hit on the non-exclusive early read must avoid the expensive `CAS` (or `DCAS`) operation. Second, when the code avoids the expensive atomic operation, it must spin again, and retry.

These two conditions are met when we implement spin-locks, and an equivalent case even occurs once in the universal transformation I presented in Chapter 3. There it arises in the second use of `DCAS` in Figure 3.7 in the routine `trans_conclude`. There, for algorithmic reasons, we want the current owner of the log to have absolute priority over all other transactions, and not allow any other transaction to cause the owner's `DCAS` to fail. When `trans_conclude` loops, it sets `id` to `NULL` — expecting its own `DCAS` to fail until the current transaction relinquishes ownership of this log (which sets `log->id` to `NULL`). When `log->id` is not `NULL`, then `trans_conclude` is able to skip the `DCAS`, abort early, and loop again.

5.4.4 Speculation and `LLP/SCP`

Some modern processors use *speculation*. The CPU executes instructions *before* it knows whether the instruction should execute. For example, the CPU may fetch and execute instructions beyond a branch before determining whether the branch is taken or not. The effects of the instruction are either withheld until the instruction commits, or else are undone if the instruction should not have been executed.

Speculation raises two issues with respect to `LLP/SCP`.

The first issue is whether the `SC` should succeed or not. Consider a speculatively executed instruction that follows an `SC`. It may flush a cache line used by `SC` or `SCP` *before* the `LLbit` is tested by `SC`. This can happen either due to dynamic scheduling or by a memory operation done in an earlier stage in the pipeline. Should this clear the `LLbit` and cause `SC` to fail?

The second issue is how to couple the two writes (from `SCP` and `SC`) if `SC` were to be speculatively executed. The simple scheme outlined in this chapter is not feasible in the face of dynamically scheduled speculative instructions.

The simplest solution to both of these problems is to prevent speculative execution after an `SC`. In fact, this is the solution already effectively taken by the R10000 (the issue of speculative execution causing `SC` to fail is equally applicable to single `LL/SC`). MIPS recommends that 32 `NOP`s be inserted after each `SC` (the R10000 assembler inserts these automatically), precisely to avoid this sort of problem.

5.4.5 Hardware Contention Control

As a further extension, a processor can provide a conditional load instruction or `Cload`. The `Cload` instruction is a load instruction that succeeds only if the location being loaded does not have an *advisory lock* set on it, setting the advisory lock when it does succeed.

With `Cload` available, the version number is loaded initially using `Cload` rather than a normal load. If the `Cload` operation fails, the thread waits and retries, up to some maximum, and then uses the normal load instruction and proceeds. This waiting avoids performing the update concurrently with another process updating the same data structure. It also prevents potential starvation when one operation takes significantly longer than other operations, causing these other frequently occurring operations to perpetually abort the former. It appears particularly beneficial in large-scale shared memory systems where the time to complete a `DCAS`-governed operation can be significantly extended by wait times on memory because of contention, increasing the exposure time for another process to perform an interfering operation. Memory references that miss can take 100 times as long, or more, because of contention misses. Without `Cload`, a process can significantly delay the execution of another process by faulting in the data being used by the other process and possibly causing its `DCAS` to fail as well.

The cost of using `Cload` in the common case is simply testing whether the `Cload` succeeded, given that a load of the version number is required in any case.

`Cload` can be implemented using the cache-based advisory locking mechanism implemented in ParaDiGM [26]. Briefly, the processor advises the cache controller that a particular cache line is "locked". Normal `loads` and `stores` ignore the lock bit, but the `Cload` instruction tests and sets the cache-level lock for a given cache line or else fails if it is already set. A store operation clears the bit. This implementation costs an extra 3 bits of cache tags per cache line plus some logic in the cache controller. Judging by our experience with ParaDiGM, `Cload` is quite feasible to implement. However, the software-only contention reduction protocol described in Chapter 3 is comparably effective at reducing contention under reasonably low contention, and has only slightly more overhead in the case of no contention. Under high contention, `Cload`, by itself, helps, but requires additional software mechanisms to be competitive with the software-based algorithms described earlier in Chapters 3 and 4. Further, under very high load the contention reduction achieved by `Cload` appears brittle. Further experiments are needed to understand how to control `Cload`'s effectiveness in such situations. Given the slight performance benefit, contention reduction alone is not sufficient to justify implementing a `Cload` instruction. If one exists for other reasons, of course, it may be exploited for contention reduction.

5.5 DCAS: Hardware vs. OS implementation

DCAS functionality (as well as CASn (n-location CAS)) *can* be implemented in software with OS support, using a technique introduced by Bershad [17]. DCAS is implemented by a few instructions, using a lock known to the operating system. In the uncommon event that a process holding this lock is *delayed* (e.g. by a context switch, cache miss, or page fault) the operating system rolls back the process out of the DCAS procedure and releases the lock. The rollback procedure is simple because the DCAS implementation is simple and known to the operating system. Moreover, the probability of a context switch in the middle of the DCAS procedure is low because it is so short, typically a few instructions. Thus, the rollback cost is incurred infrequently.

This approach has the key advantage of not requiring hardware extensions over the facilities in existing systems.

However, the software approach has only been tested for single CAS and under situations where all processes were applying CAS to a single data-structure [17]. The measurements presented there are not directly relevant to an OS/software implementation of DCAS. The case of a single write possesses certain properties that make it more amenable to a software implementation than the multiple write case: (a) the write is atomic, so no backout or roll-forward of actual data modifications is required. The lock must simply be cleared and the PC be reset. (b) Since there's only one write, there's no issue of other processes seeing "intermediate states".

There are also a few other concerns with the software approach for primitives such as DCAS.

First, there is the cost of locking. The straightforward implementation requires the DCAS procedure to access a *single* common global lock from all processes. In a multi-level memory with locks in memory, the memory contention between processors for this lock can be significant.

It is difficult to reduce contention on the Bershad lock when a single lock couples *every* data structure. A more sophisticated implementation will try to associate different locks with each DCAS instance, or reduce contention by some other means of using multiple locks. Then there is more cost and complexity to designate the locks and critical section to the operating system and to implement the rollback. The locking and unlocking also modifies the cache line containing the lock, further increasing the cost of this operation because writeback is required. Finally, using multiple locks re-introduces the possibility of deadlock, and code dealing with deadlock introduces another layer of complexity.

However, assume all these obstacles can be overcome, and assume that the cost of determining *which* lock(s) to acquire is zero. The result of using multiple locks is to have each single lock protect a smaller number of locations or data structures. I parameterized all my tests by S, the number of independent locations protected by each lock. Increasing the number of locks simply shifts the performance curves to smaller values of S per lock. Thus, the performance results in Figures 5.9 through 5.14 can be used to provide an upper bound on the performance improvements possible by the more sophisticated approach. At best, multiple locks reduce the ratio of data structures to locks, but does not eliminate the lock as bottleneck.

At first glance, an attractive approach seems to be allowing the software to specify the lock meant to control access to the memory locations, and have the lock passed as an argument to the routine implementing `DCAS`. If such a scheme could work, then contention on the lock should be no worse than contention on locks in a blocking implementation — which we know to be low. The problem with such an approach becomes apparent when we consider the linked list example from Chapter 4. Descriptors can move from one list to another. If a processor descriptor moves from the ready queue to the free list, and two processes simultaneously try to delete that descriptor from both lists then each process may call `DCAS` under the protection of a different lock. This will allow each process to see intermediate state introduced by the other process — in short `DCAS` will not be atomic. Either the free list and the ready queues must share a single `DCAS` lock, or finding the appropriate lock must be a more complicated and expensive approach based on individual descriptors.

A more complicated implementation of compare*-and-compare-and-swap using queuelocks [9, 66] instead of spinlocks might help here, but the complexity of such a scheme is high. I have not investigated this approach.

Using the Proteus parallel architecture simulator [21] I compared the OS approach to a hardware implementation to determine the impact of a single global lock. I simulated a 64 processor multiprocessor with a shared bus using the Goodman cache-coherence protocol and a cache with 2 lines per set. Memory latency is modeled at 10 times the cost of a cache reference. The cost of a `DCAS` is modeled at 17 extra cycles above the costs of the necessary memory references. The additional cost of a `CAS` over an unsynchronized instruction referencing shared memory is 9 cycles[12].

We varied the amount of work (i.e. computation other than synchronization in each transaction), and the number of active processors. The measurements reported in [17] were for systems in which there was a single data structure as well as a single global lock. Realistically, the number of data structures would usually be greater than one. We wanted to measure the impact of a single global lock protecting `DCAS` on *multiple* data structures. Our tests therefore varied the number of data

[12] As noted in Section 5.4.3, this penalty for `DCAS` over `CAS` is conservative. It is unlikely that `DCAS` would actually cost twice as much as `CAS`.

structures being accessed simultaneously on the system.

We tried to use roughly the same test as Bershad did in [17]. Our test consisted of each active processor continually performing transactions. The aggregate throughput results are in units of successful transactions per 100,000 bus cycles. Each transaction consisted of a processor randomly[13] choosing one of the S data structures, reading its state, performing W cycles of "work", and writing a new state (if the structure had not been modified in the interim). Note that the W cycles of "work" are performed after reading the initial state and before updating with the new state. Thus it was part of the transaction. Each structure's state was recorded in 2 words so the update could be done in a single `DCAS`. The structures were each on their own page of virtual memory. Unlike the algorithms in the previous chapters, we do *not* employ any contention reduction techniques here, since we want to understand how `DCAS` behaves under contention.

I limited the simulation reported here to one process on each processor and did not take any page faults. Thus there were no preemptions, no long delays and we never had to roll back. This is, in many ways, a best-case scenario for the OS approach — we never incurred any OS overhead. Any change that introduced rollback would have made the OS approach look worse compared to the hardware implementation.

In spite of stacking the odds in favor of the OS approach, the impact of the global lock is still significant. I display representative results in Figures 5.9 through 5.11. They report throughput as a function of the number of active processors when work = 100 cycles (so the base synchronization cost is about 10% of the total cost of a transaction), and as we vary the number, S, of data structures being accessed. For OS assisted `DCAS` throughput drops noticably as the queue for the global lock and the cost of transferring the lock's cache line becomes comparable to the total cost of a single transaction. The throughput of the hardware `DCAS` implementation, however, scales up roughly as the number of processors approaches the number of data structures. Contention remains low for individual data structures until the number of processes is several times the number of data-structures.

The performance difference is still noticeable on systems with lower access rates. Tests with much larger values of *work* (e.g. lower access rate) show the same *shape* curves, but the peak is shifted to a larger number of processors. Figures 5.12, 5.13 and 5.14 show the data for work = 1,000 cycles, so the synchronization cost is only about 1% of the total cost of a transaction. Figures 5.15, 5.16 and 5.17 show the results measured while holding the number of structures fixed at

[13] The overhead of `random` and setting up each transaction amounts to over 120 cycles. The cost of synchronization, not including spinning on the lock, averaged a little under 30 cycles.

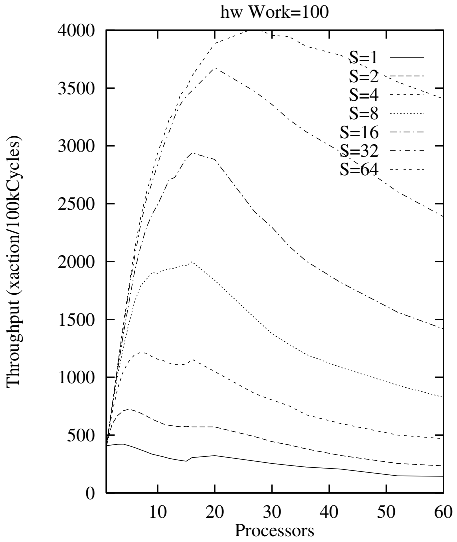

Figure 5.9: Throughput of system with work=100 using hardware `DCAS`. Throughput increases as the number of structures increase. When the number of processors increases to a point comparable to the number of structures, memory contention begins to degrade throughput.

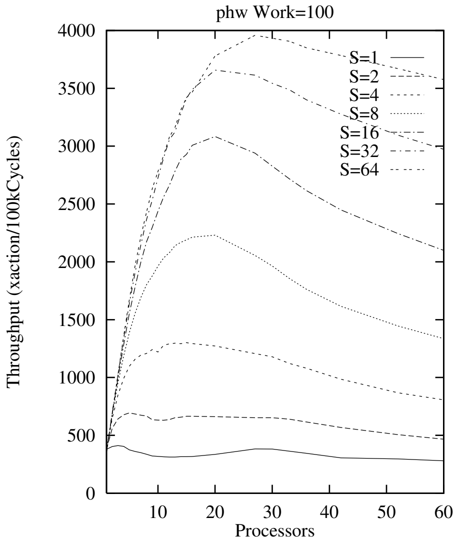

Figure 5.10: Throughput of system with work=100 using hardware `DCAS` without acquiring exclusive access on an `LL`. Degradation due to contention is slightly less severe than with exclusive ownership in Figure 5.9. For low contention, throughput is slightly worse.

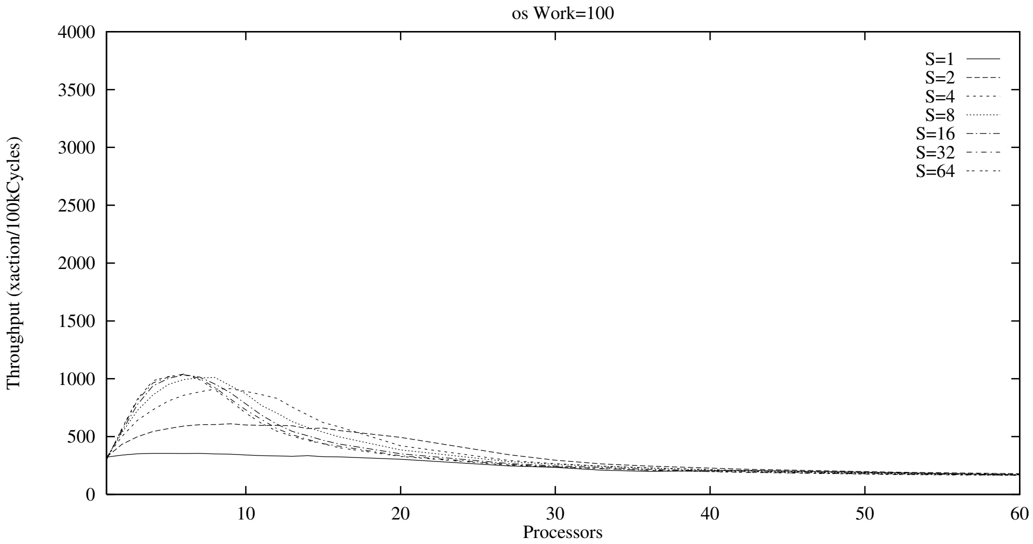

Figure 5.11: Throughput of system with work=100 using OS assisted `DCAS`. Negligible gain in throughput as the number of active structures increases because of contention on the global locks.

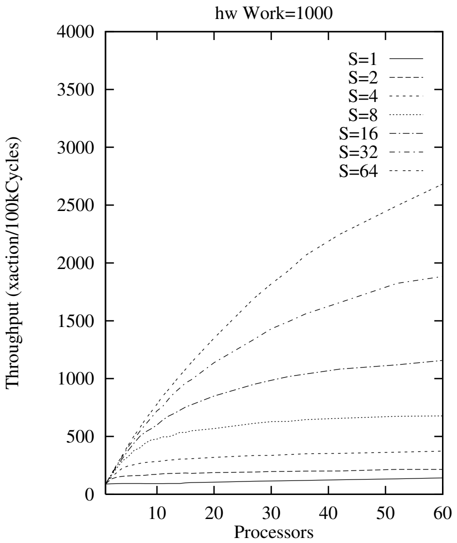

Figure 5.12: Throughput of system with work=1000 using hardware DCAS, where LL(P) acquires exclusive access of the cache line. Throughput still increases as the number of structures increase.

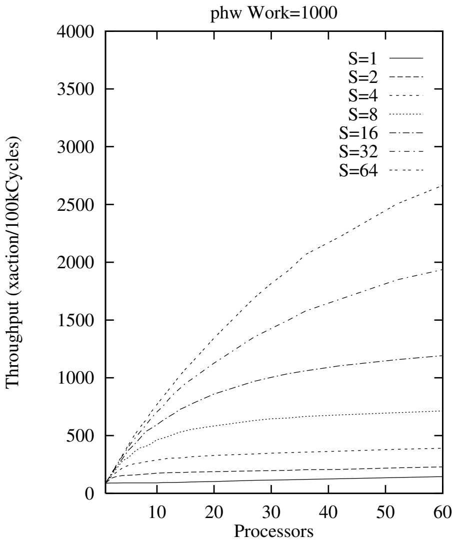

Figure 5.13: Throughput of system with work=1000 using hardware DCAS without acquiring exclusive access on an LL. Throughput is comparable to Fig. 5.12.

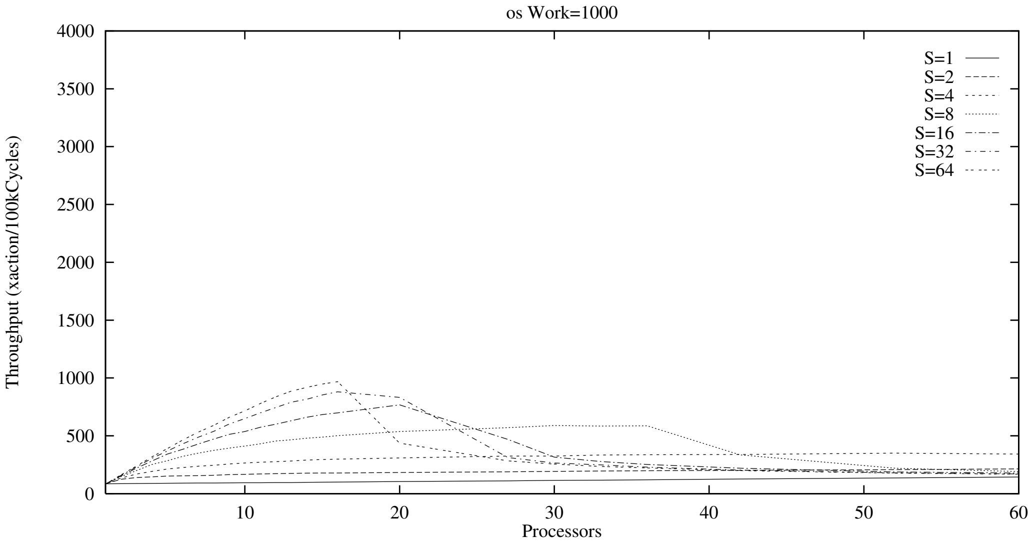

Figure 5.14: Throughput of system with work=1000 using OS assisted DCAS. Small gain in throughput as the number of active structures increases. The global locks are still hot-spots, although synchronization overhead is only around 1% of the time for an individual transaction.

$S = 64$ and varying work, W, from 1 to 10,000 cycles. As a point of reference, [59] measured synchronization cost for fine grained parallel simulation of multiprocessors at 70 to 90% of run-time. This corresponds most closely to the curves with work = 10 cycles. So, at least one real system would suffer an even worse penalty due to OS rather than hardware DCAS than Figures 5.9 through 5.14 would suggest. For comparison, according to [102], 3 of the programs in the SPLASH-2 benchmark suite spend upwards of 30% of their runtime spent in synchronization, another 3 exceed 10%, and the remaining 6 spent between 2 and 10% of their run-time performing synchronization. Thus all would be comparable to our experiments with work between 10 and 100 cycles.

It is instructive to note that the lower the contention on individual data structures the more significant the advantage of hardware DCAS over OS support. Real systems may show even more benefits than demonstrated in Figures 5.9 through 5.11 on page 140, since most are designed to exploit locality and minimize contention. I modified the simulation to improve the locality of the data structures. In the modified experiment, each processor accessed a single structure 95% of the time (the structure numbered PID mod nstructs). The other 5% of the time it performed an operation on a structure chosen at random, uniformly, from all the structures. Figures 5.18 through 5.20 show representative results for the modified experiment which exhibits more locality.

For hardware DCAS it is possible to restructure an application to have fine-grained parallelism, and a high access rate, as long as the structure of the application preserves temporal and spatial locality. For OS-supported DCAS, the global lock causes hot-spots that are difficult to eliminate through restructuring.

Beyond the issue of contention on the global lock, other issues are relevant. The second issue, on multiprocessors, is that care must be used by readers of shared data structures if they want to support unsynchronized reads. Without depending on the lock, readers can see intermediate states of the DCAS, and read tentative values that are part of a DCAS that fails. In addition to the higher cost of executing more instructions, requiring synchronization for reads significantly increases contention on the global lock[14]. In contrast, systems that provide hardware DCAS require no additional read synchronization — the DCAS itself guarantees that no intermediate results will be visible. Further experience and measurements are required to determine if this is a significant issue on real systems.

Third, the OS approach must treat asynchronous handlers (interrupts or signals) specially. Either interrupts or signals are disabled during the body of the DCAS, or else we must treat each interrupt

[14] In the Cache Kernel, the use of *Type Stable Memory-Management* (as described in Section 4.2) reduces the danger of unsynchronized reads because the reads cannot cause type errors. Writes are protected by the global lock, and the final DCAS will detect that the unsynchronized reads were suspect, and fail. This does not help totally read-only operations, though.

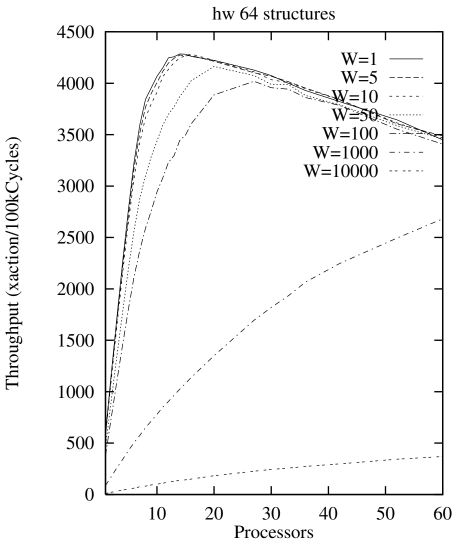

Figure 5.15: Throughput of system with 64 structures using hardware DCAS. LL acquires exclusive access to the cache line. With 64 structures, there's a reasonable amount of concurrency. But because there's no locality, throughput tops out after a handful of processors. For large number of processors, throughput is dominated by contention costs.

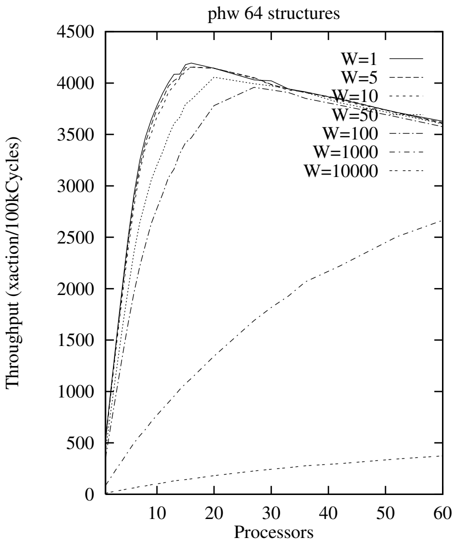

Figure 5.16: Throughput of system with 64 structures using hardware DCAS without acquiring exclusive access on an LL. This optimization has very little impact over exclusive access as explained in the text.

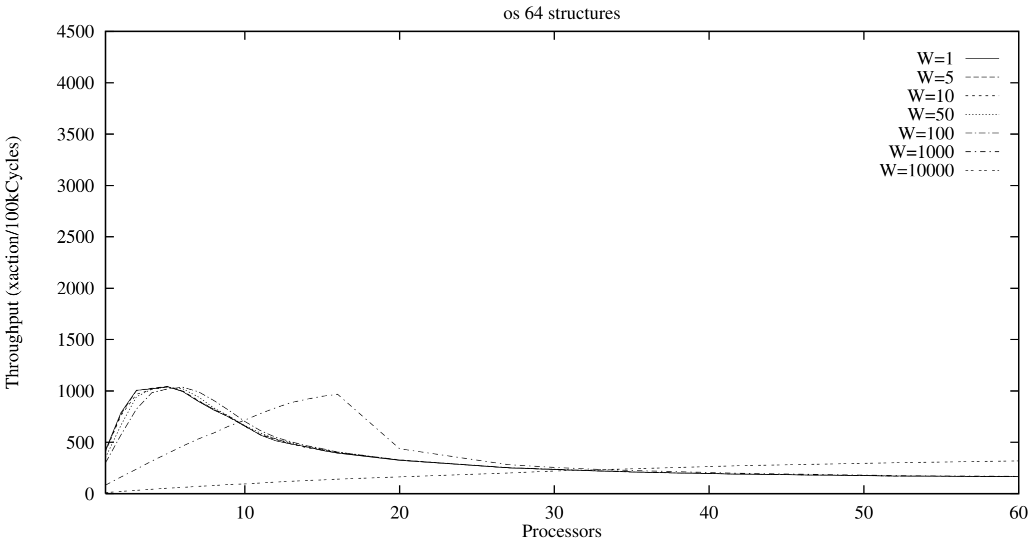

Figure 5.17: Throughput of system with 64 structures using OS assisted DCAS. No gain in throughput as the number of active structures increases since all processes convoy on the global lock.

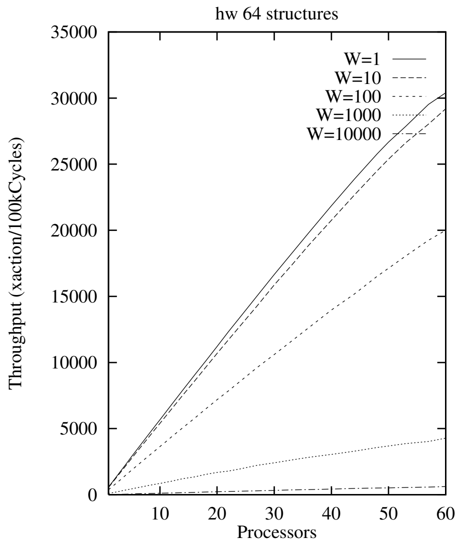

Figure 5.18: Hardware DCAS, LL exclusive access, mostly local structures.

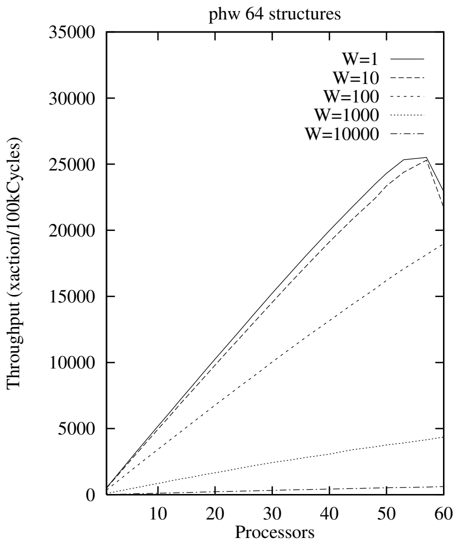

Figure 5.19: Hardware DCAS, LL no exclusive access, mostly local structures.

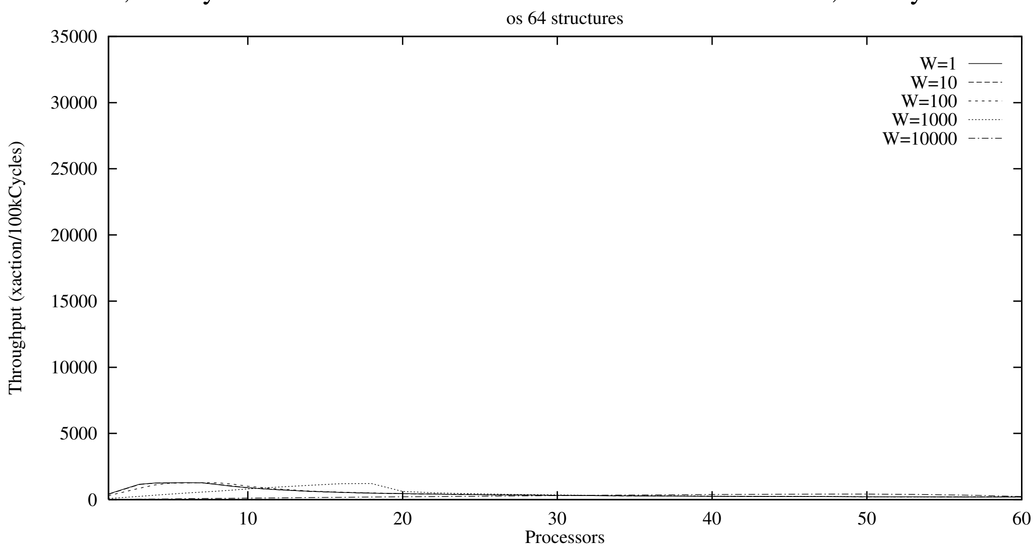

Figure 5.20: OS assisted DCAS, mostly local structures.

as a context switch and check to see if we need to roll back. An additional (but minor) consequence of this is a loss of portability for code that might interlock with asynchronous handlers, The cost of disabling asynchronous signals is often high enough (especially on processors that require an off-chip controller to disable interrupts) so that such code would be avoided in library routines used by functions that did not interact with asynchronous handlers.

Finally, the Bershad mechanism seems harder to test under all conditions. For instance, it is possible that one of the write operations that the rollback needs to undo is to an area of memory that has been paged out or that one of the addresses may fault. The system also needs to ensure that a thread is rolled back out of any `DCAS` critical section if it is terminated. We believe our hardware implementation is simpler to verify and operates naturally on top of the virtual memory management of the system and on top of directly accessible physical memory at the lowest level of the system software.

5.6 Related Work

Most processors provide at most single `Compare-and-Swap` (`CAS`) functionality to support non-blocking synchronization. A few processors such as the Motorola 68040 provide a multi-word atomic instruction but that functionality is rare and is not present in any RISC processor to our knowledge. The RISC-like extension that I propose in Section 5.2 suggests that it is feasible to support `DCAS` functionality in modern processors without significant changes.

Other proposals for adding hardware support for non-blocking synchronization either err on the side of too much or too little. The maximalist proposals are too complex or expensive, so the odds of implementation are low. The minimalist proposals do not provide enough functionality to be useful in a wide enough range of cases.

Transactional Memory [43] proposes hardware support for multiple-address atomic memory operations. It is more general than `DCAS` but comes at a correspondingly higher cost. The proposed hardware implementation requires six new instructions, a second set of caches in the processor, twice the storage for cache lines actively involved in a transaction, and a more complicated "commit" protocol. Transactional Memory avoids issues of cache collisions by (a) maintaining a completely separate "transactional cache" and (b) mandating that the transactional cache be fully associative. Both these features significantly complicate the processor/cache controller implementation. Further, a fully associative primary cache would either require a longer clock cycle, slowing down the entire machine, or would involve changes to the data-path to deal with two different hit times in the 3 memory stages of the R4000 pipeline. LLP/SCP appears to be a more practical solution because DCAS functionality is sufficient and significantly simpler to implement.

Oklahoma Update [93] provides an alternate implementation of multiple-address atomic memory operations. Rather than duplicating entire cache lines involved in transactions (as Transactional Memory does), Oklahoma Update requires only a reservation register per word used in their version of Load Linked. This register contains flags plus two words . This contrasts with our implementation which requires a "link address retained" register per word and a single cache-line buffer for the delayed SCP. Our design can also work with a word register instead of an entire cache line to buffer the SCP. However, this approach adds complexity to the chip's logic, slows down the SC and increases the time the cache is locked so the savings are questionable. The Oklahoma Update attempts to implement some features in hardware (e.g. exponential backoff or sorting addresses to avoid deadlock) which are better done in software, and which needlessly increase the complexity and size of the chip. Measurements have shown that, for both blocking [9] and non-blocking [39, 42] algorithms, exponential backoff needs tuning to perform well. It therefore seems especially questionable to implement this in hardware. Also, buffering of certain requests that come in during the "pre-commit" phase can cause two processors with non-interfering reservation sets to delay each other, increasing contention. This arises from the policy of making the commit phase blocking. Consider processors P_1, P_2 and P_3. P_1 accesses cache lines Y,Z, P_2 X,Y, and P_3 W,X (addressed in ascending alphabetical order). P_1 and P_3 should not interact. However, if P_1 holds Y and Z and P_2 holds X, then when P_2 asks P_1 for Y, P_2 stalls, and buffers P_3's request for X. Thus, P_1 delays P_3. Longer chains can be constructed. This is not as bad as the shared global lock required for the OS/software implementation, but does raise some questions about scalability. Further measurements would be required to determine whether this is significant in real systems. The answer depends upon the degree of interaction between atomic updates and the number of processors in typical systems. Finally, the Oklahoma Update does not guarantee consistency unless the programmer exercises care in performing at least one write among any set of reads shared among multiple transactions.

Both Transactional Memory and Oklahoma Update permit contingent writes to occur in the middle of a transaction. This adds complexity to the implementation since processors must keep the old values around in order to undo any modifications if the transaction is aborted. Further, the processor must allow programs to read the updated value at later stages of the transaction. In contrast, our proposed DCAS implementation requires both writes to occur consecutively (in the SCP and SC). We know at the start whether the writes will succeed or not, and the writes occur atomically, so only the `SCP` value needs to be buffered. No processor can ever legally read the buffered `SCP` value, so no changes need to be made to the normal read data path.

These proposals also differ in their approach to deadlock avoidance. Transactional Memory does not relinquish ownership to requestors during a transaction. A failed attempt to acquire ownership aborts a transaction, avoiding deadlock. An attempt to acquire ownership from a completed or failed transaction succeeds. The Oklahoma Update relinquishes ownership to the requestor until the *precommit phase*, at which point it defers all new external requests (requiring an extra buffer) until after the commit. Deadlock is avoided by *ordering requests*, but as noted above chains of non-interacting processes can delay each other. Our implementation relinquishes ownership and aborts, until the commit point. We commit only if we own both cache lines, and then defer requests during the two actual writes by `SC` and `SCP`. Deadlock is avoided by aborting the owner upon loss of ownership. We are more likely to abort a running transaction before the commit point than either Transactional Memory or Oklahoma Update. However, the addition of other forms of contention reduction, required in any event, reduces the likelihood of an ownership request arriving before the `DCAS`. This is a good balance — deadlock is avoided simply and cheaply, but the odds of ownership loss are low.

These different designs arise because of different assumptions regarding the number of memory locations that should be atomically updatable at one time. The Transactional Memory paper conjectures between 10 and 100 and Oklahoma Update places the knee at 3 or 4. If implementation were free, then more locations are better and more powerful. However, our implementation at 2 (`DCAS`) is by far the simplest extension to existing processor designs. A key contribution of this thesis is evidence that indicates that `DCAS` is sufficient for practical performance, making the extra hardware complexity of the other schemes unnecessary. In Chapter 3 we demonstrate that while `DCAS` provides significant performance over `CAS`, `CAS3` and higher arity primitives only offer incremental improvements over `DCAS`.

James [48] also aims for a minimalist proposal, arguing for even fewer locations than `DCAS`. He proposes a hardware implementation of `Conditional Lock` (`C-Lock`). `C-Lock` atomically stores `new` in `*addr2` if-and-only-if `addr1` contains `old`. Since there is only one write, issues of starvation do not arise, and the hardware implementation is simplified.

Unfortunately, `C-Lock` cannot efficiently synchronize some common data structures implemented in Appendix D. It was designed to use `addr1` as an advisory lock over a complex data structure. By using an advisory lock, `Conditional Lock` can be used to implement transactional memory in software. Careful examination of the algorithms I present in Chapter 3 shows that replacing DCAS with C−Lock increases even the complexity of the worst-case run-time of software transactional memory. Although this is not a proof that C−Lock can never achieve comparable results, I note that the cost is offset by the fact that DCAS can solve the *Leader Election problem* or *Maximal Independence Set* in $O(1)$, while C−Lock cannot. The former problem shows up (although in hidden form) in several places in the universal transformation algorithm. Thus DCAS is still needed.

Further, the argument for C−Lock over DCAS rests not on the sufficiency of the primitive, but on the simpler implementation of C−Lock. However, C−Lock is only significantly simpler than DCAS if we try to ensure forward progress, which we defer to software. Given this, we find that C−Lock must still manage two cache lines, must still ensure that the first was not modified during the write, and must not allow anyone to gain exclusive ownership of the read-only cache-line during the write. The only savings is that C−Lock has only one atomic write, while DCAS has two. The savings of a small amount of logic and a buffer for the deferred cache-line do not seem to make up for the lost DCAS optimizations to common data structures.

5.7 Conclusions

I show that efficient Double−Compare−and−Swap functionality can be provided by a modest extension to current processor instruction sets. The simplicity of the instruction extension was demonstrated by describing two additional instructions for a MIPS-like instruction set, LLP/SCP, and their implementation. The new instructions build on existing mechanism for the LL/SC instructions that are already provided to support single CAS and locking functionality. These same techniques can be applied to other instruction set architectures as well.

Hardware complexity is reduced by depending upon functionality already provided in software. The operating system scheduling support ensures that no process, including interrupt routines, can starve another or cause excessive contention by high rates of preemption. Additionally, non-blocking algorithms typically include retry loops (that incorporate backoff to reduce contention) that can eliminate livelock. These mechanisms are required even with conventional synchronization and are a necessary part of operating system functionality. Providing this support in software avoids complicating the hardware support to (attempt to) address this problem.

I show that increased complexity of hardware implementations of higher arity primitives may arise in unexpected places, such as in constraints imposed on secondary caches. Such constraints can make the costs of implementing non-blocking primitives non-linear in the number of locations.

This non-linearity, coupled with the sufficiency of 2-location primitives, makes `DCAS` the "sweet spot" for hardware implementation.

While experimenting with simulated hardware alternatives, I also demonstrate through argument and measurement that a commonly applied optimization to synchronization software is *not* generally applicable to non-blocking primitives. Protecting `CAS` by a local test is unlikely to yield significant performance improvements because in the case of non-blocking synchronization the test is repeated (expensively) for cache misses, while in spin-locks the test is repeated (cheaply) for cache hits.

I show that my proposed hardware mechanism is faster, safer and degrades more gracefully under contention than a software implementation. In particular, contention on shared global locks for synchronization imposes a significant cost, because even though the access rate for individual data structures might be low, the cumulative rate on the shared lock grows quickly.

This work identifies an important example of carefully dividing functionality between hardware and software. The hardware implementation of `DCAS` is simplified because starvation and deadlock avoidance are implemented in software. The software implementation of non-blocking applications is simplified because atomicity is implemented in hardware. More generally, without hardware `DCAS` support, software ends up being slow, complex and fault-intolerant, either because it uses blocking synchronization techniques or because it attempts to implement non-blocking synchronization in terms of limited primitives like single `CAS`. However, with more complex hardware support than that required for `DCAS`, the hardware is more expensive, complicated and possibly slower than necessary, given that `DCAS` is actually sufficient for software needs.

I conclude that the modest architectural support for `DCAS` that I propose here is an important *missing link* necessary to make non-blocking synchronization practical. As the rest of this thesis argues, non-blocking synchronization is itself an important enabling technology for the efficient development and deployment of concurrent and parallel software systems in general.

Chapter 6

Conclusions

This thesis demonstrates that a modest amount of hardware support and a small set of software structuring techniques allow efficient non-blocking synchronization of concurrent systems. Non-blocking synchronization avoids significant problems with blocking synchronization such as deadlock and priority inversion.

An efficient implementation of DCAS goes a long way towards eliminating the performance problems and conceptual difficulty with non-blocking algorithms. No efficient non-blocking implementations of critical data-structures are known without DCAS support. As noted in Chapter 4, DCAS is sufficient to implement all performance critical shared data structures we have encountered in practice. Further, as shown in Chapter 3, DCAS enables space-efficient and asymptotically optimal worst-case time algorithms for universal transformations. Such transformations allow programmers to write easy to understand sequential specifications and automatically transform them to equivalent non-blocking concurrent implementations. The identification of DCAS arose from asking a different question than most other researchers. Rather than asking "What are the best non-blocking algorithms we can design with the current primitives?", I ask "What are the best primitives we can use to design non-blocking algorithms?"

Another contribution of this thesis is the demonstration that hardware DCAS is implementable on RISC architectures, given existing support for single CAS functionality. Chapter 5 describes my proposal in detail. Partly, the increased practicality of my approach over past approaches is related to the identification of DCAS as the ideal functionality — not trying to implement too much. Partly, it is the realization that much of the complexity can be deferred to software — indeed it *must* be deferred to software. Solving problems (such as starvation) in hardware, would *still* require software solutions. Software solutions obviate the need for a hardware solution (as long as the hardware instruction is deadlock free). Therefore, the hardware implementation of starvation avoidance is superfluous.

Efficiently supported DCAS would allow fully-synchronized standard libraries and operating system software to be portable across multiprocessors and uniprocessors without extra overhead or code complication. It would allow parallel architectures to use software developed for uniprocessors, relying on the (non-blocking) synchronization required for signals to handle serialization in the parallel processing context.

Finally, NBS can improve system structure. This is counter to the folk wisdom that non-blocking synchronization introduces excessive complexity. Chapter 4 describes our experience using NBS in the Cache Kernel, and describes the system benefits that accrue through the use of NBS. The (mis)perception that NBS increases system complexity is partly attributable to unfamiliarity, and partly due to real differences between blocking and non-blocking synchronization.

Non-blocking algorithms pay a price in *local* complexity to avoid impact on concurrent processes and non-related functions. Blocking algorithms. in contrast, are locally simple, but may pay a high non-local cost in awkward system structure. Examples of this non-local impact abound: the need for different forms of synchronization in interrupt and signal handlers, deadlock avoidance and detection, and priority promotion, to name a few.

Our experience to date convinces me that the non-blocking approach is an attractive and practical way to structure operating system software. Locks will become more problematic as signals are used more extensively in libraries, synchronization becomes finer grained, and as the cost of memory delays and descheduling become even higher relative to processor speed.

6.1 General directions for future research

This thesis demonstrated DCAS-based algorithms that had better performance characteristics than equivalent CAS-based algorithms. An interesting area for future work is understanding the *fundamental* differences between CAS, DCAS, and CAS3. This will help explain why DCAS enables direct implementations of so many data-structures over single CAS, and why so few extra implementations are possible when adding an $O(1)$ CAS3.

Simpler and better algorithms can be designed when we can make stronger assumptions about properties of the system. This thesis identified a set of techniques that are beneficial to systems in general, and that also enable efficient NBS. A promising direction for future research lies in searching for other common properties that can be exploited to improve algorithms.

Further work is required to validate our experience that DCAS is in fact adequate in practice for the vast majority of shared data-structures. Part of my argument is based on the assumption that systems that are well designed to avoid memory contention will require only simple shared data structures. Verifying this, too, is not simply taking a census of data-structures in existing distributed applications. Rather, the evaluation must take place *after* the applications are suitably restructured for reasonable performance.

However, this evaluation, in turn, rests on the assumption that as concurrent programs mature, they will, in fact, be (re)designed to avoid memory contention. Over time we will see whether this assumption is well-founded.

More experience is needed converting real systems to be non-blocking. While it is tempting to simply take existing distributed applications (such as programs comprising the SPLASH benchmark) and convert them to be strictly non-blocking, such tests would not validate one of the central arguments of Chapter 4 — that NBS positively affects system design. To effectively add data-points to further test my claims, more systems must be designed with NBS from the start. This seems feasible to do in conjunction with researchers in other fields such as molecular dynamics. Many fields of computational science are in the process of moving fine grained simulations to both parallel supercomputers and networks of loosely coupled personal computers using distributed shared memory.

Systems running on loosely coupled machines have another desirable attribute from the point of view of investigating the interaction between NBS and system structure. Machines that use *software* distributed shared memory can easily be extended to support atomic DCAS. Atomicity can be ensured locally by disabling preemption or techniques such as restartable atomic sequences. Atomicity can be ensured with respect to other processors by simply not responding to communication for the duration of the operation. This avoids the scalability issues of the shared locks in previous OS-based approaches. Thus, DCAS can be deployed on such machines far earlier than waiting for a new instruction on future processors.

Software distributed shared memory will provide useful platforms for research in a variety of promising areas. It will provide useful data about shared data types, contention, and whether NBS continues to have beneficial effects on system structure. NBS on top of software DSM will not, however, provide new data on the sufficiency of software versus hardware implementations.

Work is required to further evaluate the merits of hardware support for DCAS versus various software alternatives, particularly for overall system performance. Many distributed applications are not yet well optimized for multi-processors. If the frequency of synchronization primitives is low after any needed restructuring, then the total system performance of systems implemented on top of software or OS primitives may not be significantly worse than implementations on top of hardware primitives. My conjecture is that frequency will remain high (although contention will be reduced), because the most common examples of successful distributed applications to date seem to be structurally similar to fine-grained simulation where synchronization costs incur 25 to 70% of the execution time, where 90% of the code is inside critical sections, and where the dominant wall-clock-time costs are wait-time at barriers and locks. Thus, I predict that even after I evaluate highly tuned future distributed applications and systems, the benefits of hardware implementation of binary primitives will outweigh the costs. At present this is just conjecture, and must be verified.

An interesting area of further research is to explore an efficient hardware implementation of `DCAS` in the presence of speculation. Similarly, it would be useful to investigate schemes in which each instruction in the `LLP/SCP` sequence can execute in a single cycle when it hits in the cache. Such performance would be attractive for an implementation with zero-overhead when there is no contention or sharing. It would not just be attractive, but would become *necessary* as the frequency of `LLP/SCP` increased and the cost of disabling speculation after `LL/SC` also increased.

6.2 The future of NBS

A fair question to ask is whether NBS will ultimately become common, or even supplant blocking synchronization. Software designers are unlikely to use NBS without additional support, such as efficient `DCAS`. Hardware designers are unlikely to provide efficient `DCAS` unless a reasonable body of software uses `DCAS` functionality. It is reasonable to fear an insurmountable chicken and egg problem here.

Fortunately, a scalable OS implementation is possible in software DSM systems. Such platforms are becoming more common. In a similar vein, for very small scale multiprocessors and uniprocessors, Bershad's technique may be adequate. Consequently non-blocking algorithms will have an arena in which to demonstrate their value prior to availability of hardware `DCAS`.

Interpreted languages also offer the potential to implement `DCAS` under the covers. Java, in particular, is a prime candidate for such an implementation. First, the syntax of synchronized methods can equally hide a lock or a non-blocking algorithm using universal constructions. Second, early Java implementations have had difficulties dealing with data-structures held by failed threads. They have gone so far as to "deprecate the use of" any operation that can kill a thread. Third, the overhead of interpreting non-compiled Java may mask the extra cost of `DCAS` compared to regular memory operations.

The possibility of non-blocking software that can use DCAS, and that can exist prior to hardware support for DCAS, leaves me hopeful that demand for DCAS can eventually be demonstrated to hardware manufacturers.

I am actively working to convince processor designers to add DCAS support to commercially available microprocessors. I hope that our work and that of others can lead to a broad consensus on this direction. Ultimately, I hope that application and system software developers can safely count on this hardware and operating system support for non-blocking synchronization, allowing them to write software that is reusable across single process, parallel process, and signal handling applications with simplicity, efficiency and fault-tolerance.

6.3 Perspective: A Continuum of Transactions

Non-blocking synchronization allows a unified approach to concurrent programming. The transactional programming model (atomic transactions with full ACID semantics) dominates concurrent programming of distributed databases and distributed applications in general. Within a single shared memory, however, mutual exclusion is a far more common way to control concurrency. At the low level, where individual instructions must deal with concurrency, the instructions are again atomic, with a transactional flavor.

At the high level, in distributed systems, there is rough consensus that atomic transactions are the best way to deal with concurrency and to implement scalable, robust, fault-tolerant systems. Atomic transactions bear a certain performance cost over more ad hoc approaches, but the benefits in fault-tolerance, recovery, and ease-of-reasoning have long been considered to be worth that cost. Atomic transactions are intended to support the ACID semantics: atomicity, consistency, isolation, and durability. Additionally, some transaction systems behave much like the universal constructions discussed in this thesis: the programmer need only write a sequential program and the system automatically converts the code into an atomic transaction.

At the instruction level, too, we seem to have converged upon atomic transactions as the best way of dealing with concurrency. The synchronization primitives, Compare-and-Swap or Load-linked/Store-Conditional, that deal with concurrency are very similar to "mini atomic transactions". They are not full atomic transactions because, in the absence of NVRAM, they do not support *durability*. However, these instruction sequences are *atomic*: they either complete totally successully or abort with absolutely no effects. They support *isolation* or *serializability*: while executing, no partial state is visible to other processes or processors. At the level of instructions they are also *consistent*: the newly computed state is guaranteed to be a valid successor of the previous state, or else the “transaction” fails.

It is only in the (large) gap between instructions and high level atomic transactions that we resort to non-transactional programming models. Non-blocking synchronization fills this gap by provides atomicity, isolation, and consistency. Recent work (e.g. Rio Vista [61]) has shown how to provide atomicity and durability very efficiently if a small amount of non-volatile memory is available. Efficient NBS is the missing piece to provide the full ACID semantics, and thus atomic transactions, across the entire spectrum of programming systems.

6.4 Concluding remarks

Blocking synchronization is filled with problems and pitfalls. It seems a tractable technique to contemporary practitioners only because we have spent years of research attacking these problems. As solutions have been proposed, we have then had years of practice deploying and internalizing these solutions.

Non-blocking synchronization has clear advantages over blocking synchronization. It is worth a small cost to purchase these advantages. If the performance cost (or the conceptual complexity) is too large, then NBS will not be deployable — the general perception is that NBS is too expensive except for a small number of specialized data structures. This thesis demonstrated that given the right infrastructure this perception is wrong — non-blocking synchronization can be implemented cheaply for a large class of interesting systems (for example, systems designed to minimize memory contention).

I hope that the work presented in my thesis, if adopted by hardware and system designers, will make NBS a practical approach for all synchronization needs. That, in turn, will be one of the many steps towards removing the software bottleneck that has slowed the wide-spread deployment of scalable, fault-tolerant, parallel applications.

Appendix A

Common Myths and Misconceptions about NBS

This appendix lists several common misconceptions about non-blocking synchronization.

A.1 Non-blocking algorithms never block

The first misconception about non-blocking synchronization has to do with the phrase "non-blocking". An algorithm is non-blocking if the system, as a whole, never blocks. Individual processes, however, may block. Some processes block indefinitely due to starvation. Some processes block temporarily while waiting for another computation to finish.

On the other hand, processes do not block waiting for processes that are not making progress. In a non-blocking system, if one process is blocked then some other (related) process must be running.

A.2 NBS increases concurrency

The second point that needs addressing is the confusion between optimistic locking, highly concurrent data-structures, and NBS. NBS does *not* automatically increase concurrency. Optimistic locking is *not* (in general) non-blocking.

Optimistic locking reads state and optimistically assumes there will be no contention for the lock. The general approach is to read all the necessary state words *without* (or before) acquiring a lock. Then, perform the computation locally, and finally, *with the lock held*, check that the initial state is unchanged (*validate* the computation). If the state is unchanged, then the computation is still valid, and one may make the modifications dictated by the computation (with the lock still held). The lock hold time is reduced since the computation is done outside the lock. Note, though, that if a process is delayed while it holds the lock, it still delays all other processes and it is not tolerant of failures.

Optimistic locking reduces the window of inconsistency (although the inconsistency still spans multiple writes). Optimistic locking is also useful when the delay to acquire a lock is large; the optimistic computation can proceed in parallel with lock acquisition.

Highly concurrent data structures are (mostly) independent of the method of synchronization. Highly concurrent data structures reduce the granularity of synchronization and, where possible, separate state into read-only sub-objects that can support multiple simultaneous readers. However, both locks and NBS impose a total order on operations on a data structure. To see this, recall that most non-blocking algorithms require that values are not modified by other transactions from the time the locations are read until they are modified (possibly by `CAS` or `DCAS`). Therefore in order for a transaction, T, to succeed, it must have exclusive access to those locations during the interval. (Clearly if any other transaction wrote those locations during the interval, then T would fail. Further, if any other transaction *read* those locations during the interval, *it* would fail after T modified those locations.) So, although NBS has many advantages over blocking synchronization, automatically increasing concurrency is not one of them.

To explain this another way, it is possible (up to deadlock avoidance and fault-tolerance) to convert any concurrent algorithm that uses NBS to an equivalently concurrent algorithm using locks. Simply assign a lock per location referenced in the NBS algorithm and acquire each lock before the first access to its corresponding location and release them after the last access. (NBS obviously makes this easier by eliminating locks and using the data locations themselves for synchronization. The locking version will be cumbersome and expensive and suffer from the many other problems that plague blocking synchronization but it will exhibit just as much concurrency.)

This is consistent with the second law of concurrency control mentioned by Jim Gray "Concurrent execution should not have lower throughput or much higher response times than serial execution."([32], page 377).

A.3 Concurrency is good

The third truism that needs qualification is that increased concurrency is, in and of itself, a good thing.

In practice, we have found that actual contention on most data structures is low. Care must be taken that increasing the *potential* for concurrency under contention does not increase the cost of the common case where there is neither contention nor concurrency.

Even in cases where multiple processes contend for accesss to a single shared data-structure, increasing concurrency is not always beneficial. The naive view is that by doubling the number of processes allowed to concurrently operate on a data structure we can roughly double the throughput of the system. This ignores the cost of communication and remote memory accesses, and the costs due to communication contention and memory contention. Increased parallelism is a good thing, but not necessarily at the cost of increased contention.

Care must be taken that the costs invested in increasing concurrency are justified by the benefits we can reap in each particular case. If a data structure is accessed serially by processes on separate processors then we can expect the second processor to take several cache misses. Each cache miss can take hundreds of times the memory latency of a cache-hit (and the gap is growing as processor speeds increase relative to memory speeds).

Mutual exclusion eliminates *simultaneous* access to shared locations (other than, perhaps, the lock word itself). However, even reduced contention does not eliminate cache misses. Consider processor P_1 operating on shared cache line A. Even if processor P_2 does not access A until P_1 has finished, it will still take cache misses — or acquire the cache lines some other way. The key is not simply to reduce simultaneous contention, but to design algorithms such that data remains local to a given processor (more precisely, so that data can remain in a local processor's cache).

Memory costs must be borne in mind when considering properties such as *disjoint-access-parallelism* [47] (discussed more fully in Section 3.2.5.1). An algorithm is disjoint-access parallel if two transactions that access disjoint sets of locations can execute concurrently without interference. Clearly, any algorithm that paid any price for increased concurrency within a single cache line is flawed (the memory system will serialize the transactions in any case). Often, concurrency over larger blocks of memory is detrimental, too, because there are other algorithmic reasons that access is serialized. Any cost paid to make inherently serialized accesses concurrent is wasted. Finally, even if concurrent access is possible, before trying to implement disjoint-access-parallelism, care must be taken to determine the appropriate granularity of the concurrency lest you pay a price only to "benefit" by damaging performance due to memory contention.

A.4 Complicated data structures need to be shared

The key to reducing memory costs is to design algorithms such that data remains tied to a single processor's cache. It follows that well-designed concurrent algorithms will have processors *mostly* operating on local data-structures — computations will be structured so that (ideally) the parallel portions all execute out of local memory. So, *shared* data structures are mostly used for communication and management of parallel tasks working towards a common goal, but operating on (mostly) local data structures.

Most complex data structures are used as part of algorithms where asymptotic behavior is important (else simple data structures with small constant factors would be sufficient). Given a system structure where parallel computations are executing on *local* objects, most of these more complicated objects will be part of the local computations. Simple objects should be sufficient for communication and management, else the problem was probably structured inefficiently — the memory and communication costs are probably outweighing the advantages of this degree of parallelism. Fortunately, given `DCAS`, most simple objects have relatively efficient non-blocking implementations.

It is conceivable that a small minority of algorithms might require *sharing* of complex data structures. However, major justification is required to overcome the costs of communication and memory traffic.

A.5 NBS improves performance

NBS can occasionally improve system throughput by eliminating cases where many processes wait for a stalled process. In the (common) absence of such delays it is *not* the primary goal of NBS to improve average-case performance. NBS simplifies system design, and provides fault-isolation and fault-tolerance. The question is whether the performance cost of such benefits is too expensive. Therefore, the performance goal of NBS is not to improve over locks, but to be *comparable* to the equivalent locking implementation in the common case when there are no delays.

There are generally two comparisons we need to make. First, we must compare the constant overhead of the synchronization in the common case of no contention. Second, we must make sure that the system doesn't degrade badly (e.g. non-linearly) in the presence of multiple processes.

Linear degradation is the "natural" limit of acceptable performance hit. If the performance cost of NBS is linear in the number of processes, then the total run-time of the non-blocking concurrent algorithm running on p processors is at worst (up to big O) equivalent to running the sequential version on a single processor.

If it happens, as it sometimes does with simple data structures, that a non-blocking implementation out-performs all blocking implementations, that is a lucky circumstance, but is not a goal.

It is important to bear in mind that the percentage of time operating on the shared data structure is likely to be low compared to the time operating on the private data structures. Thus, while access to the shared code shows no improvement in worst-case time complexity, the private operations can be totally overlapped and may exhibit up to a factor of p speedup if the parallelism exists.

A.6 Locks are not non-blocking

A "lock" is simply a convention that grants a particular process exclusive ownership over some abstract object. The common implementations of protocols that use locks are not non-blocking, because if a process is delayed or dies while holding a lock, all accesses to the locked object must wait, or block, until the process releases the lock, relinquishing ownership. However, as described in Chapters 2, 3, and Appendix B, preemptible or interruptible locks exist. There is no reason that non-blocking algorithms cannot be implemented using such locks — in fact, some are [52, 99].

These were some commonly held misconceptions about non-blocking synchronization. Shedding these preconceptions will make some of my assumptions clearer to you while reading the rest of this thesis.

Appendix B

Taxonomy of Universal Constructions

Any non-blocking universal transformation in a system where the basic primitives are of arity k must be able to deal with algorithms that update $N > k$ locations. This means that there must be some point at which the data-structure being updated is in an inconsistent intermediate state (although perhaps internally consistent), and cannot be immediately taken over by another transaction without some additional work.

We characterize all universal transformations by their approach to three issues dealing with intermediate state.

B.1 Update-in-place vs. local-copy

The first subdivision is between algorithms that update data structures in-place, and between those that update local copies. Algorithms that operate on local copies may divide the data structure into *blocks* in order to reduce the amount of data that must be copied. Blocks may (or may not) vary in size or number — a *block* is simply the unit of copying. Any attempt at modification within a block triggers a copy of the entire enclosing block, and the modification is made to the local copy.

On every `read`, the algorithm tests whether the location is already part of a local block or not. If it is part of a local block, the value is read from the local copy. If not, the value should be read out of the original data structure. On every `write`, the algorithm tests whether the location is already part of a local block or not. If not, the enclosing block is copied from the original data structure to a local copy. The new value is written to the local copy.

Upon completion of the update, the algorithm must atomically replace all changed locations at once. Some data structures consist of blocks and pointers, and possess a single root. For such

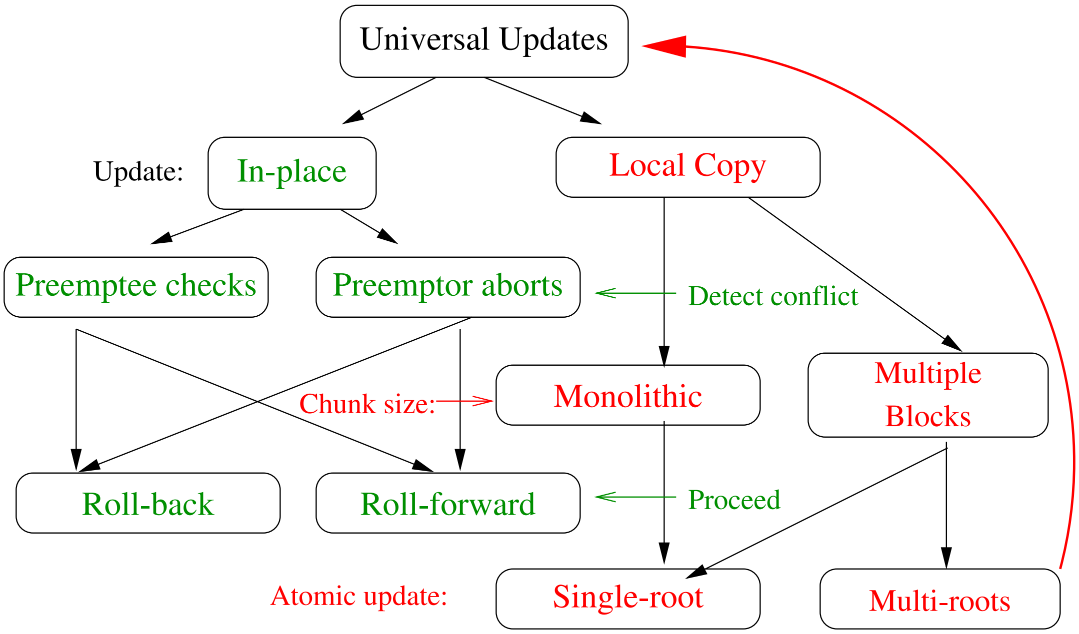

Figure B.1: Relationships between categories of update algorithms.

data-structures, atomically swapping in the new root is sufficient. For universal algorithms that transparently copy and update arbitrary local blocks, the equivalent of CASn must be used to simultaneously update all modified roots at once. In the extreme case, where a table of root pointers to *every* block is used, then CASn atomically replaces all modified blocks at once.

Updating local copies incurs the overhead of copying at least W locations for a transaction that modifies W locations. In addition, it incurs a lookup cost for each read or write to check whether the transaction should access a local copy or the original data structure. By copying larger chunks than individual words you can reduce the cost of the lookup, but you increase the amount you copy. This additional overhead is not dependent on the arity primitives used to implement the NBS algorithms; it is a fundamental cost of operating on local copies. *All other things being equal, we would prefer to update in place rather than to operate on local copies in order to avoid the copying and lookup costs.*[1] (Of course, in the unlikely case that the vast majority of locations in a data structure are written in a given update, then copying the entire data structure might be appropriate. This increases the copying cost (which must be paid regardless), but completely eliminates the lookup cost).

[1] Gray[32] and Reed[82] both argue that update-in-place is a worse form of update than copying. They are not talking about the same classification as we are; the alternative to update-in-place is version storage, a record of the complete object history. In their terminology, *both* approaches I discuss are “update-in-place”. To them, the approach I refer to as copying is just a particularly inefficient form of update-in-place.

It is instructive to note that a distinction still exists between updating-local-copies and updating-in-place even when the block size is a single word. In the local-copy case, we still must determine whether references are to local copies, we still need to connect words by pointers, and there is almost certainly space overhead for *every* word in your memory.

Updating-in-place has associated costs, too, however. The data structure is potentially logically inconsistent during the update. This means that if another transaction wants to proceed it must somehow get the data structure into a consistent state. (Algorithms that operate on local-copies have this problem only during a brief window while doing the final `CAS`n (or equivalent)). Therefore update-in-place algorithms must always pay the cost of keeping the data structure in a state where it *can* be returned to a consistent state. In practice, however, such costs seem small compared to the copying and lookup costs of the update-local-copy algorithms. In the uncommon event that a transaction is stalled and another attempts to proceed, the second transaction must actually return the data structure to a consistent state as outlined in Section B.3. I assume, however, that contention is low (contention reduction techniques help ensure that this is true) and that long delays are infrequent and expensive. Therefore the cost of actually returning to a consistent state is incurred infrequently, and can be mostly ignored. We still must always consider the bookkeeping costs to keep the data-structure in a state where it *can* be restored, because this cost is borne in the common case.

Algorithms that update in place can themselves be subdivided based upon their behavior when two transactions conflict. There are two distinct problems that must be solved. The first problem is detecting transaction conflicts. The second problem is how to proceed when the preempting transaction wishes to operate on a data structure that is in an intermediate state.

B.2 Detecting conflicts between transactions

Two different sides must detect a transaction conflict. First, a transaction, P_2, wishing to access a data structure, D, that is already part of another transaction, P_1, (which has possibly stalled or failed) must detect the fact that D is already "owned" by P_1, and either back-off and try again, abort, or "take over" the data structure. This first case is trivially detected by conventionally marking a data structure as currently being part of a transaction (and, hence, potentially inconsistent). This is the approach taken when using locks (This first case of conflict detection is also applicable to the copying protocols described above; however it is merely used to reduce contention and not to protect the integrity of the data structure). It is a well understood technique.

However, in non-blocking systems, "ownership" can at best be only advisory, it cannot be permanent (else an owner can delay all other processes in the system). This gives rise to the second case of conflict detection. If transaction P_1 were delayed while owning a data structure, then transaction P_2 may "take over" ownership of the data structure. P_2 is the owner, but P_1 still *believes* it is the owner. Similarly, in non-blocking systems that eschew the notion of ownership, multiple transactions will attempt to proceed in parallel. In such cases, even though P_2 can legally proceed, and even though it has already begun updating D, P_2 must still detect whether another transaction, P_3, has taken over that data-structure. P_2 must detect this, since once P_3 has acquired D, P_2 must be careful to avoid making any forbidden modifications.

There are two general approaches to detect the fact that a data structure has changed "underneath you", so to speak. The first is to maintain some indication of the current status of the data structure and atomically check this simultaneously with each write. The second is to depend upon some operating system support to allow an external agent (e.g. the operating system or the preempting transaction) to alter the flow of control of the preempted transaction upon conflict.

The drawback with checking on each write is the potential performance cost. Using single `CAS` either some indication of the state of the transaction must be co-located with every location, roughly doubling the size of the data structure, or else the status check and the write are not performed atomically. In the latter case, there is always the possibility of a change of status between the check and the write, so a more complicated and expensive protocol must be used involving multiple phases and considerable expense.

Using `DCAS` a single state word can be associated with each transaction and with each data structure thus reducing the space requirements. `DCAS` allows the check and write to always be done atomically, eliminating the need for any further protocol. There is a cost, but it is directly proportional to the number of writes, independent of the number of reads and/or the number of active processes.

An alternate approach, aimed at reducing the cost of expensive atomic `CAS` and `DCAS` is to depend on active OS support. The basic idea is to protect the integrity of the data structure using locks, but avoid or recover from long delays. The first general method using this approach is to use OS support to reduce the likelihood of long delays by prohibiting preemption while a lock is held. This is not always possible (some preemptions, such as those triggered by page-faults, are coerced), and not desirable since it provides a loophole through which processes can gain an unfair share of the machine. Further, it is not fault tolerant (what happens if a process fails while holding a lock?), and not sufficient, since, if you *are* preempted the recovery code must be the same as in systems without such OS support. A variant of this approach is useful as a contention reduction technique (depend upon the OS to release the lock if you are preempted), but it is not sufficient to protect data integrity and provide non-blocking implementations.

The second general method of depending upon active OS support is to aid the preemptor, rather than the preemptee. The goal is, once again, to provide an efficient alternative that allows us to use normal reads and writes rather than expensive atomic `CAS` or `DCAS`. The basic idea is to use a lock to protect the data structure and use normal reads and writes while holding the lock. If the preemptor wishes to access a locked data structure, it must (a) inform the lock-owner (when it resumes) that it no longer holds the lock, and (b) it must transform the modified data-structure to a consistent state. Assuming no copying, this requires knowing *exactly* where the lock-owner stopped operating on the data-structure, in order to know precisely which locations have been modified. Therefore, the lock-owner must maintain this state. If we just use normal reads and writes, then care must be taken to avoid race conditions between logging each modification and altering the location in memory. One would normally not use `CAS` or `DCAS` instead, because the point of using locks was to avoid such expensive operations.

There are several drawbacks with depending on active OS support for either the preemptor or the preemptee.

First, the OS approach must treat asynchronous handlers (interrupts or signals) specially. Either interrupts or signals must be disabled while the lock is held, or else we must treat each interrupt as a context switch and check whether we were preempted. A related (but less significant) consequence of this is a loss of portability for code that might interlock with asynchronous handlers.

Second, the use of locks (even preemptible, advisory, locks) re-introduces problems of potential deadlock. Deadlock avoidance is possible, but increases the complexity of the code.

Third, the OS approach is less fault-tolerant than solely `CAS` or `DCAS` based approaches. The OS approach fails if the OS does not release a lock held on a particular processor. However, a study by Jim Gray [33] claims that 90% of failures are OS failures, and not hardware failures. Thus, it is far more reasonable to assume that the memory system on a failed node is consistent than it is to assume that the OS will continue to function correctly on that node.

Fourth, the OS approach may be less portable than the `DCAS` approach. The interface and specification of `Double-Compare-and-Swap` functionality is straightforward, and easy to encapsulate in an inline routine. The OS support required here is more complex, harder to specify, and less likely to be standardized. On the other hand, if support for a facility similar to Scheduler Activations [10] became standardized and universally deployed, it may be sufficient to make the OS approach portable.

Fifth, the OS approach seems to be less modular and less secure than the application level approach. All the OS approaches require some modification to the scheduler's behavior. For reasons of security and fairness it is preferable that algorithms react and respond to decisions of the scheduler, rather than alter the *behavior* of the scheduler. Thus setting your own priority is reasonable, but altering decisions (preempting other processes or inhibiting preemption of oneself) is undesirable.

Further, for safety and modularity it is best if the scheduler need not access external data-structures (especially not in user space), nor have any knowledge of internal process actions (including synchronization). Reading information that the scheduler itself chooses to export (e.g. current process) is safe, but requiring the scheduler to clear locks in the process's private address space is not.

For all these reasons, unless the performance issues are overriding, *checking for transaction conflicts is preferable to depending upon active OS support*.

B.3 How to proceed when conflict is detected

Regardless of whether one detects conflicts by OS support or by checking each write, in the event that a conflict *is* detected, the preemptor must decide whether to proceed or to wait. In local-copy protocols this decision only affects performance due to contention, because each process operates on its own local copy. Even in Allemany and Felten's SOLO protocol [2], which is a copying protocol hiding in update-in-place clothing, the preemptor makes a local copy before acquiring ownership. In update-in-place protocols only one process may operate at a time, so proceeding is a weightier decision. Before proceeding, a process must find some way of preventing the preemptee from making any future modifications.

Still, in general the decision whether to proceed is straightforward: it makes sense to proceed only if the original process is not making any progress. If it is making progress, it behooves us to wait for it to complete. The question is how to determine whether the process is making progress or not. One can either guess or one can know (subject to certain assumptions). Guesses involve heuristics (such as the amount of time since we last know the process did something, or waiting a reasonable amount of time until we assume it *should* be finished if it were making progress). These guesses are not guaranteed to be correct, but should be correct with high probability. The basic assumption needed in order to *know* that a process is making progress is to assume that the code is correct; therefore if it is currently running on some processor then it is making progress. Therefore "knowing" whether a process is making progress comes down to knowing whether it is currently running. Either the owning process (with active OS assistance) can record when it is preempted (a'la Allemany and Felten), or the process attempting to take over can check (through passive OS assistance) whether the current owner is still running.

Assume the preemptor knows or guesses that the owner is stalled, and wants to proceed despite detecting a conflict with the preemptee. If so, it must restore the data structure to some consistent state. The preemptor can either *roll-forward* the original transaction to completion, getting to a new consistent state, or else it can *roll-back* the original transaction to a previous consistent state[2].

Roll-forward algorithms are implemented by encoding the desired operation in the shared data-structure so that competing transactions can complete stalled partial transactions ("helping") before beginning their own operation. Unfortunately, to implement roll-forward in an update-in-place universal transformation (where the mechanical translation is done with no specific knowledge of the original sequential algorithm), one is either required to know the behavior of the transaction *in advance*, or else perform some form of process migration from a stalled process to the currently active process.

One way to establish (in advance of the transaction's operations on shared memory) which operations the "helper" needs to perform is by running the transaction twice (running it once and caching the reads and writes) — but this is effectively updating a local copy with all the associated costs. At the other extreme is process migration: if a process is stalled, we migrate the process to the processor running the preempting transaction. Process migration, however, is tricky to implement at best, is not possible in many cases if the original process has terminated, assumes homogenous architectures (or else pay the cost of interpretation), and can be costly. In practical terms it is not clear whether it is better to wait for the original process to resume, or to perform the migration (when the first transaction completes we need to restore control to the *original* process, and simultaneously proceed with the second transaction in the preempting process). For this thesis, in order to compare against algorithms which employ helper functions, we ignore the details of exactly how one constructs a particular helper function, and simply assume that one is produced (possibly expensively).

An alternative that lies between local copy and process migration is possible when the algorithm

[2]It is important to note that my use of the term "roll-back" is significantly different than aborting a transaction because some other transaction has completed successfully. In the cases where progress has already been made, roll-forward of the incomplete transaction is generally not even possible. Second, when some other transaction has completed, cleaning up (sometimes called "rolling back") the partial state of the incomplete transaction, does *not* restore the data-structure to the *previous* state. It is transformed to the state *after* the other transaction has finished. When Moir [74] and others refer to "roll-back", they use it in the sense of cleaning up an aborted transaction after progress has been made by another transaction.

is specialized for a particular data structure. The basic requirement is to be able to register some high level notation specifying the desired operation, and some shared location describing the current state of the computation. The implementation of CASn described in Section 3.2.1.1 is a concrete example of this approach. A CASn operation is fully determined by the initial list of arguments, and the state is simply an index recording which arguments we are processing (the i-th CAS) and some bits recording whether the combined CASn has been successful so far or has already failed.

Appendix D shows examples of algorithms using helping in data-structure specific implementations — that is, not in universal constructions. No code is passed around, no helper function is used, but the algorithm itself is written in such a way that it can incorporate helping. Examples of this approach are the doubly linked lists of Section D.7 and the heap implementation of Israeli et al. [46]. Unfortunately, while these are good solutions, the approach depended on careful algorithmic design. These are no longer universal constructions that are oblivious to the emulated algorithms. They require extensive (human?) pre-processing. Given that one of the goals of the universal constructions is to avoid placing a conceptual burden on programmers, it is questionable whether this technique is reasonable in general, although it is certainly useful for *some* specific data-structures which can be incorporated in code libraries.

In addition to the cost and complexity of constructing helper functions, they suffer other drawbacks. First, if the original process was delayed quasi-deterministically (for example, because of a page fault), then the "helper" will also be delayed, and no progress will be made (in fact, the likelihood of a fault might get worse if you switch to another processor). More generally, any argument or data dependent fault (protection fault, algorithm bug) will eventually occur in *every* process accessing the shared data structure as it tries to help the failing transaction. This is precisely the opposite of the intended effect of non-blocking algorithms — one of the goals of non-blocking algorithms is to isolate faults.

Second, algorithms that depend upon helper functions implicitly assume that helper functions are context-free. This may not always be an unreasonable assumption, but care still must be taken to avoid global variables, and to execute the helper in the context of the original process in order to respond appropriately to environmental queries, and to acquire the protection rights of the helped process. Finally, when synchronizing with asynchronous handlers, helpers must be able to be executed by the handler (interrupt handlers are a special case of this tricky problem).

Third, roll-forward implicitly assumes that all operations are of equal priority. More specifically, any algorithms that exclusively uses helper functions determine the order of operations at the time a transaction "acquires ownership" of (i.e. registers their helper function in) a data structure.

Roll-forward algorithms behave like algorithms that use locks: once you begin an update, you are committed to completing it. In Chapter 1 I enumerate the advantages of non-blocking synchronization over blocking synchronization. One of the key advantages listed there is the fact that NBS eliminates priority inversion. More generally, NBS can eliminate synchronization interactions that subvert scheduler priorities. It is true that *any* non-blocking algorithm will avoid basic priority inversion: it is impossible for a medium priority process to delay a high priority by preempting a low priority lock-holding process. However, algorithms that strictly roll-forward cannot eliminate an entire class of scheduling problems: cases where transactions, rather than simply processes, have priorities. Consider a very, very, long transaction being executed on behalf of a very, very, low priority process. It is not always reasonable for the high priority process to first *finish* the long running low priority transaction. That itself might constitute an unacceptable delay for, say, an interrupt handler.

The alternative to roll-forward is to roll-back to a previous consistent state. Roll-back avoids all of the drawbacks associated with helping. However, roll-back of compound operations has one *big* drawback of its own: it can never be strictly non-blocking. Roll-back can never be non-blocking because there is no way to guarantee progress when roll-back returns the system to a previous state. The sequence of events that caused the system to roll-back can be endlessly repeated. In contrast, primitive atomic operations have the property that the only way a transaction can fail due to contention is if another transaction has succeeded. Roll-forward guarantees that transactions never fail solely due to contention.

Therefore, roll-back is only an acceptable choice if we are willing to depend upon probabilistic, rather than deterministic, guarantees of progress. If we choose to use roll-back, algorithms can be *effectively non-blocking*, but cannot be strictly non-blocking. Given the costs of roll-forward, and given that contention and preemption are both relatively infrequent, and given that effectively non-blocking algorithms make progress with probability 1, usually *roll-back is a more attractive option than roll-forward*. There are no issues of prophecy or prediction: one merely has to record what the transaction has already done. The extra cost over using locks is recording enough state so that any modifications can be undone. The odds of not making progress are low.

Appendix C

CASn is Non-Blocking and $O(n)$

C.1 CASn is non–blocking

In this section it is useful to remember that STM with roll-forward (and CASn in particular) always completes a transaction (with either success or failure) after a *single* transaction attempt. STM with roll–back may retry and take multiple attempts if the transaction attempt is undone, but had not finished.

Lemma 1 *Every write (in* `trans_write` *or* `CASnInternal`*) increments* `idx` *and every* `undo` *decrements* `idx`*. Nothing else changes* `idx`.

Lemma 2 *While a log is active (owner of any domains), all writes in the transaction precede all* `undo`*'s of that transaction attempt.*

Writes explicitly (and atomically) check that `state.undo` is FALSE, or else the write fails. `state.undo` is set to TRUE before `cleanup` performs any `undo`s. So, all writes must precede `cleanup` (and all `undo`s occur in `cleanup`). `cleanup` explicitly checks that `state.undo` is TRUE while undoing each log entry. If `state.undo` is no longer TRUE, then some other process has already cleaned up this transaction attempt, and `cleanup` returns.

The only time `undo` is reset to FALSE is by `trans_init`. `trans_init` is called once at the beginning of each transaction attempt, *before* any attempt is made to install `log` as owner of a domain. So the only writes that may follow an `undo` are in a *different* transaction attempt (the log must have been cleared from owner, and original values must have been restored). □

Lemma 3 *While a log is active (between being installed as owner and releasing ownership), each value of* `log->state` *is unique (each pair of values* `(idx, undo)` *occurs precisely once).*

It follows from Lemma 2 that there are no cycles between increments or decrements of `idx`. Thus, `state` takes on monotonically increasing integer values with `state.undo == 0`. If the transaction is aborted, it then goes through the same integer values, but with `state.undo == 1`. Each integer may be present at most twice, but each time with a different value of `undo`. □

Lemma 4 *The* `owner` *field is only cleared of* `log` *if the transaction which allocated* `log` *was finished and either completed or was aborted. (The transaction completed only if* `log->state` *records that* `finish == TRUE`*, and* `idx == max`*, and* `undo == FALSE`*. The transaction aborted if* `idx == -1` *and* `undo == TRUE`*).*

The pointer to the `log` in the `owner` field is only cleared to `NULL` by `trans_cleanup`. This function is only called in two places: `trans_cleanup` is either called by `trans_commit` or it is called immediately after a call to `trans_abort`. `commit` is only called when a transaction is complete, and therefore when `idx` is at its maximum possible value for this transaction (once `finish` or `undo` is set, any attempt at a write fails). `commit` sets `finish`, and `abort` sets `undo` and sets `idx == -1` after undoing. `cleanup` ensures that `idx == -1`. □

Note that the converse is not true. `idx` may be -1 while the `owner` field still contains `log`. For CASn, an aborted active log means a complete (but failed) transaction. There is no retry. This is also true of all the STM algorithms with "roll-forward", or "helping". STM with roll-back, however, must retry after an aborted active log (unless `finish` was set.) The proof sketch that the algorithm is non–blocking only applies to STM with roll-forward (CASn being one particular example) — it does not apply to STM with roll–back.

Lemma 5 *All references to* `log` *are either bracketed by* `trans_init` *and* `trans_conclude` *or by successful* `trans_begin`*s and* `trans_conclude`.

That `trans_init` and `trans_conclude` are matched is clear by inspection. However, `trans_begin` is a slightly trickier case.

If `trans_begin` returns `FALSE` when called, then `trans_open` (the caller) retries the loop. `trans_open` calls `trans_begin` again, before referencing any log. We still must show that if `trans_begin` returns `TRUE`, then the matching `trans_conclude` (called by `trans_open`) is called on the identical transaction that `trans_begin` was called on. Since the local variable `owner` does not change value, we must only establish that the log pointed to by `owner` is not deallocated and reallocated to a new transaction (possibly storing that identical `log` into `*domain` again). Note that for `trans_begin` to have returned `TRUE`, the `DCAS` in the `while` clause of `trans_begin` ensured that the reference count for `log` was incremented while `*domain == log`. (The `DCAS` is race-free). A log is only deallocated if the reference count goes to zero. There is no decrement corresponding to the increment in `trans_begin` other than `trans_conclude`. So, log cannot have been deallocated.

The indirection through `&(log->helperCount)` at (3.1:1) is *not* protected by the reference count. We depend on *type–stable memory management* [34] to ensure that the indirection is type-safe and does not raise errors, even if `log` were deallocated. We assume, without much further comment, a system with type-stable memory management. Section 4.2.3 and Appendix E briefly defend this assumption as reasonable (and that implementation is practical). Further, to my knowledge, every paper on non-blocking algorithms either implicitly or explicitly makes equivalent assumptions (e.g. [39]), pays a high cost in ensuring that objects are not deallocated [100], or else uses static allocation of all data structures (e.g. [74]). Thus, our results are comparable to previous work because TSM does not extend the basic model we are working with. □.

Lemma 5 depends upon type–stable memory management (TSM). To clarify the limits of our use of TSM (it is not a magic bullet), I note that we must check for `log == NULL` ourselves — TSM cannot help here. TSM only ensures type-safety, that is if the object pointed to by `log` was once an object of type `Log`, it is *still* an object of type `Log` as long as `log` has not been rewritten. Here, however, we are concerned whether the value of `log` was *ever* a valid `(Log *)`.

Lemma 6 `log` *can only be stored in the* `owner` *field once in any transaction attempt. No log from any other transaction can be stored in* `owner` *during this transaction attempt.*

The operative line of code is (3.2:2). `trans_start` can only successfully store a log in `owner` if the field currently contains NULL. By Lemma 4 it can only be set to NULL outside of a transaction attempt. □

Since a `log` cannot be re-allocated until it has been deallocated, and because `helperCount` can only be incremented if `*laddr == log`, and because `*laddr` is cleared before the *first* decrement of `helperCount`, and because `log` can be installed in `domain` at most once between allocations, then we know that once `helperCount == 0` it will stay 0. Then precisely one thread executes statement (3.1:4) so we can safely call `deleteLog` once we've seen a zero once.

Lemma 7 *A log cannot be freed and reallocated for a new transaction, or used in another transaction attempt, while any live references to it exist from an earlier transaction attempt.*

By our assumption that we are only dealing with roll-forward, the original caller never loops and retries. (`CASn` has no loop, and for STM with roll-forward, the transaction is guaranteed to be successful on the first iteration.) Thus the only possible reuse of a log must go through the free pool of logs.

The log is not deallocated and returned to the free pool until the reference count is 0. By Lemma 5 all uses of references to a log are bracketed by calls that increment and decrement a reference count. The reference count (`helperCount`) is positive while any references to `log` may be used. □

Lemma 8 *After a successful* `trans_start(domain, log)`*, if* `log->state.idx` *is non-negative then* `*domain == log`.

By Lemma 4 the log can be cleared from owner only if `idx` reaches the maximum value without undo, or is decremented to -1 by `trans_cleanup`. If the log is cleared without undo, then `trans_cleanup` atomically sets `idx` to -1 along with clearing the log from the owner field. So, regardless if committed or aborted, `idx == -1` immediately following the log being cleared. `log->state` (including `idx`, `undo`, and `finish`) is initially 0. By Lemma 3, states cannot repeat.

By Lemma 7 we know that if we reference `log` it cannot have been reallocated for another transaction attempt, and, thus, a non-negative value of `idx` guarantees that `log` is still stored in the `owner` field of the domain. □

Lemma 8 allows us to check (at (3.3:2)) `loc` and `&log->state` atomically, without checking `*domain`, yet still be guaranteed that `*domain == log`. Thus, `DCAS` can be used to simultaneously update the memory location and increment the index into the log, *and* ensure that the transaction has not been aborted.

TSM alone is not sufficient here. To ensure that Lemma 8 holds we need the explicit reference counts to make sure that log is not reused by another `CAS`n until no one is looking at `log.state`.

Lemma 9 *If the* `if` *clause (including the* `CAS`*) in* `trans_commit` *(at line 3.2:3) fails, then another process has already aborted or committed this transaction attempt and executed this statement.*

The `if` clause can only fail if `finish` were already set (only possible by some process executing this statement) or if `undo` were set because the transaction attempt were aborted. We know that `log` cannot be reallocated to another transaction during this routine by Lemma 7. □

Lemma 9 guarantees that any time a process reaches (3.2:3) either it can complete the transaction or someone else already has.

If the algorithm reaches commit (3.2:3) and has not yet failed, it marks `finish` so no other process tries to undo a successful transaction. (`force == TRUE` is used to allow a transaction to complete (not retry) even if `undo` is true.)

The lemmas above show that if processes follow this protocol, once a transaction installs a log as owner of a domain, no other transaction may access that domain, so transactions are not interleaved. Further, either all of the writes in a transaction complete, or they are all completely undone. If they complete and successfully mark `finish` using `CAS` at (3.2:3), then they are "committed" and cannot be undone. It is easy to see that a successful `CAS` here[1] is the point of linearization [44]. (In fact, the exclusive access given by *owning* the domain, enforces a strict sequential order.)

The lemmas have not yet, however, guaranteed that progress is made in finite time.

Lemma 10 *A transaction's* `log` *fails to be installed as owner over a domain only if another transaction has already acquired ownership.*

Lemma 11 *Assume the sequential specification of an algorithm completes in a finite number steps. Assume there are w writes to memory in this algorithm. Then using the protocol described here, each helper must complete within at most $2w$ "steps".*

The body of each helper function only interacts with the protocol through explicit calls to the function `trans_write`, or by folding in the `trans_write` functionality (as `CASnInternal` does). There are no loops in `trans_write`. There are at most w calls to `trans_write`. If the transaction is undone then each of these w writes may also be undone.

All helpers are executing the same code and every write or `undo` modifies `idx`. By Lemma 3, states cannot repeat. Therefore, if an attempt at the ith write fails due to a conflict with another process executing the same transaction, it must have encountered a "future" state. The ith state must have already succeeded in some other process. So, when the local transaction has executed $2w$

[1] Strictly speaking, given the strict sequencing imposed by the exclusive access one can choose *any* point from the time the log is successfully stored into the domain until this commit point as the point of linearization. However, that is only true for the algorithms that use helping ("roll-forward") and don't support multi-objects. For algorithms that use multi-objects and/or rollback, this is the appropriate point of linearization.

steps, either they all succeeded and the transaction has completed, or else some other process has completed the transaction for us. □

Lemma 12 *Each iteration in* `trans_open` *either installs log as owner and returns to the caller, or else helps complete another transaction (which is already the owner of this domain).*

Lemma 13 `trans_begin` *can only be called at most once for each transaction by each process.*

Proof: `trans_begin` is only called by `trans_open`. Consider process P. P executes a call to `trans_begin` with an argument of `log = *domain`. There are only two possible ways to return from `trans_begin`.

`trans_begin` returns `FALSE` only if `*domain` is no longer pointing to `log`, which by Lemma 4 can only occur if the `log`'s transaction completed. Once a transaction has completed, its log is never again stored in any domain.

If `trans_begin` returns `TRUE`, then `trans_open` calls `trans_conclude`. The call to `trans_conclude` ensures that the transaction corresponding to `log` completes before process P gets a chance to call `trans_begin` again.□

Lemma 14 `trans_conclude` *can only be called at most once for each log by each process.*

Proof: `trans_conclude` is only called by `trans_open` if and only if `trans_begin` returned `TRUE` (and incremented the reference count on `log`). By Lemma 13, `trans_begin` is called at most once for each log by each process. □

Lemma 15 *The loop in* `trans_begin` *(and hence the increment in the* `while`*) can only be executed at most* $2p$ *times for a given transaction.*

Proof: The loop repeats only if the `DCAS` fails. The `DCAS` fails only if `log` has been removed from `*laddr` (in which case `trans_begin` returns FALSE and a transaction has completed) or else some other process incremented or decremented `*cnt` between reading the arguments to `DCAS` and executing the atomic instruction. Increments of `*cnt` occur only in successful `trans_begin`s. Decrements only occur in the call to `atomic_decr` in `trans_conclude`. By Lemmas 13 and 14 `trans_begin` and `trans_conclude` are each called once per process per transaction. There are p active processes contending for a domain, so the `DCAS` in `trans_begin` can fail due to contention at most twice per process per transaction (and at most once due to transaction termination (`*laddr` not pointing to `log`)).□

Lemma 16 `atomic_decr` *can only loop at most* $p+1$ *times.*

Proof: The simple implementation of `atomic_decr` is to read the value, x, and attempt to atomically `CAS` in $x-1$. Each time x changes between the read and the `CAS`, the `CAS` fails. By Lemma 13 the reference count in `log` is incremented at most once by each actively contending process. The reference count starts at 1 upon allocation by the original transaction, so the maximum value it can reach is $p+1$. □

Lemma 17 *Given a set of operations each of whose sequential specification always executes in a finite number of steps. Assume each operation is implemented as a helper function with* w *writes to shared memory. Then, each process executes a finite number (at most* $O(p+w)$*) of steps before* some *process completes a transaction (makes progress).*

Proof: By the preceding lemmas, each procedure in the protocol either completes in a finite number of steps or guarantees that another transaction has completed in a finite number of steps. (There are a finite set of procedures listed in Chapter 3).

Management of the reference count can take $O(p)$ steps. The body of the helper function (including cleanup) must complete in $O(w)$ steps. □

Theorem 18 `CASn` *in Figure 3.3 is a non-blocking implementation of atomic* `CAS`n*.*

Proof: In this simple version, with a single global domain, atomicity and correctness follow directly from the fact that ownership of the domain (by a transaction, not a process) is exclusive. Exclusive ownership guarantees that no `CAS`n reads any intermediate state, and imposes a strict serial order on all `CAS`ns. The guarantee of progress follows immediately from Lemma 17. □

C.2 `CAS`n is $O(n)$

The general strategy I use to show that `CAS`n is $O(n)$ is to show that the transaction goes through at most $O(n)$ state changes. The first steps toward this were shown in the lemmas leading up to Theorem 18. In the preliminary version of the algorithm the cost of managing the reference counts on the log was $O(p)$. I show how the modifications introduced to deal with contention reduction reduce that cost to $O(1)$.

The $O(n)$ state changes may take place in multiple processes, and individual processes may be delayed arbitrarily long. However, the delay between each individual state change is $O(1)$.

Lemma 19 `log->ID` *can only change if the current owner is preempted by the scheduler or interrupted.*

The owner must have been running at the time it acquired ownership of `log->ID`. `log->ID` is only modified by the `DCAS` at (3.6:5). That code is protected by (3.6:3), which tests whether `log->ID` is currently executing. This guarantees that the owner must not have been running when the waiter executed (3.6:3) (although it mayhave since resumed).□

The `DCAS` at (3.6:5) ensures that the current process takes over ownership of the log `owner`, *only if* `owner` still owns `domain`. Once we acquire ownership of the log, we call `trans_begin`, which only succeeds in incrementing the reference count if the log we own, still owns the domain.

Lemma 20 `owner` *is valid inside the body of the* `while` *loop in* `trans_open`*.*

If we reached statement (3.6:1), then `*domain` was not NULL. If it had been NULL, then the call to `trans_start(domain, log)` would have succeeded, and we would have returned from the `while` loop before (3.6:1).

We store the current value of the owner of `domain` in the local variable `owner`. We test to make sure owner didn't complete its transaction between `trans_start` and the assignment. If it did, and `*domain == NULL`, then we need to retry the loop in order to execute trans_start. If `owner` is non-null then it points to a log that was the owner of `domain`. All subsequent references through `owner` come after the call to `trans_begin` which only returns successfully if it atomically incremented the reference count while owner *still* owned domain. Thus, if we pass `trans_begin`, `owner` is not reallocated.□

Lemma 21 `trans_begin` *returns FALSE iff the current process had been preempted.*

The `DCAS` at (3.7:1) fails only if another process has installed their ID in log. By Lemma 19 this is only possible if the current process had been preempted at some point. The statement immediately preceding the call to `trans_begin` checks to make sure that `myId` is installed in `Log->ID`.□

Lemma 22 *The loop in* `trans_begin` *executes at* most *once per call, and is called at most once per preemption*

There are two options. If `log->ID` still points to `myId`, then `cnt` could not have been modified, and the `DCAS` must succeed. Alternatively, if `myId` is changed, then (3.7:1) returns immediately.

`trans_begin` is only called if `ID` is successfully changed to `myID` by the new owner, which by Lemma 19 happens at most once per preemption. □

Lemma 23 *The process currently owning log can exit* `trans_conclude` *in* $O(1)$ *steps. At least one other waiter also exits* `trans_conclude` *in 1 extra step and is able to acquire ownership of the domain for a new transaction.*

If the transaction was preempted locally n times, then up to n waiters may be looping in `trans_conclude` waiting to decrement the reference count. Only one process whose process ID is stored in `Log->ID` can succeed at the `DCAS` on line 2. Once it succeeds, `log->ID` is null. Because process ID's are unique, no other process can contend with the decrement, and the owner exits `trans_conclude` in constant time.

n waiters now contend for decrementing the reference count. Although, one waiter will only succeed after $O(n)$ steps, one waiter will also succeed in O(1). □

The body of `CASnInternal` can take $O(n)$ steps to execute. The transaction can only be preempted $O(n)$ times, each with a call to `trans_begin`. By Lemma 22, each call to `trans_begin` loops only once. By Lemma 23 the current owner can exit `trans_conclude` in $O(1)$. Thus we have:

Theorem 24 `CAS`n *executes in worst–case time* $O(n)$.

C.3 Multi-object updates are non-blocking and livelock free

Section 3.2.3 introduced enhanced protocols that supported multi-object domains for both STM with roll-back and roll-forward. These enhancements affect the reasoning behind the proofs that the simpler algorithms were non-blocking.

Can two or more transactions deadlock? No, because transactions only wait for running processes. No running process is forced to wait for a waiting (non-running) process, and therefore cyclical dependencies (and, thus, deadlock) are impossible.

For STM with roll-back, we can ask whether two or more transactions can livelock? (By livelock we mean that two or more transactions persistently abort each other so no progress is made). Livelock is unlikely, given exponential backoff, but this assurance is not sufficient to prove that the algorithm is non-blocking (static transactions (such as `CAS`n even with no helping) can be made livelock free by simply sorting the arguments in ascending address order. However, this is not a general solution, and clearly does not work for dynamic STM).

To deterministically avoid livelock, we introduce the explicit notion of precedence. In the single domain case we implicitly gave precedence to running processes over blocked processes. We avoided livelock and ensured progress (in the absence of infinite preemptions) by guaranteeing that once a transaction acquired ownership of a domain, it was guaranteed to complete without being aborted or preempted. In the multi-domain case we can make no such guarantee with the original approach. Consider two transactions, TX_1 and TX_2, which access domains (A, B) and (B, A) respectively. If TX_1 proceeds to the point it owns A and TX_2 passes the point at which it owns B, one (at least) must be aborted before the other can proceed.

By imposing an order on transactions, (and establishing that a maximal precedence transaction exists), we can guarantee that the highest precedence successfully running transaction is not aborted. The fixed order guarantees that there are no cycles — if transaction TX_1 aborts TX_2, we guarantee that TX_2 will not later abort TX_1. The existence of a maximal precedence ensures progress — there is always some transaction that no one will abort. By giving running processes precedence over waiting processes we ensure that processes experiencing long delays do not cause other processes to wait. (The waiting is still cycle-free (and thus deadlock free) because running processes never wait for stopped processes, regardless of precedence).

Note, however, that with roll-back and without helping, an adversary can always force livelock by perpetually repeating the sequence of steps that caused the initial rollback. However, the adversary must accomplish five steps to force livelock: It must (a) force TX_1 to relinquish the processor after *every* time it acquires A, (b) further force TX_1 to remain descheduled until TX_2 is ready to start, (c) it must schedule TX_2 while TX_1 is descheduled, (d) it must also interrupt TX_2 after every time *it* acquires ownership of B, and (e) it must resume TX_1 at that point. Under the assumption that infinite process descheduling is not allowed (no algorithm can make progress under such conditions), then the adversary cannot do either (a) or (d), and so cannot force livelock. Even in the case where we allow infinite preemption, an adversary cannot delay a runnable process if no other processes are runnable on the machine, so by increasing the delay of TX_2 between attempts, we can reduce the probability to 0 that the adversary can wait long enough in (b) to cause livelock. The policy described above ensures that as long as there is *one* run in which TX_1 is not preempted while TX_2 is runnable, then livelock is impossible and progress is probabilistically guaranteed.

This policy of aborting on conflict, however, introduces some inefficiencies. Consider two transactions that only overlap on domain A (or any set of domains that they both access in the same order). If transaction TX_1, with lower precedence, acquires A first, then if TX_2, with higher precedence, attempts to acquire A before TX_1 has completed, then TX_1 would be needlessly aborted, wasting the work done so far. In truth, transactions only need to be aborted if a cyclical dependency should arise. If the dependencies are not cyclical, we should be able to serialize them in any order (precedence is introduced to avoid livelock, not to ensure that higher precedence transactions complete first!). TX_2 should be able to wait until TX_1 finishes, and then proceed.

To reduce the work wasted due to needless aborts, we defer assigning precedence until two transactions conflict for the first time. Initially precedence is 0 (unset). If either transaction has a precedence of 0 at the start of a conflict, it is impossible for them to have any cyclical dependencies. Therefore, if both transactions are currently running, and at least one of them has zero precedence, the second transaction to attempt to acquire ownership of a given data structure will defer and back-off rather than abort. This does not eliminate all needless aborts, but it catches the most common case with very little cost.

For STM with roll-forward, a different form of wasted work is introduced by so-called *redundant helping* [87, 74, 1]. Redundant helping is not a problem caused by aborting too frequently, but rather with not aborting aggressively enough.

The first problem introduced by redundant helping is reduced parallelism caused by inteference between disjoint accesses. If we follow the rule that a transaction T_i must help all transactions T_j with which it conflicts before proceeding on its own operations, then long transitive helping chains can be built up. Assume T_1 conflicts with T_2, which in turn conflicts with T_3, and assume T_1 and T_3 access disjoint locations. When T_1 detects its conflict with T_2 and tries to help T_2, it discovers the conflict with T_3 and tries to help T_3. This effectively serializes T_1 and T_3, because T_1 cannot proceed until T_3 has completely finished. Arbitrarily long chains can be built up. If a chain of C interefering transactions exist, breaking the chain by aborting at least one transaction every k links in the conflict graph will abort C/k transactions, but also allow C/k transactions to proceed in parallel.

Some approaches [87, 74] attack this problem by simply eliminating recursive helping when first detected, aborting the first transaction that tries to recursively employ helping. (This aborts a full half of the conflicting transactions in a chain of C transactions, but also allows the remaining $C/2$ transactions to proceed concurrently). Others [1] actually operate on the conflict graph, and schedule optimal helping policies. The contributions of [1] are orthogonal to the point of this thesis. While in this thesis I eliminate recursive helping without explicitly optimizing by consulting the conflict graph, I note that one can apply their techniques both to my STM with roll-forward to optimize the helping, and to STM with roll-back, to minimize the needless aborts. It is important to note that in my STM algorithm, where domains are data-structures, long chains are unlikely. And when they do occur, most likely real dependencies exist (thus breaking the chains would not increase parallelism by much). Therefore the cost of such optimizations are unlikely to result in useful benefits. I have not measured this tradeoff in any real systems.

Helping introduces a second problem: wasted work due to helping a transaction that is bound to fail. If a transaction reads old values, performs a computation, and then tries to update using CASn, a conflict probably means that the original CASn will fail (exactly how probable will depend on the granularity of the conflicts). A chain of n conflicting transactions might well result in $n-1$ failures. This problem does not arise *unconditionally* in the dynamic STM algorithms, because the reads only occur *after* acquiring ownership. It depends on the algorithm being emulated. The high probability of failure upon conflict does occur for CASn (because the transaction depends upon values read *before* the transaction began), but not for `Fetch&Incr` (where the transaction is commutative, nor would it occur for idempotent transactions). For CASn, where the problem does arise, eliminating recursive helping also reduces this inefficiency.

Note, however, that avoiding redundant helping by aborting means that even in STM with roll-forward some transactions roll-back and are forced to retry. Non-blocking algorithms that work by updating local copies (such as [74]) only abort due to conflict in the case that some other transaction has succeeded, and hence there is still a guarantee of progress, so the algorithms are provably non-blocking. In our algorithm, which updates in place, conflict is detected by partially completed transactions when they attempt to acquire ownership of a data structure that is owned by another partially completed transaction. We need to establish that if a roll-back occurs, we have a guarantee that *some* transaction has completed and made progress.

Precedence numbers come to our aid, here, too, as any time an abort occurs, some transaction with highest precedence number[2] will complete. (Note that if it is preempted, the completion is accomplished by helpers, not the original process.)

Once again, linearizability follows from the strict ordering imposed by the exclusive ownership of domains. Any two transactions that interact (access a shared domain) are strictly ordered with respect to each other. Further, the strict ordering applies between transactions that transitively interact because all domains are released after completion of the transaction.

[2]This is not a unique transaction, although the precedence number is unique. We cannot guarantee that two or more transactions will not have identical precedence numbers because the precedence is bounded, and precedence can wrap-around. However, the code presented here works correctly even in the case of wrap-around. Equal precedence transactions wait rather than abort. Two equal precedence transactions can actively interfere over only one domain at a time. Thus no cycles can be introduced between the equal-precedence transactions. As long as at least one transaction with maximal precedence exists, *one* of them is guaranteed to proceed.

Appendix D

Non-blocking Implementations of Data-Structures

D.1 Introduction

A programmer wishing to implement an efficient non-blocking algorithm for a given data structure must proceed by trial and error. There is no known recipe to achieve both non-blocking progress and efficiency. Herlihy [38] and others have shown that it is possible to implement *any* data-structure in a non-blocking (or wait-free) manner, but his construction [41] and other universal constructions, are often inefficient. Chapter 3 presents another general technique for converting *any* sequential algorithm into an equivalent non-blocking algorithm with reasonable asymptotic performance. Unfortunately, in the common case (small-scale and low contention) even these universal constructions do not produce efficient implementations compared to well-tuned blocking algorithms for a specific data-structure. *This* appendix addresses the question of how one can design efficient non-blocking implementations of particular data-structures. This appendix describes techniques a programmer can apply, and gives one or two concrete examples of implementations of data structures using each technique.

It is important to stress that this appendix does not claim to be "universal". The techniques presented here are not guaranteed to cover *every* data structure. They are also not mechanically applicable. A programmer must carefully choose the techniques that apply to his or her data structure.

Though the set of tricks and techniques presented here is not provably exhaustive, I hope it is representative enough to enable readers to convert most performance critical data structures that they might encounter in practice. There is anecdotal evidence to support my claim that this hope is realistic: at first, *each* new data structure I attempted to implement added a new technique to my list. Gradually, new additions to the list of techniques became rare. More recently every new data structure I have attempted to implement has been amenable to a non-blocking implementation using a technique I had already encountered.

By presenting techniques that are not "universal", I am implicitly relying on answers to the questions "what data structures are important?" and "what are the typical patterns of usage of a given data structure?" I think this is legitimate: it is good system design to optimize for the common case, and one should be willing to take advantage of features of that common case (e.g. single writer multiple reader, or low contention). To the extent that system designers have similar answers to these two questions, the techniques in this chapter will be useful. I believe that there *is* a rough consensus on important data structures and typical patterns of usage in the case of performance critical shared data structures.

It has been our experience that, where performance is relevant, contention rates on data structure are low. This is sometimes fortuitous [98, 67] and sometimes by design [25].

In either case, the low contention arises since memory contention is usually significantly more problematic than synchronization contention. That is, even assuming synchronization is free, and unlimited concurrency is algorithmically possible on a shared data structure, the cost of faulting the data from processor to processor can swamp other costs (even the cost of direct communication must be amortized over the time of the computation). Therefore, it is preferable to (re)structure data structures and algorithms to reduce memory contention (and communication costs) and partition the data so that each processor typically caches a portion of the data and works locally on it[25]. Once data is partitioned in this manner, synchronization becomes a *mostly* local issue.

Therefore it is reasonable to assume that contention on shared data structures is usually low and optimize for that case. (We require graceful degradation when our assumption is wrong).

What kind of data–structures are shared? It seems questionable from a reliability standpoint to have threads from separate address spaces (particularly on separate processors) sharing access to complex data structures. These data structures are also more difficult to program and to maintain and often provide marginal performance benefits in practice, particularly when synchronization overhead is taken into account. Their asymptotic performance benefits are often not realized at the scale of typical operating system data structures. (Based on first principles, this should also hold true for all applications distributed across multiple processors in general, although our direct experience supporting this assertion is more limited.)

Thus, *where performance is an issue*, shared data structures will commonly be simple data structures with low contention. (This is not to say that there is no place for complex data structures, just that in well-designed systems they will usually not be shared between separate processors in critical cases.)

In a real sense, then, the performance gains achieved by using `DCAS` (rather than `CAS` or locks) directly (rather than through universal transformations) in simple data structures are at least as compelling advantages of `DCAS` as the asymptotic results presented in Chapter 3 — because simple data structures are most likely to be used in performance critical areas.

D.1.1 Approaches

There are two general approaches to one can use to implement a data-structure-specific non-blocking update.

1. *Detect conflicts and retry*: Locally record an initial state of the data structure. Compute an updated state based on this initial state. Then, atomically test that the data structure is unchanged and, if so, install the new state. If the state *has* been changed, begin again.

   Note that if the test and installation are truly atomic, then the state can have changed between the read and the test/installation of the new state only in the case that some other operation updated the state successfully. This condition is sufficient to imply that this approach is strictly non-blocking because the only way that one process can delay another process is by making progress. (Strictly speaking, we also depend upon the fact that the computation of a new state occurs in finite time.)

   If the test and installation are not atomic, then there exists some possibility of interference between two partial updates. The result of the interference might be that neither update is successful. In such cases, the best we can aim for is to be non-blocking with probability 1, or to be effectively non-blocking as described in Chapter 1.

2. *Cooperate*: Record your intended operation in shared memory and have other processes *help* you complete if you are delayed. Before modifying a data structure, check if any operation is pending. If so, first complete the pending operation. If there is no pending operation then atomically record your intended operation. Finally, begin your update, checking to make sure no one else has completed it for you.

   If there is a non-blocking implementation of "recording intended operations" (there is), and if execution of each operation takes finite time (it does, by assumption), then this approach is non-blocking.

Section B.3 points out some drawbacks with fully general helping mechanisms. For some particular data structures these drawbacks do not apply.

A small number of basic techniques and tricks can help realize these approaches for specific data structures. Each of the following techniques is described in its own section.

- Direct implementation.
- Single Server
- Layers of read-mostly hierarchy
- Relaxing Consistency Requirements
- Copying small sub-objects
- Explicit and Implicit Helping
- Adding bits to invalidate states
- Consistency and hints

Appendix A addresses the misconceptions that NBS increases concurrency and that increased concurrency is necessary or sufficient to implement NBS.

Increasing concurrency reduces, but does not eliminate, conflicts between different transactions attempting to modify a single object. One still must make sure that the algorithm is non-blocking in the case of updates that still *do* conflict.

Conversely, it is also important to understand that non-blocking algorithms do not necessarily increase concurrency. non-blocking updates may still impose a strict total-order on the updates applied to a data structure, and these might execute serially resulting in no obvious throughput gain.

While the techniques described in this chapter occasionally increase concurrency for a given data-structure, I am mostly interested in the non-blocking property itself. For the purpose of this presentation, increased concurrency is incidental, and is rarely discussed.

```
Push(*stack, *entry)
{
  pointer_t old_top;

  do {
    old_top = stack->top;
    entry->next.ptr = old_top.ptr;
  }while(!(CAS(&(stack->top),
               old_top,
               <entry,
                old_top.count>)));
}
```

```
entry *Pop(Stack *stack)
{
  pointer_t old_top;
  entry *top;

  do {
    old_top = stack->top;
    top = old_top.ptr;

    if (top == NULL) {
      return (FALSE);
    }
  }while(!(CAS(&(stack->top),
               old_top,
               <top->next.ptr,
                old_top.count+1>)));
  return(top);
}
```

Figure D.1: Code implementing Treiber's non-blocking stacks

D.2 Common cases

Given atomic `DCAS`, we know that any atomic update that only modifies two words of memory can be implemented directly. In the trivial cases where the entire data structure can be explicitly implemented in one or two words (say, a double-width counter) the implementation is straightforward and uninteresting. However, the state of many data structures of arbitrary size are fully determined by only two words at any given time.

D.2.1 List-based stacks

Consider a stack supporting two operations, `push`, and `pop`. `push` adds a new element to the *top* of the stack. `pop` removes the top element of the stack and returns it to the caller. Clearly the total state of the stack is larger than two words — it is proportional to the number of elements in the stack. A non-blocking `push` can still be implemented easily. Suppose we want to push `entry` onto the stack. On each attempt, we can set `entry->next` to point to `top = stack->top` with no synchronization (since `entry` is purely local to our process until it is successfully stored on the stack). Next, we can attempt to atomically update the top of the stack to point at `entry` using `CAS`. If the attempt to atomically change `stack->top` from `top` to `entry` succeeds, then the stack remains in a consistent state. If the attempt fails, then someone else must have successfully altered the top of the stack, and we retry.

pop is slightly more problematic. We can no longer simply look at stack->top. Consider a stack whose top three elements are A, B, and C, in order. Process P begins a pop operation which reads the contents of stack->top (A) into top, then reads top->next (B) into next. At this point some other process may pop both A and B off the stack, and then push A back on. To complete the pop operation, P wants to store next (pointing to B) into stack->top. P cannot simply check that stack->top still points to *top* (which still contains A) — in our example A is the top element but the next element is C! P must ensure that it has correct values for *both* the top two values. For CAS-only implementations this can be achieved by either adding version numbers to the stack->top pointer (e.g. Treiber [84]), or guaranteeing that A will never be reused (e.g. atomic-pop on the Lisp Machine [97]).

Treiber's non-blocking algorithm for LIFO stacks (pseudo-code in Figure D.1) illustrates how this problem is solved in a CAS-only system. The stack top pointer consists of a version number and a pointer to the top-of-stack. On machines with a double-width CAS the representation is straightforward — a word for the version number and a word for the pointer. On machines that only operate atomically on smaller width words, the pointer can be represented as an index into an array, so that both the "pointer" and version number can fit into a single word. The version number is only incremented on pop to reduce the rate of wrap-around (this effectively gives us 1 extra bit, doubling the time between wrap-arounds).

Using DCAS, however, pop can be implemented as simply as push. We record newTop = top->next at the start of an attempt. If stack->top is top and top->next is newTop, then changing stack->top to point to newTop atomically changes the stack from one consistent state to another. Once again, we can only fail if some other process successfully pushed or popped the stack between the initial read of both stack->top and top->next and the execution of DCAS. The entire state of this stack at any *given* time is maintained in these two locations. One of the two locations, though, varies over time.

The non-blocking code implementing list-based stacks is shown in Figures D.2 and D.3.

```
do {
  backoffIfNeeded();
  top = stack->top;
  entry->next = top;
} while      /* Retry if changed */
  (!CAS(&(stack->top), top, entry));
```

Figure D.2: Push on stack (list).

```
do {
  backoffIfNeeded();
  top = stack->top;
  if (top == NULL)
    { return(FALSE); }
  newTop = top->next;
} while
  (!DCAS(&(stack->top), &(top->next),
         top,             newTop,
         newTop,          NULL));
```

Figure D.3: Pop from stack (list).

`backoffIfNeeded` hides the details of an exponential backoff algorithm to improve performance of this algorithm under contention. An exponentially growing backoff interval is stored (either in the shared data structure or in a local variable). A number is chosen, uniformly, from within that interval, and the process waits that amount of time before doubling the interval and retrying.

Note that the approach of minimizing logical contention, outlined in Chapter 4, can sometimes eliminate the need for any backoff algorithm. If we know that the maximum number of processes contending on a given data structure is bounded, and we know the minimum time before a process that successfully performed an update could possibly retry, it is sometimes possible to demonstrate that backoff is unnecessary. In practice this is fairly common inside an OS kernel, where a successful update might mean that the given processor will not access that data structure again until after, at least, a context switch.

D.2.2 Array-based stacks

An array-based stack can be more efficient[1] than a list-based one – it avoids allocation and deallocation for each operation, and avoids the space overhead for list pointers in each entry. Further, this same approach can be generalized to support an implementation of a *set* of arbitrary elements (where the potential size of the set is either large or not known in advance, so that a bit-vector

[1] In practice, care must be taken in deciding on the representation of a stack. Allocation and deallocation can be done out of per-processor free-lists, so no synchronization is needed and we can expect few cache misses for heavily used stacks. Also, array-based stacks require *two* writes (the stack-pointer and the top-element of the stack), invalidating two cache lines. In contrast, list-based stacks only write `stack->top`, while the second argument (`top->next`) is only read. Further CMDS (Section 4.3), would suggest padding out `stack-entry`'s to be full cache-lines. If we pad each location in a stack-based array we lose the space advantage over list-based stacks. If we do not pad, then we increase memory contention.

approach is infeasible).

First consider the array-based stack itself. If we implement a stack as an array, then although the stack may grow and shrink, any operation on the stack still affects only two words: the stack-pointer and one value. Figures D.4 and D.5 implement `push` and `pop`, respectively, for a non-blocking array-based stack implementation.

```
do {
  backoffIfNeeded();
  index = stack->sp;
  if (index >= stack->size)
    { /* signal stack_overflow */ }
  oldVal = stack->data[index];
} while
  (!DCAS(&(stack->sp),
 &(stack->data[index]),
         index,        oldVal,
         index+1,      val));
```

Figure D.4: Push on stack (array).

```
  do {
    backoffIfNeeded();
    index = stack->sp-1;
    if (index == -1) {
      /* signal stack underflow */
    } else {
      returnValue = stack->data[index];
    }
  } while
  (!DCAS(&(stack->sp), &(stack->data[index]),
         index+1,      returnValue,
         index,        (T)NULL));
```

Figure D.5: Pop from stack (array).

The implementation is straightforward. (As discussed in Chapter 2, Synthesis [64] tried to be clever, and used `CAS` rather than `DCAS` in `pop`. Their cleverness introduced a race condition.) Section D.9.2 considers extending our implementation to an array-based `set`, where we support constant time deletion of arbitrary elements.

D.2.3 FIFO queues

In both array-based and list-based implementations of stack, the 2-location update always modifies one fixed location (`stack->sp` and `stack->top` respectively). The second argument (`stack->data[index]`) in the array-based implementation varies. (Although the second argument (`top->next`) to `DCAS` in the `pop` operation of the list-based implementation varies, too, it is essentially just read, not written.) Despite this variation in the second argument, the operations are strictly linearized based on updates to the one fixed location.

The following `DCAS`-based non-blocking implementation of FIFO queues (see Figures D.6 and D.7) varies *both* first and second word of state. It is loosely derived from the `CAS`-only implementation of Michael an Scott[70] (see Section D.5.1). This example demonstrates how the degree of concurrency in a non-blocking implementation matches the degree of concurrency in the equivalent blocking implementation. The Michael and Scott implementation allows a concurrent enqueue and dequeue to execute in parallel as long as the queue is not a singleton (when the head and tail are the same object). They provide a `CAS`-only non-blocking implementation and a lock-based blocking implementation.

The `DCAS` implementation presented here is noticeably simpler than the `CAS`-only implementation. It eliminates the complexity of managing both head and tail pointers when dequeuing a singleton node, thereby avoiding the need for a dummy node. Further, because the queue always transitions from consistent state to consistent state there is no need to include counters in each pointer. Finally, in the common case of enqueuing on the tail of a non-empty queue, the `DCAS` algorithm does not have to deal with updating the `tail` pointer *after* the update of the `next` pointer — both are updated atomically and so can never get out of synch.

D.2.4 Priority queues: linked lists

It is not the case that *any* data-structure that only modifies two words per operation is trivially implementable using `DCAS`. Consider the case of inserting an element into a singly linked list. In such a case the update consists of modifying a single word (the `next` pointer of the previous element). However, that modification is only correct if the *entire* chain of `next` pointers from the head of the list still points to `prev`. If we are inserting the element into the nth position in the list, then the state depends on n `next` pointers.

An example that illustrates this problem is an attempt to insert `entry` into a list L. Process P finds that `entry` belongs after p and before q, and then goes blocked. While P is blocked other

```
entry->next = NULL;
do {
  backoffIfNeeded();
  tail = q->tail;
} while      /* Retry if tail changed */
  ((tail == NULL)?
   (!DCAS(&q->tail,      &(q->head),
          NULL,          NULL,
          entry,         entry)):
   (!DCAS(&(tail->next), &(q->tail),
          NULL,          tail,
          entry,         entry)));
```

Figure D.6: Enqueue on FIFO Queue.

```
do {
  backoffIfNeeded();
  head = q->head;

  if (head == NULL) {
    /* Queue empty */
    return(FALSE);
  } else
    { newHead = head->next; }
} while
  ((newHead == NULL)?
   (!DCAS(&(q->head), &(q->tail),
          head,       head,
          NULL,       NULL)):
   (!DCAS(&(q->head), &(head->next),
          head,       newHead,
          newHead,    NULL)))
return(head);
```

Figure D.7: Dequeue from FIFO Queue.

processes first delete q and then p and insert them both on another list L' (the free list would be a particularly bad case of L'!). When P resumes, p is still pointing at q, although both are now in L' rather than in L. The insertion, however, would succeed, and `entry` would now be inserted into L' rather than L.

Fortunately, in any case where the state depends upon an arbitrary number of reads and only one write, we can conventionally protect the entire data structure with a version number. If the version number is incremented on every modification, we can use it to guarantee that the state of the data-structure is unchanged from the beginning of an update until the time we wish to atomically perform the write. In the previous example the insertion would fail despite the fact that p is still pointing to q, since the version number of L would have been changed.

Figures D.8 and D.9 display the implementation of a non-blocking priority queue as a linked list. The `DCAS` simultaneously updates the `next` pointer and increments the version number.

Note that the list is initialized with a dummy entry (i.e. the "empty" list consists of a single entry). Nothing ever looks at the contents of the dummy entry, but the code is slightly simpler this way, since we do not need to special case insertion or deletion at the head of the list.

```
do {
  backoffIfNeeded();
  version = list->version;

  for (p=list->head;
        (((next = p->next) != NULL) &&
         (next->datum < datum));
        p=next) {}
  entry->next = next;
} while  /* retry if list changed */
  (!DCAS(&(list->version), &(p->next),
         version,          next,
         version+1,        entry));
```

Figure D.8: Insertion into middle of a singly-linked list. Note that list is initialized with a single, "dummy" entry.

Given that word sizes are bounded, the correctness of this algorithm depends upon the version number not wrapping around to have the same value during the course of a single update. The probability of this happening must be assessed on a per-instance basis. If the probability of wrap-around (or the cost of incorrect operation) is too high, then several alternatives to version numbers

```
do {
  backoffIfNeeded();
  version = list->version;

  for(p=list->head;(p->next!=elt);p=p->next){
    if (p == NULL) {  /* Not found */
      if (version != list->version)
        { continue; }    /* Changed, retry */
      return (NULL);  /* Really not found */
    }
  }
} while(version != list->version ||
        !DCAS(&(list->version), &(p->next),
              version,          elt,
              version+1,        elt->next))
```

Figure D.9: Deletion from the middle of a singly-linked list.

are possible for a modest cost.

Elsewhere in the literature (e.g. [11]) people have proposed using the "bounded tags" approach developed by Moir[73] to deal with the problem of version-number wrap-around. Bounded tag algorithms guarantee that *p* objects can be assigned unique IDs from a fixed size set of names.

I propose another alternative, described more fully in Chapter 3 (where I use it in my universal constructions), to use a unique ID rather than a version field. The unique ID must have the property that it cannot be re-used during the lifetime of any transaction that has observed it. In other words, if a process, P_1, reads a particular unique ID at the beginning of an update and the unique ID field still points to the same unique ID at the end of the update, then it is *guaranteed* that no transaction has made any modification to the data structure during that time, regardless of how long P_1 waits.

This is trivial to implement by using a pointer to a shared object in memory as the unique ID. If `list->uniqueId` is NULL at the start of an update, the process atomically installs a newly allocated unique ID with a reference count of one into `list->uniqueId`. If `list->uniqueId` is not NULL, then the code must atomically increment the reference count of the unique ID while simultaneously checking that *this* unique ID is still installed in `list->uniqueId`.

When we finish with an attempted update, instead of incrementing the version number as part of the final `DCAS`, we try to set `list->uniqueId` from the unique ID we read originally to NULL. After the `DCAS`, whether or not the `DCAS` succeeded, we decrement the reference count. The `DCAS` can only fail if some other process has already set `list->uniqueId` to `NULL` once. The object is only deallocated and eligible for re-use when the reference count goes to zero. Note that only one process can possibly see a reference count of zero: the reference count monotonically increases until `list->uniqueId` goes to NULL, at which point it monotonically decreases.

D.2.5 Double-ended queues: deques

The examples of direct implementation just given above dealt with data structures whose state naturally fit into two words of memory, and hence were easily handled by `DCAS`. What of data-structures where the state requires more than 2 words? Sometimes it is possible to encode 2 or more independent state variables into a single machine word.

A simple example of this technique is simply compressing full-word pointers into half-word array indices. Consider the implementation of a non-blocking *double-ended queue*, or a *deque*.

A deque is a queue that allows enqueues and dequeues at both ends. That is, it supports four operations that modify its state: pushTop, pushBottom, popTop, and popBottom. Deques have become important data-structures in the implementation of work-stealing [20] schedulers.

Work-stealing schedulers maintain a per-processor queue (deque) of runnable threads. If a processor empties its list of runnable threads it becomes a "thief", and steals a thread from the top of a "victim" double-ended queue. Local processors treat the bottom of the deque as a stack of runnable threads. (The work-stealing algorithm has been refined by [15] to match "thieves" with "victims" to efficiently support distributing computation across a wide area network.) Work-stealing scheduling has many nice theoretical and practical properties and is deployed in a number of systems (e.g. [31, 19, 35, 29, 91]).

The deque is central to an implementation of work-stealing scheduling. Recently, Arora, Blumofe, and Paxton [11] have proposed a non-blocking implementation of deques using only `CAS`. Their implementation depends on the fact that only a single process operates on the bottom of any given deque (so calls to `pushBottom` and to `popBottom` will not execute concurrently), their implementation does not support a `pushTop` operation, and their implementation fails if a counter wraps around between invocations. My `DCAS`-based implementation of deques supports all 4 functions, allows multiple processes to access both the top and bottom of the deque, and does not depend on probabilistic correctness.

A deque consists of a header with two pointers, `top` and `bottom`, which point into a contiguous area of memory. If the area of memory is a bounded array, we can use array indices rather than pointers and fit both top and bottom into a single word. Figures D.10 and D.11 implement a non--blocking deque using this trick.

```
void pushTop(deque *dq, void *newval)
{
 if ((dq->ptrs).top == 0) { ERROR(); }
 do {
   ptr = dq->ptrs;
   if ((ptr.bot <= ptr.top) &&
       CAS(&(dq->ptrs, ptr, <0,0>))) {
     ptr = <0,0>;
   }
 } while (!DCAS(&(dq->ptrs),          &(dq->entries[ptr.top-1]),
                <ptr.bot,ptr.top>,    NULL,
                <ptr.bot,ptr.top-1>, newval));
}

void *popTop(deque *dq)
{
  do {
    ptr = dq->ptrs;
    if (ptr.bot <= ptr.top) {
      if (CAS(&(dq->ptrs, ptr, <0,0>)))
        { ptr = <0,0>; }
      return((void *)NULL);
    } else {
      val = dq->entries[ptr.top];
    }
  } while (!DCAS(&(dq->ptrs),          &(dq->entries[ptr.top]),
                 <ptr.bot,ptr.top>,    val,
                 <ptr.bot,ptr.top+1>, NULL));
  return(val);
}
```

Figure D.10: Insertion and deletion at the top of the deque

Ideally, operations at the top and bottom of a deque should be able to execute concurrently. Both my implementation and the published implementation [11] preclude this by making `bot` and `top` share a single word. We will revisit the deque example with an alternate, concurrent, non-blocking implementation when we illustrate the technique of making intermediate states "consistent" in section D.5.1.

```
void pushBottom(deque *dq, void *newval)
{
 do {
   ptr = dq->ptrs;
   if ((ptr.bot <= ptr.top) &&
       CAS(&(dq->ptrs, ptr, <0,0>))) {
     ptr = <0,0>;
   }
 } while (!DCAS(&(dq->ptrs),          &(dq->entries[ptr.bot]),
                <ptr.bot,ptr.top>,    NULL,
                <ptr.bot+1,ptr.top>, newval));
}

void *popBottom(deque *dq)
{
  do {
    ptr = dq->ptrs;
    if (ptr.bot <= ptr.top) {
      if (CAS(&(dq->ptrs, ptr, <0,0>)))
        { ptr = <0,0>; }
      return((void *)NULL);
    } else {
      val = dq->entries[ptr.bot-1];
    }
  } while (!DCAS(&(dq->ptrs),          &(dq->entries[ptr.bot-1]),
                 <ptr.bot,ptr.top>,    val,
                 <ptr.bot-1,ptr.top>, NULL));
  return(val);
}
```

Figure D.11: Insertion and deletion from the bottom of a deque

D.3 Single server model

It is common to discover that a particular data–structure has a single consumer/reader and multiple producers/writers (or vice versa). In these cases we can exploit the lack of concurrency to avoid synchronization — the single consumer or writer can serialize tricky operations. The serialization already occurs naturally (that is, the lack of concurrency is not the result of converting to a non--blocking implementation). We are not paying any penalty for this serialization.

D.3.1 Examples from other work

Synthesis [64] proposed using servers to serialize *all* operations for data-structures that were difficult to synchronize. This, unfortunately, introduces the single server as a serial bottleneck where none existed before. (Fortunately, they never needed to resort to this technique except for preliminary prototype implementations).

The interesting cases, though, are when one can exploit *naturally* occurring lack of concurrency.

In an earlier Synthesis paper [63], Massalin describes several array-based circular FIFO queues, implemented using `DCAS`. Versions are optimized for the cases of single producer and single consumer, single producer and multiple consumers, and multiple producers and single consumers. The fact that one path through the code always went through a single process simplified the synchronization needed, and improved performance. Matching the appropriate implementation to each application benefited their thread and I/O systems.

The `CAS`-only implementation of deques in [11] exploits the fact that, in the work-stealing algorithm, only one process ever manipulates the bottom of a deque. Thus manipulations of the `bottom` pointer need not be synchronized, `popTops` must be synchronized only against each other, and the only complicated synchronization involves making sure that `bot` and `top` do not pass each other. This observation significantly simplifies the code in [11] (although at the cost of more limited functionality compared to fully general deques).

Chesson [28] designed a stack for a non–blocking shared-memory message-passing library, which SGI ships as a product. It exploited multiple-writers/single-readers to avoid the version number used by Treiber [84].

D.3.2 Single server heap

It is often the case with a heap (or other implementations of priority queues), that one only actually `heap-delete-tops` (removes the top item) one item at a time (exhibiting no parallelism) but heap-insert might get called at any time by any number of processes (concurrent with at most one delete-top and many other inserts). In such a case heap-delete-top can act as a single server, and serialize all requests. heap-insert can "lie" and simply atomically enqueue the new item on a "pending insert" list. heap-delete-top first real-heap-inserts everything on the pending list sequentially, and only then does a real-heap-delete-top. ("There is no need for concurrency control if there is no concurrency.")

There *are* concurrent non-blocking implementations of heaps (e.g. [46]), but if the natural structure of the system is limited to a single deleter, there is nothing to be gained by parallelizing.

D.3.3 A FIFO queue with a single dequeuer

Imagine a FIFO queue with many enqueuers and a single dequeuer. (Such a data structure might used for the output queue for a single port on a (hypothetical) network switch.) We can exploit the fact that there is a single dequeuer to eliminate almost all synchronization on the dequeue operation. The only case we must protect against is when there is a single entry in the queue and we simultaneously try to enqueue and dequeue. Enqueue is unchanged from Figure D.6, while the new version of dequeue is presented in Figure D.12.

```
head = q->head;
if (head == NULL) {
  /* Queue empty */
  return(FALSE);
} else {
  newHead = head->next;
  if (newHead == NULL) {
   if (DCAS(&(q->head), &(q->tail),
            head,       head,
            NULL,       NULL)) {
    return(head);
    }
  } else {
    newhead = head->next;
  }
  q->head = newhead;
  head->next = NULL;
  return(head);
}
```

Figure D.12: Dequeue code for FIFO Queue with single dequeuer.

Since there's only one dequeuer, and all enqueues are done at the tail of the queue, head

`= q->head` cannot change during a dequeue (unless it starts as `NULL`). Similarly, `newhead = head->next` will not change once it is no longer `NULL`. If `head` starts out as `NULL` then we can legitimately return `NULL` with no synchronization since the dequeue operation can be serialized before any of the enqueues. If `newhead` is not `NULL`, then no enqueuer will interfere with the first two elements on the queue, and the dequeue can be done with no synchronization. If `head` is not `NULL` and `newhead` is `NULL`, then there is only one element in the queue. A dequeue must update both `q->head` and `q->tail`, and an enqueue must set the `next` pointer of the element being dequeued — synchronization is needed. `dequeue` tries to `DCAS` both `q->head` and `q->tail` from `head` to `NULL`. If it succeeds, the dequeue is complete and can return. If it fails, then some enqueue has succeeded and the queue is now in a state where enqueue does not conflict with dequeue, so we can proceed one more time with no synchronization. This time we are guaranteed to succeed.

D.4 Reducing contention by adding read-only hierarchy

A common reason to increase complexity by adding a layer of hierarchy or indirection to a data structure is to reduce contention. For example, as noted in Chapter 4, a large list that is searched frequently may be revised to be a hash table. Then, searches and updates are localized to a portion of the list (a single bucket), reducing the conflict with other operations, assuming they hash to different buckets.

The key feature that can be exploited to simplify non-blocking implementations is the fact that the upper levels of the hierarchy are read-only or read-mostly: descriptors are only added at the leaves. No synchronization is needed for access to the hash table – no conflicts are possible unless two processes access the same bucket.

Operations in the hash table are implemented by using the hash-key mod the number of buckets as an index to find the target bucket. The bucket is implemented as a list, protected by a version number. Insertion, deletion, and lookup are implemented simply as the corresponding operations on lists.

This implementation is complicated somewhat if operations on the entire hash table are desired (e.g. resizing the hash table). Assuming such operations are rare, two simple implementations are possible. The first approach enlists "helping". If the hash table needs to be resized, it is marked as dirty, and every subsequent operation first tries to help finish the resizing before performing its own operation. The second approach also marks the hash table as dirty, but instead of helping, each operation operates on both the old and the new bucket arrays.

Hash tables are also used to implement lookup tables, and *not* to reduce contention. In such cases the base approach still works: A single version number can protect the entire hash table. The only modifications to the hash table are still insertions and deletions into lists rooted in each individual bucket. Resizing the hash table is also a single write — the hash table is enlarged by overwriting the pointer to the bucket array. A client calling a routine that determines it is time to resize the hash table creates a new bucket array of the appropriate size (for example, the smallest prime number greater than twice the current number of buckets). It walks over the current hash table, inserting each entry into its private new copy of the bucket array. When it is finished, if the version number has not been changed, it replaces the old array with the new, and increments the version number.

D.5 Relaxing consistency requirements

Data structures often constrain independent words to maintain relationships with each other in order for the data structures to remain consistent. If these invariants are violated operations may fail.

Invariants that relate $k > n$ independent words, where primitive atomic operators have arity n, *must* be violated at some point. If only k words are updated, $n - k$ words may be inconsistent. A process running an algorithm may halt while the object is in such an inconsistent state. Algorithms that use locks never expose these states unless a process fails. Using NBS, these states may be exposed in the normal course of execution.

A programmer can deal with this issue by relaxing the consistency requirements. Consistency, with respect to the data structure, may be redefined to allow these formerly inconsistent states to exist – operations must now be prepared to deal with the larger variety of states. The key is to redefine consistency in such a way that each new operation can maintain the (new, relaxed) strict consistency by modifying only one or two words.

Two alternatives exist: First, the contract with the client is unchanged. In this case, all recovery from the new intermediate states is handled *internal* to the data structure. Second, the new states are exposed to clients — clients must be reprogrammed to handle the new semantics of the data structure.

D.5.1 Making internal intermediate states consistent

The list-based FIFO queue implementation of Michael and Scott [70] is the most efficient non--blocking implementation in the literature. It out-performs all blocking and non-blocking implementations.

Their basic approach is similar to Treiber's stacks: they include a version number with each pointer. However, version numbers by themselves are not sufficient to implement an efficient non--blocking FIFO queue.

The usual invariant for FIFO queues that allow insertion at the tail and deletion from the head, is to require the queue's `tail` pointer to always point at the "last" element — an element whose `next` pointer points to NULL. Michael and Scott's main insight was to relax that invariant and allow `tail` to point to *any* element in the queue. The nearer the tail, the more efficient, but even if `tail` happens to point to the head of the queue, operations are still correct.

Michael and Scott implement the version numbered pointers by using a double-width `CAS` (not `DCAS`). Each pointer is two words wide. The first word contains the actual pointer and the second word contains a version number. The version number guarantees that deletions from the head of the list occur correctly.

Dequeuing an entry from the head of the queue is done by reading the current `head` (and version number), and remembering the `next` pointer of that head entry. Using `CAS`, atomically replace the `head` of the queue with the second element of the queue (the saved `next` pointer) if-and-only-if `head` has not changed (the version number detects changes). The empty queue contains a dummy entry, so head and tail never both point at the same entry in a non-empty queue.

Without the version number, the following sequence would corrupt a queue initially containing only A and B:

Process 1	Process 2
(Begin `Dequeue`)	
Read `head` (= A)	
Read `head->next` (= B)	
	Dequeue A
	Dequeue B
	Enqueue A
	Enqueue C
Replace A in `head` with B	

The queue head now points to B, which may be on another list (even the free list)! Meanwhile `tail` points to C, and the data structure is incorrect. Adding version numbers to pointers avoids this problem, because if any intervening dequeues or enqueues occurred our attempt to change `head` from A to B fails.

Enqueuing on `tail` is slightly trickier, since both `tail` and the `next` pointer of the last entry in the queue need to be updated. The authors solve this problem by relaxing the requirement that `tail` always point to the last entry in the queue. As long as `tail` is always guaranteed to point to some entry in the list, the actual tail can be found by following the `next` pointers until you reach NULL. At the start of enqueue (or dequeue when `head` == `tail`), incorrect tail pointers are updated. (This algorithm is explained in more detail in Section D.7.1). To enqueue an entry at the `tail` of the queue, it is now correct to proceed in two independent steps. First, atomically replace the next pointer of the last entry in the queue (which must currently be NULL) with the pointer to the entry being inserted. The queue is now in a valid state, since `tail` can legally point to any element in the queue. Then, in a separate step, update `tail` to point to the newly inserted entry if-and-only-if `tail` is still pointing to the entry that was at the tail of the queue when this operation began (and has the same version number). If not (in the rare case that a conflicting update completed while we were enqueuing), a subsequent enqueue or dequeue is guaranteed to fix up `tail`.

D.5.2 Expose more states to robust clients

It is sometimes the case that implementing a non-blocking version of a library routine is difficult, particularly in routines being ported from one system to another. In such cases, rather than always trying to enforce the original semantics, it is sometimes preferable to expose more states to clients and re-implement the clients to deal with the new variety of states. This redefinition not only finesses a difficult problem, but it can also enhance the robustness of the system as a whole by managing unusual situations at the highest level at which they are meaningful. (Unfortunately, it is not always possible to redefine clients — especially for widely used modules with many existing clients — in such cases this approach is not applicable.)

The value of exposing more states to clients is best illustrated by a concrete example. Consider signal delivery in the Cache Kernel. There, signal delivery was made unreliable to simplify the code that ensured that signals were only delivered to the correct process. Clients were explicitly required to recover from unreliable signal delivery. Since there were other reasons that signal delivery was not always reliable, this change actually increased the robustness of the system.

In the Cache Kernel [24], objects managed by the kernel are only "cached" versions of objects owned by higher level *application kernels*. This is analogous to well-understood memory caches. If the kernel must manage a new object, and no additional space exists in the kernel's cache for that type of object, the kernel looks for some object to flush from that cache. Dirty objects are "written back" to the client (application kernel) before being flushed from the cache. However, the same kernel object is now a cached version of a *different* client level object.

At the Cache Kernel/application kernel interface objects are referred to by *ckoid*s (cache kernel object IDs) rather than raw pointers. `Ckoids` are 32-bit quantities composed of an index into the object cache (effectively a pointer) and a version number. This convention is designed to eliminate the problem of pointers to cached objects going stale, by checking the version number against the version number in the kernel's cache.

However, inside the kernel such approaches do not always work.

Consider the following example. Process objects, like all other objects managed by the kernel, can be written back to their Application Kernels. Each process object has a version number that is updated when it is taken off of the list that indicates the process is still alive.

There is also a signal TLB for improving the performance of vectoring signals into user space. When a signal arrives on physical address `physaddr P`, the system looks `P` up in a hash table. It scans the hash chain (of TLB entries) for an entry `E` that matches `P`. `E` contains a virtual address (reverse-translation) as well as a pointer to the signalling process `S`.

How do we synchronize adding a TLB entry that refers to `S` with writing back process `S`? We must ensure that we do not deliver signals to whatever process ends up occupying `S`'s slot (and translate `P` to the wrong virtual address).

One option is to use the same approach as at the Cache Kernel/Application Kernel interface and use ckoid's rather than raw pointers. We can (and do) tag the pointers in the TLB entries with the version number we expect the process to be. Unfortunately, this approach is not sufficient. The entries can sit around forever, and there is no way to prove that the version number will not be reused. (Version numbers are only 13 bits long).

We use the following approach to implement thread deletion.

1. increment ckoid version.

2. take thread out of list of threads.

3. copy into writeback buffer.

4. finally, invalidate signal TLB.

(This can be generalized to any delete sequence that invalidates the signal TLB after incrementing the `ckoid` version.)

We use TLB entry insert code like this:

1. find the destination process by ckoid

2. create a TLB entry labelled ”suspect”

3. check that the ckoid is still valid. If so, clear the suspect bit. Else (if it is no longer valid) remove TLB entry.

Given insertion and deletion code structured in this way, the signal delivery code has a choice. It can either spin on the suspect bit, or simply ignore suspect entries. If it spins, care must be taken to avoid deadlock — the resulting code is not non-blocking. If the delivery code skips suspect entries, then we have seemingly changed the semantics of signal delivery: we must drop signals in this race case. (Consider a signal which arrives between inserting a suspect TLB entry and clearing the suspect bit.)

Upon closer inspection, dropping signals in this race case is acceptable — and not simply because it is an unlikely case. The client code should *already* be robust in the face of dropped signals, since signals can be dropped for other reasons.

The system, as a whole, gains robustness by moving the recovery code up to the level of the system which can meaningfully deal with the failure. Building extra reliability and consistency at the lowest levels of the system can lead to inefficiency[86]. Care must be taken, however, to apply this technique appropriately. Sometimes it *is* necessary to solve hard problems — but, in our experience, situations that require difficult non-blocking solutions are always worth flagging to see whether the problem is best solved at a higher level. In this case, if we assume that signal delivery *must* be made reliable, then the alternatives to relaxing the semantics (or making clients more robust) are either to accept a blocking implementation or to have insertion, deletion and signal delivery use more expensive techniques derived more directly from the universal protocols described in Chapter 3. In this case, relaxing the lower level semantics and implementing reliability at higher levels seems the best choice.

One final note. The algorithm that allows us to drop signals, as described here, cannot succeed without the "suspect" bit. Although insertion that overlaps with deletion will not *leave* an invalid TLB entry in the table, a signal that arrives during insertion cannot distinguish between an (invalid) TLB entry that is going to be deleted and a (valid) TLB entry that will stay. There is no simple way to only insert "valid" entries, since deletion can always occur between determining that a process is valid and the actual insertion.

D.6 Copying small descriptors

There are data structures that maintain invariants relating many individual words of state. The previous section assumed that only one or two words need be written in any given update. This one or two word limit either arose naturally or was achieved through techniques described in this Appendix. However, it is not always practical to design one or two word incremental updates. In data structures where such updates are not possible, we can group fields that are updated together into a distinct descriptor. The descriptor is referenced from, but not included in, the object's header. Operations modify a copy of this descriptor (possibly copying some fields that are not being modified), and atomically install a new, modified, copy as part of the operation. For variable size data structures the amount of data copied is constant — it is proportional to the size of the descriptor regardless of the number of entries in the object. The same technique can be used to update multiple fields per-entry with a single write. If operations on the data structure only involve updating a single entry (regardless of the number of fields) and any number of fields in the header, `DCAS` supports a simple atomic non-blocking update using this technique.

Consider extending the linked list of Section D.2.4 to include the length of the list (it is easy to see how to add the maximum value element and the minimum value element, too, for example. I only deal with a single attribute, `length`, here to keep the example simple.).

We require a unique ID to detect conflicts between updates to this list. It is insufficient to merely keep a version number *in* the descriptor. `DCAS` cannot simultaneously check that `list->fields` still points to the descriptor, *and* that `uniqueId->versionNumber` has not changed, *and* that

```
int incrRefCount(T **address, T *object)
{
  int count = object->refCount;
  while (!DCAS(address, &(object->refCount),
               object,  count,
               object,  count+1)) {
    if (*address != object)
      { return(FALSE); }
    count = object->refCount;
  }
  return(TRUE);
}
```

Figure D.13: Definition of `incrRefCount`: to ensure that it is safe to use and copy the object, `incrRefCount` only succeeds if it can atomically increment the reference count *while* a pointer to `object` is still stored in `address`.

```
copy = new listFields;
do {
  if (uniqueID != NULL) /* NULL 1st time */
    { decrRefCount(uniqueId); }
  backoffIfNeeded();
  uniqueId = list->fields;
  if (!incrRefCount(&(list->fields),uniqueId))
    { uniqueId = NULL; continue; }
  *copy = *uniqueId;
  copy->length++;
  copy->refCount = 1; /* In case we succeed */

  for (p=list->head;
       (((next = p->next) != NULL) &&
        (next->datum < datum));
       p=next) {}
  entry->next = next;
} while  /* retry if list changed */
  (!DCAS(&(list->fields), &(p->next),
         uniqueId,        next,
         copy,            entry));
decrRefCount(uniqueId);
decrRefCount(copy);
```

Figure D.14: Insertion into middle of a singly-linked list that maintains a count of length.

```
copy = new listFields;
if (uniqueID != NULL)
 { decrRefCount(uniqueId); }
do {
  backoffIfNeeded();
  uniqueId = list->fields;
  if (!incrRefCount(&(list->fields),uniqueId))
    { uniqueId = NULL; continue; }
  *copy = *uniqueId;
  copy->length--;
  copy->refCount = 1; /* In case we succeed */

  for(p=list->head;(p->next!=elt);p=p->next){
    if (p == NULL) {  /* Not found */
      if (uniqeId != list->fields)
        { continue; }    /* Changed, retry */
      return (NULL);  /* Really not found */
    }
  }
} while(!DCAS(&(list->fields), &(p->next),
              uniqueId,        elt,
              copy,            elt->next));
decrRefCount(uniqueId);
decrRefCount(copy);
```

Figure D.15: Deletion from the middle of a singly-linked list that maintains a count of list length.

`p->next` still points to `elt`.

The address of the descriptor now replaces the version number in the original linked list example. For the address to function correctly as a unique ID, we must guarantee that `uniqueId` is not reused (freed, reallocated, and reinstalled in `list->fields`) while any process has a `copy` of `uniqueId`. To ensure this, we increment a reference count on the descriptor before copying `uniqueId` to `copy` and decrement the reference count after `DCAS`. `uniqueId` is not returned to the free pool unless the reference count reaches 0, so if the `DCAS` succeeds we are guaranteed that it is still the *same* instance of `uniqueId` and no modifications have been made to `list` in the meantime. This reference count also protects the non-atomic copying operation, so we can avoid the extra check words used in Herlihy's general methodology. Finally, note that the increment and decrement of the reference count need not be performed atomically with the state update.

D.7 Helping

For data structures with inexpensive operations, `DCAS` enables a simple form of *helping*, or *roll-forward*. The basic idea is straightforward. At the start of each transaction the process about to update the data-structure "registers" its operation and arguments in the data-structure. If it is preempted by another process before it completes, then the other process completes the original transaction before beginning its own.

D.7.1 Implicit Helping

The `CAS`-only FIFO-queue implementation of Michael and Scott[70] employs *implicit helping* to make sure the tail-pointer is correct. Entries are enqueued on the tail of the queue. If a process succeeds in setting `tail->next = entry;`, but is preempted or fails before executing `tail = entry;`, then the preempting process finds `tail` pointing to a queue-entry with a non-NULL next pointer. This must mean that an `enqueue` operation is incomplete, and it sets `tail` to the last entry in the list, "helping" the previous `enqueue` operation finish before it executes its own operation. This "registration" is implicit. Once you perform the first modification in the enqueue, you are implictly registered to update the tail pointer, too.

Another way of understanding the Michael and Scott algorithm is to recognize that the tail pointer is only a hint used as a performance optimization. (The use of hints to aid non-blocking implementations is described more fully in Section D.9.) We can detect an incorrect `tail`: the `next` pointer is non-NULL. We can recover if `tail` is incorrect: it is always possible to find the tail of a FIFO queue by following `next` pointers even if `tail` is not up to date. `tail` can still be useful as a performance optimization: a tail pointer that points to any entry past the second element in the queue *does* reduce the search time to find the tail. The only correctness requirement is that `tail` point to an entry that is actually in the list. The Michael and Scott algorithm simply defers updating `tail` until a non-preempted operation completes, or a dequeuer tries to remove the entry pointed to by `tail` from the queue. It is easy to see that any operation that fails to update `tail`, or defers updating `tail`, is nevertheless guaranteed to leave the list in a correct state.

D.7.2 Explicit Helping: roll-forward

Sometimes implicit helping is inefficient (or insufficient). Consider the case of doubly-linked lists using `DCAS`. One could imagine helping being invoked implicitly if you ever discover that `node->next->prev` is not equal to `node`. (I use this approach as an example of the technique in Section D.9, "Consistency and hints"). In such a case, sometimes search must be used to recover the correct `prev` pointer. This can occasionally be expensive for long lists experiencing heavy contention.

The alternative is *explicit helping*. In the doubly-linked list example, we add a field `nextOp` to the header of the list. This, combined with suitable updates to the version number, provides enough information to allow helping.

The general idea is that each process, P, registers its intended operation in the header of the doubly-linked-list. Once P successfully registers, it is guaranteed that its operation will complete within the next 4 steps that any process (including P) takes.

We keep a version number with the list which is incremented with every valid step any process takes. The version number (mod 4) tells us what step a process is up to in performing the operation and item specified in `d->nextOp` (we take advantage of the assumption (here) that list entries are (at least) word aligned, so the bottom bit will always be 0 and we can encode INSERT or DELETE in it). When step = 0 (i.e. the version number is congruent to 0 mod 4) it is legal for some process to try to insert its own item and operation in `nextOp`. Once an operation and item are successfully installed in `nextOp`, then all processes co-operate on performing `nextOp`.

The state transitions are straightforward and can be tracked by following the `DCAS`'s in the code below. The only *slightly* non-obvious step is the need for extra `DCAS`'s in step 1 of INSERT. They are needed because a helper might have gotten ahead of the original inserter, so the inserting process cannot assume that `entry` is not yet in the list. If `entry` has already been inserted, the inserting process cannot set the `next` pointer (or the back-pointer (`prev`)).

```
int dl_do_insert (dl_list *d, int version, (void *) entry)
{
  for (p = d->head;
       (((next = p->next) != NULL) &&
        (next->datum < entry->datum));
       p = next) {}
  DCAS(&(d->version), &(entry->next),
       version,       (void *)NULL,
       version,       next);
  DCAS(&(d->version), &(entry->prev),
       version,       (void *)NULL,
       version,       p);
  DCAS(&(d->version), &(p->next),
       version,       p->next,
       version+1,     entry);
}

int dl_do_delete (dl_list *d, int version, (void *) entry)
{
  p = entry->prev;
  new = entry->next;
  if DCAS(&(d->version), &(p->next),
          version,       entry,
          version+1,     new) {
    /* Only one xaction can get in here */
    entry->next = entry->prev = NULL;
  }
}
```

Figure D.16: Main step for insert and delete into doubly linked list. These routines set the forward pointer (`next`) for doubly linked lists.

```
int dl_list_op (dl_list *d, int op, (void *) entry)
{
  int version, started = FALSE, start = 0;
  int entryOp = (int)entry | DL_OP_INSERT;

  do {
    version = d->version; int step = version%4;
    void *oldEntry = d->nextOp;
    int oldOp = (int)oldEntry & 0x1;
    oldEntry &= ~0x1;

    switch (step) {
    case 0: /* register in nextOp */
      if (DCAS(&(d->version), &(d->nextOp),
          version,            (void *)NULL,
          version+1,          (void *)((int)entry|op))) {
        started = TRUE; start = version;
      }
      break;
    case 1: /* set next pointer */
      if(op==DL_OP_INSERT){dl_do_insert(d, version, oldEntry);}
      /* else op == DL_OP_DELETE */
      else                {dl_do_delete(d, version, oldEntry);}
      break;
    case 2: /* fix up backpointer */
      if (op == DL_OP_INSERT) { new = oldEntry; }
      /* else op == DL_OP_DELETE */
      else                    { new = oldEntry->prev; }
      DCAS(&(d->version), &(oldEntry->next->prev),
           version,       oldEntry->next->prev,
           version+1,     new);
      break;
    case 3: /* unregister op */
      DCAS(&(d->version), &(d->nextOp),
           version,       d->nextOp,
           version+1,     (void *)NULL);
    }
  /* Keep going while our Op has not been started yet,
   * or we're in the middle of operating on ours.
   */
  } while (!started || (version>=start && version < start+4));
}
```

Figure D.17: Main utility routine to perform operations on doubly-linked lists.

```
dl_list_insert(dl_list *d, (void *) datum)
{
  dl_list_entry *entry = new (dl_list_entry);
  entry->datum = datum;
  dl_list_op(d, DL_OP_INSERT; entry);
}

dl_list_delete(dl_list *d, (dl_list_entry *) entry)
{
  dl_list_op(d, DL_OP_DELETE; entry);
}
```

Figure D.18: Insertion and deletion calls from the middle of a doubly-linked list.

D.7.3 Roll-back

One should note that if the objections to helping (e.g. page faults) raised in section B.3 apply in a case where doubly-linked lists are needed, the algorithm for doubly-linked lists presented here can be adapted to *roll-back* rather than roll-forward.

The idea behind roll-back is to restore the data-structure to a previous consistent state, rather than going forward with a half-finished operation to a new consistent state. (The use of roll-back is no longer *strictly non-blocking* , since there is now a small possibility that *every* operation is rolled back and no progress is made.) The roll-back is performed by inspecting `version` mod 4 and the `nextOp` field. There is already enough information to undo any steps already taken.

The modification to support roll-back is straightforward.

Consider a client, C, attempting to delete an old entry or insert a new entry. If the version number is not congruent to 0 mod 4, then a partial operation is in progress by some earlier process, P. C must undo each step of the original operation, while simultaneously incrementing `version` by 7.

If we simply decrement version for each step we undo, then the original process, P, might see the version number repeat during C's operation. Incrementing by 7 decrements step by 1 mod 4, but produces a new version number guaranteeing that P will know not to proceed. We increment by 7 for *each* step we undo in case another process, C' tries to help undo P's operation also. If it succeeds, and proceeds, then C might see a repeated version number due to steps taken by C'. If so, C may try to continue undoing an already undone operation.

When the operation is undone and P's `nextOp` is replaced by C's `nextOp`, C may proceed.

P will keep trying to start over from scratch until it succeeds in executing all 4 steps itself — there is no helping, so if P does not finish the operation, no one did.

An alternate approach to incrementing by 7 on every step is to reserve one bit from `version` to use as `direction`. `direction` can be either `FORWARD` or `BACK`. Then we increment by 4 only for the transition between `FORWARD` and `BACK` or between `BACK` and `FORWARD`. After that we can simply decrement, because there will be no repetition of version numbers.

The schemes are roughly equivalent except for how long it takes the version number to wrap around. Reserving a bit reduces the version space by a factor of two regardless of contention. Incrementing by 7 potentially increases the size of an operation (in version number space) by a factor of 6 — from 4 to 24 — but only if there is roll-back. Given our experience with low contention on shared data structures, the increment-by-7 method is probably most suitable.

D.7.4 Hybrid approach: roll-back and roll-forward

It is also possible to extend this implementation to support an approach that *mixes* roll-forward with roll-back. Each client that finds a transaction in process decides (based on some case-specific criteria) whether to roll-forward or roll-back in order to proceed. For simplicity, we assume that once roll-back starts for a given operation we do not switch back to roll-forward.

The modifications to the implementation are once again fairly straightforward.

We reserve the lower bit of `version` to encode `direction`: either `FORWARD` or `BACK`. We again consider the case of a client, C, attempting to delete an old entry or attempting to insert a new entry. If the version number is not congruent to 0 mod 4, then a partial operation is in progress by some earlier process, P. If C elects to roll-forward, the behavior is the same as the original implementation. If C elects to roll-back, then C first sets `direction` to `BACK` so that the original process knows not to take forward steps, and simultaneously increments the version number by 4. If we do not signal a change in direction, then P and C might cause `step` to oscillate between 1 and 2, infinitely looping, causing livelock, in violation of the rule against transitioning from roll-back to roll-forward.

We also need some way to let P know when to quit trying to perform its operation. It can no longer determine from the version number. Assume P is making progress and reads version number 103, and then goes blocked. If it resumes to find version number greater than 108, it is possible that a helper rolled-forward to 104, or that a client rolled-back to 108. Since it cannot distinguish between them, we need some way that a helper can communicate success or failure back to P.

Clearly a per-list bit associated with `nextOp` is insufficient, since many operations can be outstanding at a given time. We need some form of per-operation state. A table consisting of entries for each process has drawbacks. First, the size scales as the total number of possible processes in the system, and not as the size of the actively contending processes. Second, per-process state is insufficient if asynchronous handlers can execute operations that contend with operations in the process whose stack the handler is running in.

The solution is to allocate a record from the heap to encode and store `nextOp`. It now includes `entry`, `operation`, and `status`. No synchronization is needed to write `n->entry` or `n->operation` since there is only one writer, and it has exclusive ownership of these two fields during writing. Afterwards, the fields are read-only. Access to `nextOp->status` is protected by adding another stage to the step in the version number. The only potentially tricky issue is taking care to be explicit about the "commit" point. If the convention specifies that once `status` is `SUCCESS` it will never change, you are no longer free to roll-back after that point.

Section D.2.4 and Chapter 3 discuss using the address of an object in shared memory as a unique ID, rather than depending on a version number. Data-structures that allocate a record from the heap for each operation are prime candidates for such unique IDs. Recall that version numbers are a performance optimization to save the cost of heap allocation and reference count. Unique IDs are guaranteed to be unique, while version numbers depend on (convincing) probabilistic arguments to ensure correctness. Given the cost of allocating `nextOp` from the heap, extending that to support unique IDs rather than version numbers is a logical step – the incremental cost of maintaining the reference count is small. If we add a unique ID to the doubly-linked list, we would add `step` and `status` fields to the unique ID and need no longer worry about decrementing, or where to store the return value.

D.8 Adding bits to invalidate states

Most of the preceding techniques discussed ways to reduce the number of words of state (or words per update) to one or two, to enable direct implementation using `DCAS`. This section proposes an alternative technique. Rather than striving to reduce state, we can augment the original sequential data structure with extra bits in strategically located fields. The redundant bits can be viewed as compact (possibly lossy) encodings of *other* state, not normally co-located with this field.

The simplest use of this technique is to use double-width pointers that incorporate a version number. The version number does not encode the *value* of any other field in the data structure, but *does* record the fact that a state change has occurred. There is clearly increased cost compared to the sequential specification: the storage required for each object is larger and dereferencing the pointer requires an extra indirection. However, adding version numbers allows a `CAS` or `DCAS` to atomically test and fail even if a pointer is reassigned to an old value. I have described many examples of this technique, such as Treiber's stacks [84], Michael and Scott's FIFO queue [70], and Arora, Blumofe and Paxton's deque implementation [11], among others.

D.8.1 Tag Bits: Concurrent Heap

Another example of increasing the size of an object in order to encode extra information occurs in the heap-based priority queue implementation of Israeli and Rappaport [46].

A *heap* is a complete binary tree in which each node has a value with equal or higher priority than either of its children. The highest priority item is at the root of the heap. The standard implementation of a heap is in an array, where the left and right children of the node stored at index i are stored at indices $2i$ and $2i + 1$, respectively.

`Insert` places a node at the first open array index (if the heap is of size N, index $N + 1$ is the first open space), and proceeds to swap it with its parent, until the new node propagates up to a point in the heap where it has lower priority than its parent.

`DeleteMin` generally removes the root of the heap, and takes the node at the greatest occupied index (if the heap is of size N, index N is the greatest occupied index) and places it at the root (decrementing the size of the heap). The low priority node at the root is now (recursively) swapped with its highest priority child, until it reaches a point when it is either at a leaf, or it has higher priority than either of its children.

The heap is potentially inconsistent during insertion or deletion. During insertion the node that is in the process of bubbling up may not be of lower priority than its parent. During deletion, the node that is being swapped down may not be of higher priority than both of its children. These states may be exposed during a non-blocking update. A node that violates the order invariant can cause a subsequent insertion to terminate prematurely, permanently leaving the heap out of order. It can also confuse a delete operation in indeterminate ways.

The non-blocking version proposed by Israeli and Rappaport in [46] augments the heap's data structure by storing extra state at each node. The new state specifies whether this node is in the process of being floated up (insertion), or swapped down (during deletion). Operations can execute concurrently, without interference until they encounter a node that is moving either up or down. Given the extra state, the new insertion or deletion can make the heap consistent before proceeding. One process may have to help another process in order to convert a subtree into a state where an operation can complete.

The swaps are performed using `DCAS` functionality, thus requiring no auxiliary storage. (I ignore many subtle details in the interest of space).

It is interesting to compare this algorithm to a universal transformation, such as the one presented in Chapter 3. The heap-specific algorithm appears to possess three advantages over the universal construction. First, no logging is required. Second, it can safely roll-forward rather than simply roll-back. Third, many operations can execute concurrently assuming they do not conflict.

The first two advantages (no logging and roll-forward) result from the fact that "how to proceed" is trivially encoded in the data structure as part of the swap (using `DCAS`). As mentioned, each node includes a two-bit wide field specifying whether the value is in the process of percolating upward or downward or is stable. If an operation encounters a "moving" node while inserting or deleting, it knows that the subtree rooted in this node is out-of-order in the heap. Enough information is provided to finish the previous operation before performing the current operation on a newly consistent heap.

The third advantage, increased concurrency, results from the observation that if two operations access disjoint sets of nodes, then their results will be consistent, regardless of their order of execution. If they do overlap, then the helping done by the algorithm ensures that their relative order of execution is identical for every node in which they overlap.

On the other hand, despite these advantages, it is important to note the actual savings accrued are small. The costs are similar even though this algorithm avoids logging, because in both algorithms every write must be done with `DCAS`, and the logging (three unsynchronized writes) is just noise compared to the `DCAS`. The other advantages only arise under relatively high contention: with low contention, there are few opportunities to exploit concurrency, and roll-back or roll-forward rarely occur so the difference between them is not significant.

D.8.2 Invalidation: Concurrent Deque

Another common technique augments the sequential specification by increasing the number of different values a given field may take. The simplest such technique is to invalidate a field by replacing a consumed data value with a distinguished `INVALID` marker.

A concurrent deque implementation can serve as an example. Our first attempt at a non-blocking deque implementation (in section D.2.5) was strictly sequential. It disallowed any concurrent accesses to the deque. Conceptually, though, operations at the top of the deque should not affect operations at the bottom. Both should be able to proceed in parallel (when there is more than one element in the deque).

Simply treating the two ends independently will not guarantee consistency, since we must make sure that we do not simultaneously `dequeue` one end past the other. Figures D.19 and D.20 contain an implementation of a concurrent deque that solves the problem of simultaneous `popBottom` and `popTop` on a one element deque by augmenting the set of values that each element can hold.

By adding more values to elements of the deque that are outside the valid range, we can prevent both sides simultaneously dequeuing, thus allowing `top` and `bottom` to be stored in two independent words and to proceed mostly in parallel. The only requirement is to reserve at least one value for entries in the deque that represent invalid bit patterns. This can either be a reserved "`INVALID`" value, or else the type of value stored in the deque might already have "undefined" values. For example, on some machines pointers must be aligned, i.e. an integral multiple of the word size in bytes. An odd pointer is impossible in the normal course of events, so we can reserve them to mean `INVALID`).

Given an `INVALID` entry, it is easy to see that `popbottom` and `poptop` cannot both pop the same word (or pop past each other). If `popbottom` and `poptop` both try to execute concurrently on the same word (the only word in the deque), then only one can succeed. The first person to succeed converts the entry to `INVALID`, guaranteeing that the second pop will fail.

Even if bottom and top pointed at the same entry when the deque is empty, then we still would not enqueue two entries in the same location, since enqueue will only succeed if the entry were originally `INVALID`, and so only one enqueuer (bottom or top) could succeed. In fact, however, this is unnecessary, since bottom and top grow in the opposite directions, so there's no conflict and no synchronization needed even on simultaneous enqueues from both directions on an empty queue.

A final note: the heap algorithm of Israeli and Rappaport [46] use this same trick of `INVALID` entries to ensure that the heap size is consistent with the number of entries actually in the heap. By checking that `array[size+1]` contains `INVALID`, multiple operations can coordinate the transfer of the lowest entry to the root during a `DeleteMin` operation.

D.9 Consistency and hints

Lampson [56] recommends using hints to speed up normal execution. A *hint* is the saved result of some computation. However, a hint may be wrong.

Because a hint is used to speed up normal computation, it must be correct most of the time. The bigger the gain of a successfully used hint relative to the "normal" computation, the more frequently

```
void pushTop(deque *dq, void *newval)
{
 do {
   bottom = dq->bot;
   top = dq->top;
   if (top == 0) {
     if ((bottom <= top) &&
         DCAS(&(dq->bot), &(dq->top),
              bottom,     top,
              INITPOS,    INITPOS)) {
       bottom = INITPOS; top = INITPOS;
     } else { ERROR(); }
   }
 } while (!DCAS(&(dq->top), &(dq->entries[top-1]),
                top,        INVALID,
                top-1,      newval));
}

void *popTop(deque *dq)
{
  do {
    bottom = dq->bot;
    top = dq->top;
    if (bottom <= top) {
      return((void *)NULL);
    } else {
      val = dq->entries[top];
    }
  } while ((val != INVALID) &&
           !DCAS(&(dq->top), &(dq->entries[top]),
                 top,        val,
                 top+1,      INVALID));
  return(val);
}
```

Figure D.19: Insertion and deletion at the top of the concurrent deque

```
void pushBottom(deque *dq, void *newval)
{
 do {
   bottom = dq->bot;
   top = dq->top;
   if ((bottom <= top) &&
       DCAS(&(dq->bot), &(dq->top),
            bottom,     top,
            INITPOS,    INITPOS)) {
     bottom = INITPOS; top = INITPOS;
   }
 } while (!DCAS(&(dq->bottom), &(dq->entries[bottom]),
                bottom,        INVALID,
                bottom+1,      newval));
}

void *popBottom(deque *dq)
{
  do {
    bottom = dq->bot;
    top = dq->top;
    if (bottom <= top) {
      return((void *)NULL);
    } else {
      val = dq->entries[bottom-1];
    }
  } while ((val != INVALID) &&
           !DCAS(&(dq->bot), &(dq->entries[bottom]),
                 bottom,     val,
                 bottom-1,   INVALID));
  return(val);
}
```

Figure D.20: Insertion and deletion from the bottom of a concurrent deque a hint can afford to be incorrect. Because a hint may be incorrect, there must be some (efficient) way to verify the validity of the hint before depending upon it. Verifying the validity of a hint is usually easier and cheaper than computing the value from scratch. Because verification may determine that the hint is invalid, clients must provide enough information to compute the correct value.

Hints are often useful when designing non-blocking algorithms. Often, complex algorithms are used because the simpler algorithms do not provide needed performance. We can redesign such complex algorithms as hint-based performance optimizations of the "simple" versions of the algorithms. This view may allow us to relax some of the consistency requirements of the more complex algorithm.

D.9.1 Doubly-linked lists

Section D.7 describes an implementation of doubly-linked lists that performs well even under high contention. It records the state of the computation in the low bits of the version number. Processes that try to insert or delete entries on a list that is not in phase 0, first "help" finish the previous operation (or undo it), and then attempt to proceed themselves. Unfortunately, this has an unnecessarily high cost in the common case of low contention.

A better approach is to only depend upon the `next` pointers to ensure integrity of the list. The `prev` pointers serve only as hints. A doubly-linked list is implemented using the singly-linked list implementation. After a successful insertion or deletion the next step is to update the `prev` pointer using `DCAS` while keeping the version number equal to the just incremented value. If the `DCAS` fails because another operation has intervened, the operation just gives up. The list is consistent, but the `prev` pointer is no longer a valid hint.

It must be possible to verify that a hint is valid and, if it is invalid, it must be possible to recover. A first attempt at verification that the `prev` pointer is valid is to check that `(entry->prev->next == entry)`. This test correctly determines whether the back-pointer is valid only if the list precludes the possibility that both `entry` and `entry->prev` might end up being neighbors on some other list!

In cases where a pair of neighbors may find themselves neighbors on a *different* list, a different form of verification is needed. Each entry must maintain an `entry->list` field which points to the list it is on. To simplify our example, we assume that each entry has a single, exclusive, owner when it is not on any list. This owner sets `entry->list` to `NULL` after a successful delete, and sets it to `list` before attempting to insert it on `list`. These assignments do not need synchronization since ownership is exclusive. If `entry->prev` is invalid, we require that `delete` updates the `prev` pointer of the entry it is about to delete before actually performing the delete.

Given a correctly maintained `entry->list` we can verify that the `prev` pointer is a valid hint. First, read `list->version`. Second, check `(entry->prev->next == entry)`. Then check that *both* `entry->list` and `entry->prev->list` are pointing to `list`.

If `delete` did not guarantee that `entry->prev` was correct *before* deleting `entry`, then the following case would cause deletion to fail.

> Consider three consecutive elements in a list, a, b, and c. Process P_0 attempts to delete b, and processes P_1 and P_2 both attempt to delete c.
>
> P_0 deletes b, however `b->next` still points at c, and `c->prev` is not yet updated. P_0 is then delayed or stopped before setting `b->list` to `NULL`.
>
> P_1 successfully deletes c, however `c->prev` still points at `b` because P_0 did not update it. Assume that P_1 is stopped or delayed at this point and does not set `c->list` to `NULL`.
>
> P_2 has a reference to c and attempts to delete c from `list`. It reads the version number, determines that `c->prev->next == c`, and that `c->list == list` and that `b->list == list`. P_2 will incorrectly decide that it can safely and successfully delete c from `list` (again).

The modification of `delete` to guarantee that `entry->prev` was correct *before* deleting `entry` ensures that even this case works correctly. In the previous example P_1 would first try to set `c->prev = a;` thereby allowing P_2 to detect that c was already deleted.

D.9.2 Array based sets

Many data-structures, such as doubly-linked lists, have version numbers or some other mechanism that allows operations to easily detect interference and abort the update of the hint. This is not always possible. In such cases, the technique of "invalidation" can be used.

Suppose we want to extend the implementation of array-based stacks from Section D.2.2 to supply an implementation for an array-based `set`. The complexity of the implementation increases over stacks, since we need to be able to remove entries other than the top. The point of array-based sets or collections is to support efficient insertion and deletion of objects without searching for the position of a given element. (Lookup costs cannot necessarily be reduced). Insertion can be done at the first free location, at constant cost. Deletion can be accomplished without search by binding an array-index together with the value stored in the set. Clients refer to set elements through an extra level of indirection, an `entry`, in order to have access to the array-index. The indirection is required so that the set can update `entry->index` if the position of the element in the array has changed. The position can change if another client deletes an element from the middle of the set. It is desirable to maintain a compact representation of the set (to reduce space costs and facilitate lookups), so if a gap opens up, elements will be moved down from high indices to fill the gap.

But this means that we must alter three independent locations on a `delete` operation: the `size` field, the location in memory from which we delete the element, and the index field of the new entry we move into the slot occupied by the old entry. Luckily, the `index` field (like the back-pointer in doubly-linked lists) can be viewed as a performance optimization. It is just a hint, we can easily (and cheaply) detect that it is incorrect (if `elements[index]` is not == `entry`). If `index` is incorrect, it is easy (but more expensive) to reconstruct — simply search `elements` for `entry`.

```
entry->value = val;
do {
  backoffIfNeeded();
  index = set->size;
  entry->index = index;
  if (index >= set->max)
    { /* signal set_overflow,
        * or allocate larger array */ }
  /* oldVal should really be NULL */
  oldVal = set->elements[index];
} while
  (!DCAS(&(set->size),
 &(set->elements[index]),
         index,        oldVal,
         index+1,      entry));
```

Figure D.21: Insert in set (array).

Unfortunately, there is no version number that protects the integrity of the entire data structure. Therefore there is no simple way to detect whether another process moved entries around while we were performing the search. This raises problems in designing a correct implementation.

Consider starting with the simplest possible implementation: `insert` reads `set->size` into `size`, and sets `entry->index` equal to `size` with no synchronization. It then (atomically) increments `set->size` (if it is unchanged from `size`) and sets `elements[set->size]` equal to `entry`.

`delete` first reads, with no synchronization, `entry` from `elements[entry->index]`,

```
setEntry *delete (Set *set, setEntry *entry)
{
  if (!CAS(&(entry->deleted),FALSE,TRUE)) {
    /* already deleted by someone else */
    return(NULL);
  }
  int index = entry->index;
  /* Is index correct? */
  if ((oldEntry = set->elements[index]) != entry) {
    if (oldEntry != NULL) {
      int newIndex = oldEntry->index;
      DCAS(&(set->elements[newIndex]), &(entry->index),
           entry,                      index,
           entry,                      newIndex);
    }
  }
  index = entry->index; /* index is now correct */
  while (TRUE) {
    backoffIfNeeded();
    top = set->size-1;
    SetEntry *topmost = set->elements[top];
    if (topmost && (topmost->deleted == TRUE)) {

      continue;
    }
    if (!DCAS(&(set->elements[top]),
                        &(set->elements[index]),
              topmost,  entry,
              entry,    topmost))
      { continue; }
    DCAS(&(set->elements[index]),
                    &(topmost->index),
         topmost,   top,
         topmost,   index);
    DCAS(&(set->size), &(set->elements[top]),
         top+1,        entry,
 top,          NULL);
    return(entry);
  }
}
```

Figure D.22: Delete from set (array).

reads `size` from `set->size`, and reads `topmost` from `elements[set->size-1]`. It then, atomically, both decrements `set->size` (assuming it has not changed) and stores `topmost` in `elements[entry->index]` (assuming it still contains `entry`). If both stores succeed, `delete` returns successfully to the caller, otherwise it retries.

This attempt almost works. Insertion is clearly correct (it is equivalent to `Push`). Deletion will never overwrite a value in the set, even if `set->size` changes value during the operation, because the second comparison in the `DCAS` will only succeed if `elements[entry->index] == entry`, in which case `entry` is returned to the caller.

However, because `elements[set->size-1]` is not read atomically with the update, it is possible that the top value at the time of the `DCAS` is no longer equal to the original `elements[set->size-1]`. Thus we can lose a value by decrementing `size` without copying the value back into the valid range.

The example illustrates this problem. Process 1 wishes to delete A from the set; Process 2 wishes to delete B and insert C. Assume A`->index == 3`, B`->index == 5` and `set->size == 10`.

Process 1	Process 2
(Begin `Delete` A)	
Read `elements[3]` (= A)	
Read `set->size` (= 10)	
Read `elements[9]` (= X)	(Begin Delete B)
	`elements[5]` = X
	`set->size` = 9
	(Begin Insert C)
	`elements[9]` = C
	`set->size` = 10
`elements[3]` = X	
`set->size` = 9	

The result of this sequence of operations is that X now appears twice in the set, and C is lost.

A solution is to adopt the notion of invalidating entries. Array-based stacks worked by setting `pop`'ed elements to `NULL`. We will do the same thing here. However, we do not want to allow `NULL`s inside the set, to maintain our compact representation. So we first swap `entry` with the topmost entry in the set (`topmost = set->elements[set->size-1]`). For this transformation to be valid, we must ensure that if the state of the set was valid before the operation, it is still valid after we swap entries. The new state is valid (with the order changed so that `entry` is now the topmost element of the set) — *if* we can guarantee that `topmost` was still in the set at the time of the swap. We guarantee this by requiring `delete` to write `NULL` in the place of deleted entries – then the swap would fail if `topmost` had been deleted. Given the swap, we can now atomically decrement `size` and overwrite `entry` with `NULL`.

Unfortunately, `entry->index` and `topmost->index` are incorrect.

We can avoid this issue by treating `index` as a hint; then we can set it in a separate step. After successfully deleting `entry`, we set `topmost->index` to `entry->index`. Unfortunately, this does not protect against the deleting process being delayed or failing. We can try to recover by having any deleter detect an invalid `index` and search for the item. As mentioned earlier, there is no convenient way to ensure that the search was not interrupted, or that the entry will not be moved between the time index is computed and when it is set.

Fortunately, `entry` will only be moved if some deleting process finds it as `topmost`. Therefore we know that `index` will only be decremented (it will never be incremented, unless entry is being deleted), and that the move to a lower index occurs in precisely one place in the code. Once we know that index is monotonically decreasing, we can implement a synchronization-free `find` routine. Simply start at a known previous value of index, or, if there is no known valid value of `index`, start at `size`. If the search is *downwards* in the array of elements, then no swap can cause an undeleted entry to disappear — it can only be copied to a location the search has not yet inspected. If a deletion moves `entry` up, skipping over the search, then the `delete` is serialized *before* the `find`. This implementation of `find` is not only synchronization-free, but also read-only and supports infinite concurrency.

Unfortunately, we might not want to pay the price of a search to fix up `index`. A further problem is that the implementation of `delete` is not strictly non-blocking. Consider two processees trying to delete `entry` and `topmost` respectively. They may alternate swapping `entry` and `topmost` so that each time one tries to decrement `size` the other had just performed the swap, causing the decrement to fail.

A solution to both these problems is to exploit the fact that the deleted `entry` contains the correct index for `topmost` and that `topmost`'s index is pointing to the deleted `entry`. To maintain this invariant, we need only know when the topmost element of a set is "about to be" deleted — that is, when it has been swapped but the indices have not yet been updated. To do this we add a field to each entry which allows us to mark an element as deleted. Before performing the swap, we atomically mark `entry->delete` to be `TRUE` (this must be atomic to guarantee that only one deleter of a given element succeeds). This moves the "commit" of the delete operation earlier, since once you set the deleted field, the entry is no longer considered part of the set[2].

Now the algorithm is straightforward. We will attempt to guarantee that an entry's index is either correct, or else pointing to an entry that was just swapped with this entry. To guarantee this we must alter the delete operation in two ways. First, if you try to delete an entry that has an incorrect index, you update index by looking at the index of the entry that your index is pointing at. Second, if the topmost element is marked deleted, then you must update the index of swapped entry, and then decrement `set->size` and set that entry to `NULL` before you proceed. This also guarantees that once you set `deleted`, then `entry` is never moved, and that the top entry in the set is always either deleted or else has a correct index.

After checking whether it is permissible to proceed, the actual `delete` operation now takes four steps. First, mark the entry as deleted. Then perform the swap. Third, update the `index` pointer of the (former) topmost element. Finally, decrement size and replace the entry with `NULL`.

D.10 Would CAS3 make a difference?

Given the advantages DCAS provides over CAS-only implementations, it is natural to ask the question: "will CAS3 provide similar gains over DCAS?". There is no definitive answer to that question, yet.

Empirically, I report that we have not encountered any important data-structures that were not efficiently implementable using DCAS that did have an efficient CAS3 implementation. Chapter 3 points out that there is a constant time implementation of CASn using DCAS, and so there is a reasonable expectation that the answer to whether CAS3 will provide similar gains over CAS2 will remain "no".

[2]If the deleter fails at this point (before the swap), the set is still consistent. The element might be lost, true, but this is always a possibility if the process dies *after* a successful deletion, too. We can depend on TSM audit code to (eventually) reclaim the entry.

Appendix E

Implementation of Type-Stable Memory Management

As noted in Chapter 4, it is simple to implement type stability if we allow t_{stable} to be infinite. However, this can result in poor utilization of memory if the system "phase-shifts". Never actually freeing the unused memory can result in serious under-utilization of memory. Similarly, although it is simple to implement type stability if we use a reference counting scheme (or a garbage collector), we reject this approach as being too expensive (e.g. updating reference counts while a process walks a list). However, implementing TSM with finite t_{stable} is still not difficult. We need only concern ourselves with a correctly functioning system. By this I mean that if an object is part of a data structure in the heap, then it will not be on a free list. If this property does not hold then the system may fail, independent of type stable memory. Thus, the only references we are concerned with are "temporary" ones stored in local variables, such as when a process traverses the individual objects of type T stored in a collection.

We first present a brief description of some approaches to reclaiming memory in the context of an operating system kernel.

E.1 Kernel implementation

If an entire page of a free list has been untouched for some time $t > T_{stable}$, then the only issue is whether a kernel thread has been "inside" some data structure for t, and might be potentially holding a temporary pointer to one of the free objects. There are two cases: either the kernel was entered through a system call, or else was executing in a kernel-only thread.

If the thread was entered through a system call, then we only need worry if any currently executing system call started longer than t units in the past. This is easy to check. If a system call is blocked, then we can (a) wait for it (i.e. delay and retry), (b) inspect its stack for pointers into the page(s), (c) note that some blocked system calls are "safe", or (d) abort it, (on systems that support UNIX EINTR semantics).

Kernel-only threads typically have a top-level function that loops waiting for some set of conditions to become true (a set of events). There is usually at least one "safe point" in the top-level loop where we know that the process is not holding on to any temporary pointers. The kernel thread registers with the TSM manager. Each time the kernel process reaches the "safe point", the process may set a timestamp and reset the allocator high-water mark. As long as it has reached the "safe point" t units in the past, we can safely delete all objects above the high-water allocation mark.

E.2 User-level implementations

Implementation of TSM for user-level code is not much more complex. Although user-level systems may act in more complicated ways than OS kernels, we can still generalize the timestamp trick. We implement a single per-type reference count and a timestamp, meant to record the number of temporary variables pointing to T (i.e. of type (T *)). This reference count is incremented once each time a temporary variable becomes active. In a blocking system, such an active temporary variable is almost certainly protected by a lock on some data structure containing a collection of objects of type T. The reference count is decremented when the variable goes out of scope. Thus the reference count is incremented/decremented once per activation, rather than once per reference to *each* object of type T. Whenever the reference count reaches 0 and the timestamp is older than t_{stable}, we record a new timestamp and set the high water mark to the current size of the free list. If the TSM manager finds that the timestamp is not more recent than t_{stable}, then all objects above the high water mark have not been referenced for $t > T_{stable}$ time units and are available to be returned to the general memory pool. Note that for types that have separate per-processor free lists the reference count is not shared by other processors. The overhead for TSM involves no non-local writes.

It is possible (though unusual) for a system to have a large number of processes all containing temporary variables pointing at objects of a very common data type. It is possible in such a system that the reference count described above will *never* go to zero, although the free list remains large. A simple solution for these data-types is to keep two reference counts, `rfcnt0` and `rfcnt1`. One counter is the currently active reference count. A process will always decrement the counter that it incremented, but it will always increment the currently active reference count. Thus, the inactive counter can only decrease.

We start with `rfcnt0` active. After some delay[1], we swap the currently active reference counter and `rfcnt1` becomes active. From then on, whenever the inactive counter reaches 0, it becomes the active counter. Each time `rfcnt0` becomes 0, we know that *every* pointer to objects of type T became active since the previous time that `rfcnt0` became 0. Therefore, we can treat the event "`rfcnt0` equal to zero" exactly the way we treat "ref-count equal to zero" in the single refcount scheme, waiting for t_{stable} and releasing all the type T pages to the general pool.

[1]The algorithm is correct for *any* value of delay as long as `rfcnt0` was incremented at least once. However, delays comparable to t_{stable} will reduce excess overhead. Shorter delays will impose needless work, longer delays will delay reclamation.

Bibliography

[1] Afek, Yehuda, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. “Disentangling Multi-object Operations.” In *Proceedings of the Sixteenth Symposium on Principles of Distributed Computing*, Pages 111–120, August 1997. Santa Barbara, CA.

[2] Allemany, Juan and Ed W. Felten. “Performance Issues in Non-Blocking Synchronization on Shared Memory Multiprocessors.” In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, Pages 125–134, August 1992.

[3] Anderson, James H. and Mark Moir. “Universal Constructions for Large Objects.” In *9th Intl Workshop on Distributed Algorithms ’95*, Pages 168–182. Springer Verlag, September 1995. Lecture Notes in Computer Science, Vol. 972.

[4] Anderson, James H. and Mark Moir. “Universal Constructions for Multi-Object Operations.” In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Pages 184–193, August 20–23 1995. Ottawa, Ont. Canada.

[5] Anderson, James H. and Srikanth Ramamurthy. “Using Lock-Free Objects in Hard Real-Time Applications.” In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Page 272, August 20–23 1995. Ottawa, Ont. Canada.

[6] Anderson, James H., Srikanth Ramamurthy, and R. Jain. “Implementing Wait-Free Objects on Priority-Based Systems.” In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, August 1997. Santa Barbara, CA.

[7] Anderson, James H., Srikanth Ramamurthy, and Kevin Jeffay. “Real-Time Computing with Lock-Free Shared Objects.” *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.

[8] Anderson, R. J. and H. Woll. "Wait-Free Parallel Algorithms For The Union-Find Problem." In *Proceedings of the Twenty-Third ACM Symposium on Theory of Computing*, Pages 370–380, 1991.

[9] Anderson, T. E. "The performance of Spin Lock Alternatives for Shared-Memory Multiprocessors." *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[10] Anderson, Thomas E., Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. "Scheduler Activations: effective kernel support for the user-level management of parallelism." In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, Pages 95–109, Asilomar, Pacific Grove, CA, October 1991. ACM SIGOPS. Published in ACM Operating Systems Review Vol.25, No.5, 1991. Also published ACM Transactions on Computer Systems Vol.10 No.1, pp.53–70, Feb 92.

[11] Arora, Nimar S., Robert D. Blumofe, and C. Greg Plaxton. "Thread scheduling for multiprogrammed multiprocessors." In *Proceedings of the Tenth Symposium on Parallel Algorithms and Architectures*, Pages 119–129, June 28 – July 2 1998. Puerto Vallarta, Mexico.

[12] Attiya, Hagit and E. Dagan. "Universal Operations: Unary versus Binary." In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, May 23-26 1996. Phila. PA.

[13] Attiya, Hagit and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, Maidenhead, Berkshire, England, 1998.

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

[15] Bernstein, Michael S. and Bradley C. Kuszmaul. "Communications-Efficient multithreading on wide-area networks (Brief Announcement, SPAA Revue)." In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1998. Puerto Vallarta, Mexico.

[16] Bershad, Brian N. "Practical Considerations for Lock-Free Concurrent Objects." Technical Report CMU-CS-91-183, Carnegie Mellon University, 1991.

[17] Bershad, Brian N. "Practical considerations for non-blocking concurrent objects." In *Proceedings 13th IEEE International Conference on Distributed Computing Systems*, Pages 264–273. IEEE Computer Society Press, May 25–28 1993. Los Alamitos CA.

[18] Bershad, Brian N., David D. Redell, and John R. Ellis. "Fast Mutual Exclusion for Uniprocessors." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Pages 223–233, 1992.

[19] Blumofe, Robert D., Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, keith H. Randall, and Yuli Zhou. "Cilk: an efficient multithreaded runtime system." *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[20] Blumofe, Robert D. and Charles E. Leiserson. "Scheduling multi-threaded computations by work-stealing." In *Proceedings of the 35th Annual Symposium on Foundation of Computer Science*, Pages 356–368, November 1994. Santa Fe, New Mexico.

[21] Brewer, Eric A., Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. "PROTEUS: A High-Performance Parallel-Architecture Simulator." Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[22] Chandra, Tushar, Prasad Jayanti, and King Tan. "A Polylog Time Wait-Free Construction for Closed Objects." In *Proceedings of the Seventeenth Symposium on Principles of Distributed Computing*, Pages 287–296, June 28 - July 2 1998. Puerto Vallarta, Mexico.

[23] Cheriton, David R. "The V Distributed System." *Communications of the ACM*, 31(3):314–333, March 1988.

[24] Cheriton, David R. and Kenneth J. Duda. "A Caching Model of Operating System Kernel Functionality." In *Proceedings of 1st Symposium on Operation Systems Design and Implementation*, Pages 179–193, November 14–17 1994. Monterey, CA.

[25] Cheriton, David R., Hendrik A. Goosen, and Philip Machanick. "Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience." In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Pages 23–31, April 1991. Tokyo, Japan.

[26] Cheriton, David R., Henk Goosen, and Patrick Boyle. "ParaDiGM: A Highly Scalable Shared-Memory Multi-Computer Architecture." *IEEE Computer*, 24(2), February 1991.

[27] Cheriton, David R. and Robert A. Kutter. "Optimized Memory-Based Messaging: Leveraging the Memory System for High-Performance Communication." *Computing Systems Journal*, 9(3):179–215, Summer 1996. Also available as Stanford Computer Science Technical Report TR-94-1506.

[28] Chesson, Greg. "Personal Communication." December 1996. SGI.

[29] Finkel, Raphael and Udi Manber. "DIB — a distributed implementation of backtracking." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(2):235–256, April 1987.

[30] Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." *Journal of the ACM*, 32(2):374–382, April 1985.

[31] Freeh, Vincent W., David K. Lowenthal, and Gregory R. Andrews. "Distributed Filaments: efficient fine-grain parallelism on a cluster of workstations." In *Proceedings of 1st Symposium on Operation Systems Design and Implementation*, Pages 201–213, November 14–17 1994. Monterey, CA.

[32] Gray, Jim and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, California, 1993.

[33] Gray, Jim and Daniel P. Siewiorek. "High-Availability Computer Systems." *IEEE Computer*, 24(9):39–48, September 1991.

[34] Greenwald, Michael B. and David R. Cheriton. "The synergy between non-blocking synchronization and operating system structure." In *2nd Symposium on Operating Systems Design and Implementation*, Pages 123–136, October 28–31 1996. Seattle, WA.

[35] Halstead Jr., Robert H. "Implementation of Multilisp: Lisp on a multiprocessor." In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, Pages 9–17, August 1984. Austin, TX.

[36] Heinrich, Joseph. *MIPS R4000 User's Manual*. PTR Prentice Hall, Englewood Cliffs, NJ, 1993.

[37] Hennessy, John L. and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, San Francisco, California, second edition, 1996.

[38] Herlihy, Maurice. “Impossibility and Universality Results for Wait-Free Sycnhronization.” In *Proceedings of the Seventh Symposium on Principles of Distributed Computing*, Pages 276–290, August 15-17 1988. Toronto, Ont., Canada.

[39] Herlihy, Maurice. “A Methodology For Implementing Highly Concurrent Data Objects.” *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[40] Herlihy, Maurice P. “A Methodology For Implementing Highly Concurrent Data Structures.” In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, Pages 197–206. ACM, March 1990. New York.

[41] Herlihy, Maurice P. “Wait-Free Synchronization.” *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[42] Herlihy, Maurice P. and J. Eliot B. Moss. “Transactional Memory: Architectural Support for Lock-Free Data Structures.” Technical Report CRL 92/07, Digital Equipment Corp. Cambridge Research Lab, 1992.

[43] Herlihy, Maurice P. and J. Eliot B. Moss. “Transactional Memory: Architectural support for Lock-Free Data Structures.” In *1993 20th Annual Symposium on Computer Architecture*, Pages 289–301, May 1993. San Diego, CA.

[44] Herlihy, Maurice P. and Jeannette M. Wing. “Linearalizability: A Correctness Condition for Concurrent Objects.” *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[45] IBM. “System/370 principles of operation.”. Order Number GA22-7000.

[46] Israeli, Amos and Lihu Rappaport. “Efficient wait-free implementation of a concurrent priority queue.” In *Proceedings of the Seventh International Workshop on Distributed Algorithms*, Pages 1–17. Springer Verlag, September 1993. Lausanne, Switzerland, also published as *Lecture Notes in Computer Science 725*.

[47] Israeli, Amos and Lihu Rappaport. “Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives.” In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, Pages 151–160, August 14–17 1994. Los Angeles, CA.

[48] James, David V. and David Singer. "Nonblocking Shared-Data Updates using Conditional Lock Primitives." In *Proceedings of the 2nd International Workshop on SCI-based High-Performance Low-Cost Computing*, Pages 67–73, March 21 1995. Santa Clara, CA.

[49] Jayanti, Prasad. "A complete and constant time wait-free implementation of CAS from LL/SC and vise versa." In *Proceedings of the Twelfth International Symposium on DIStributed Computing*, September 24–26 1998. Andros, Greece.

[50] Jayanti, Prasad. "A Lower Bound on the Local Time Complexity of Universal Constructions." In *Proceedings of the Seventeenth Symposium on Principles of Distributed Computing*, Pages 183–192, June 28 - July 2 1998. Puerto Vallarta, Mexico.

[51] Jensen, E. H., G. W. Hagensen, and J. M. Broughton. "A New Approach to Exclusive Data Access in Shared Memory Multiprocessors, TR UCRL-97663, Lawrence Livermore National Laboratory, November 1987..".

[52] Johnson, Theodore and Krishna Harathi. "Interruptible Critical Sections." Technical Report 94-007, Dept. of Computer and Information Science, University of Florida, 1994.

[53] Kung, H.T. and John Robinson. "On Optimistic Methods of Concurrency Control." *ACM Transactions On Database Systems*, 6(2):213–226, June 1981.

[54] LaMarca, Anthony. "A Performance Evaluation of Lock-Free Synchronization Protocols." In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, Pages 130–140, August 14–17 1994. Los Angeles, CA.

[55] Lampson, Butler W. "A note on the confinement problem." *Communications of the ACM*, 16(5):613–615, October 1973.

[56] Lampson, Butler W. "Hints for Computer System Design." In *Proceedings of the ACM Symposium on Operating System Principles*, Pages 33–48. ACM, December 1983. Reprinted in *IEEE Software* 1, 1 (Jan. 1984), pp 11–28.

[57] Lampson, Butler W. and David D. Redell. "Experiences with Processes and Monitors in Mesa." *Communications of the ACM*, 23(2):105–117, February 1980.

[58] Lanin, V. and D. Shasha. "Concurrent Set Manipulation Without Locking." In *Proceedings of the Seventh Symposium on Principles of Database Systems*, Pages 211–220, 1988.

[59] Legedza, Ulana and William E. Weihl. "Reducing Synchronization Overhead in Parallel Simulation." In *Proceedings of the 10th ACM Workshop on Parallel and Distributed Simulation*, 1996. Phil. PA.

[60] Linial, Nathan. "Distributive Graph Algorithms—Global Solutions from Local Data." In Chandra, Ashok K., editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, Pages 331–335, Los Angeles, California, 12–14 October 1987. IEEE, IEEE Computer Society Press.

[61] Lowell, David E. and Peter M. Chen. "Free transactions with Rio Vista." In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, Pages 92–101, St. Malo, France, October 1997. ACM SIGOPS. Published in ACM Operating Systems Review Vol.31, No.5, Dec. 1997.

[62] Massalin, Henry. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[63] Massalin, Henry and Calton Pu. "Threads and Input/Output in the Synthesis Kernel." In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Pages 191–201, Arizona, December 1989.

[64] Massalin, Henry and Carleton Pu. "A Lock-Free Multiprocessor OS Kernel." Technical Report CUCS–005–91, Columbia University, October 1991.

[65] McCabe, Thomas J. "A Complexity Measure." *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[66] Mellor-Crummey, J.M. and Michael L. Scott. "Algorithms for scalable Synchronization on Shared-Memory Multiprocessors." *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[67] Michael, Maged M. and Michael L. Scott. "Scalability of Atomic Primitives on Distributed Shared Memory Multiprocessors." Technical Report TR528, University of Rochester, Computer Science Department, July 1994.

[68] Michael, Maged M. and Michael L. Scott. "Correction of a Memory Management Method for Lock-Free Data Structures." Technical Report 599, Computer Science Department, University of Rochester, December 1995.

[69] Michael, Maged M. and Michael L. Scott. "Concurrent Update on Multiprogrammed Shared Memory Multiprocessors." Technical Report 614, Computer Science Department, University of Rochester, April 1996.

[70] Michael, Maged M. and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." In *Proceedings of the Fifteenth Symposium on Principles of Distributed Computing*, Pages 267–276, May 23-26 1996. Phila. PA.

[71] Mitchell, James G. and Jeremy Dion. "A Comparison of Two Network-Based File Servers." *Communications of the ACM*, 25(4):233–245, April 1982. (Originally in SOSP '81).

[72] Mogul, Jeffrey C. and K. K. Ramakrishnan. "Eliminating Receive Livelock in an Interrupt-driven Kernel." In USENIX Association, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, Pages 99–111, Berkeley, CA, USA, January 1996. USENIX.

[73] Moir, Mark. "Practical Implementations of Non-Blocking Synchronization Primitives." In *Proceedings of the Sixteenth Symposium on Principles of Distributed Computing*, Pages 219–228, August 1997. Santa Barbara, CA.

[74] Moir, Mark. "Transparent Support for Wait-Free Transactions." In *Proceedings of the Eleventh International Workshop on Distributed Algorithms*, Pages 305–319. Springer Verlag, September 24–26 1997. Saarbrucken, Germany, also published as *Lecture Notes in Computer Science Vol. 1320*.

[75] Motorola, Inc. *M68000 Family Programmer's Reference Manual*, 1989.

[76] Motorola, Inc. *PowerPC 601 RISC Mircroprocessor User's Manual*, 1993.

[77] Muir, Steven J. "Personal Communication." January 1999. University of Pennsylvania.

[78] Nir Shavit, E. Upfal and Asaph Zemach. "A Wait-Free Sorting Algorithm." In *Proceedings of the Sixteenth Symposium on Principles of Distributed Computing*, Pages 121–128, August 1997. Santa Barbara, CA.

[79] Pease, Marshall C., Robert E. Shostak, and Leslie Lamport. "Reaching Agreement in the presence of faults." *Journal of the ACM*, 27(2):228–234, April 1980.

[80] Plotkin, Serge A. "Sticky Bits and the Universality of Consensus." In *Proceedings of the Eighth Symposium on Principles of Distributed Computing*, Pages 159–175, August 1989. Edmonton, Alberta, Canada.

[81] Ramamurthy, Srikanth, Mark Moir, and James H. Anderson. "Real-Time Object Sharing with Minimal System Support (Extended Abstract)." In *Proceedings of the Fifteenth Symposium on Principles of Distributed Computing*, Pages 233–242, New York, USA, May 1996. ACM.

[82] Reed, David P. "Naming and Synchronization in a Decentralized Computer System." Technical Report MIT/LCS/TR-205, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1978.

[83] Reeves, Glenn E. "E-mail and newsgroup posting." December 15 1997. `Glenn.E.Reeves@jpl.nasa.gov`, full text and related links available from `http://www.research.microsoft.com/research/os/mbj/Mars_Pathfinder/`.

[84] R.K.Treiber. "Systems Programming: Coping with Parallelism." Technical Report RJ5118, IBM Almaden Research Center, April 1986.

[85] Ryan, Stein J. "Synchronization in Portable Device Drivers." *Operating System Review*, 33(1):18–25, January 1999.

[86] Saltzer, Jerome H., David P. Reed, and David D. Clark. "End-To-End Arguments in System Design." *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[87] Shavit, Nir and Dan Touitou. "Software Transactional Memory." In *Proceedings of the Fourteenth Symposium on Principles of Distributed Computing*, Pages 204–213, August 20–23 1995. Ottawa, Ont. Canada.

[88] Shavit, Nir and Asaph Zemach. "Diffracting Trees." *ACM Transactions on Computer Systems*, 14(2):385–428, November 1996.

[89] Shirley, Donna. "`http://quest.arc.nasa.gov/mars/ask/about-mars-path/How_much_we_are_paying_with_our_tax_dollars_for_Pathfinder.txt`." July 21 1997.

[90] Silberschatz, Abraham and James Peterson. *Operating System Concepts (Alternate Edition)*. Addison Wesley, 1988.

[91] Singh, Jaswinder Pal, Anoop Gupta, and Marc Levoy. "Parallel visualization algorithms: Performance and architectural implications." *IEEE Computer*, 27(7):45–55, July 1994.

[92] Sites, R. *DEC Alpha Architecture*. Digital Press, Burlington, Massachussets, 1992.

[93] Stone, J., H. Stone, P. Heidelbergher, and J. Turek. "Multiple Reservations and the Oklahoma Update." *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.

[94] Sturgis, Howard E. "A Postmortem for a Time Sharing System." Technical Report CSL-74-1, Xerox Palo Alto Research Center, January 1974. (Also PhD thesis, UCBerkeley, May 73.).

[95] Sturgis, Howard E., James G. Mitchell, and Jay E. Israel. "Issues in the Design of a Distributed File System." *ACM Operating Systems Review*, 14(3):55–69, July 1980.

[96] SUN Microsystems, Inc. *The SPARC Architecture Manual Version 9*.

[97] Symbolics, Inc. "Genera 7.4 Documentation Set." 1989.

[98] Torrellas, J., A. Gupta, and J. Hennessy. "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System." In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Pages 162–174, October 1992.

[99] Turek, J., D. Shasha, and S. Prakash. "Locking without blocking: Making Lock-Based Concurrent Data Structure Algorithms Non-Blocking." In *Proceedings of the 1992 Principles of Database Systems*, Pages 212–222, 1992.

[100] Valois, John. "Lock-Free Linked Lists Using Compare-and-Swap." In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Pages 214–222, August 20–23 1995. Ottawa, Ont. Canada.

[101] Wilner, David. "Invited Talk, *IEEE Real-Time Systems Syposium*." December 1997. (Summarized in RISKS 19.49 on December 9, 1997, by Mike Jones.).

[102] Woo, Steven Cameron, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations." In *Proceedings of the Twenty Second Annual International Symposium on Computer Architecture*, Pages 24–36, June 1995.

[103] Zelesko, Matt and David R. Cheriton. "Specializing Object Oriented RPC for Functionality and Performance." In *Proceedings 16th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society Press, May 27-30 1996.