Synchronization and Concurrency in User-level Software Systems

by

William N. Scherer III

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Michael L. Scott

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2006

To my loving wife, Blake;

without her support,

this would never have been possible.

Curriculum Vitae

William N. Scherer III was born in Baltimore, Maryland on September 3, 1971. He attended the Carleton College from 1989 to 1993, receiving a Bachelor of Arts in Computer Science in 1993. He came to the University of Rochester in the Fall of 2000 and began graduate studies in Computer Science. He pursued his research in synchronization under the direction of Professor Michael L. Scott. He received a Master of Science degree in Computer Science from the University of Rochester in 2002.

Acknowledgments

Words cannot express the debt I owe to my advisor, Michael L. Scott. He has stood by me, and had faith in me, even through periods of drought when I had no faith in myself and questioned whether I even belonged in a doctoral program. In addition to his patience, he has been a steady hand, guiding me gently towards productive areas of research; a colleague, providing invaluable insights for problems we've worked on; a role model, exhibiting an ideal of scholarship and teaching; and a friend.

I am also extremely grateful to Mark Moir, who supervised me while I was an intern in the Scalable Synchronization Research Group at Sun Microsystems. The influence of his hand on the research I have done ever since that time is unmistakable; his suggestions on the theoretical aspects of algorithms and their analysis have been particularly valuable. He was the main inspiration for the vast majority of the work on contention management in Chapter 4, for example. Finally, he has been a dangerous competitor at the pool table, keenly identifying the ideas that make some of my craziest shots actually work and turning them back upon me.

My committee members, Professors Chen Ding and Kai Shen, have been extremely helpful in pushing me to seek out practical applications of my work and to stay targeted on ideas that matter and are of practical significance. Their feedback has been invaluable in shaping this thesis.

I am particularly grateful to Doug Lea and Maurice Herlihy, both of whom have been unceasing pillars of support in my research and wonderful people to collaborate with.

I would like to thank all of the members of the computer science department for making this place a very enjoyable site at which to perform research. The systems group has been a great source of inspiration and feedback; my work would not be as good without their input. I also would like to thank JoMarie Carpenter and Matt Boutell for innumerable conversations and discussions on everything under the sun and then some. Chris Stewart, in his zaniness, has helped me keep a sense of humor about life and work. Virendra Marathe has been a great collaborator, an excellent source of ideas, and an insightful critic. Finally, I'd like to thank several people who have kindly indulged my addiction to the game of Crokinole: Isaac Green, David Ahn, Joel Tetreault, Chris Stewart, Virendra Marathe, Kirk Kelsey, and Arrvindh Shriraman.

A special thanks goes to Kim and Alex Dresner, who have never let minor things like oceans stand in the way of friendship.

Finally, this would not have been possible without support from my family. Edie and Liam have offered nothing short of total, unreserved love, no matter how many nights I don't get home until after they go to bed. My mother has been a life-long model of scientific curiosity; she and my father have always believed in me and pushed me to excel in academics. I am in awe of my sister Beth and her drive and focus when it comes to academia, medicine, and life in general. She has been a grounding force, helping me to remember the importance of staying connected to family by coming to visit me regularly. Last but not least, my wife Blake has been unconditionally supportive of my academic career, never once complaining when I'd stay out late for yet another night debugging some algorithm or other.

The material presented in this dissertation is based upon work supported by grants from DARPA, number F29601-00-K-0182; the Department of Education (GAANN), number P200A000306; the National Science Foundation, numbers E1A-0080124, CCR-0204344, and CNS-0411127; and by funds from Sun Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of above named organizations.

Abstract

Concurrency in user applications is on the rise. Modern computers contain multiple threads per core and multiple cores per chip; and users multitask routinely. Where traditional scientific and commercial computation assumes dedicated access to hardware, user applications must tolerate preemption.

To support legacy applications, most of which use lock-based mutual exclusion, we add timeout capability and preemption tolerance to scalable queue-based locks. This allows in-place updates to scale programs with fine-grained synchronization to large multiprogrammed systems.

Unfortunately, fine-grain locking is prone to deadlock, non-composability, priority inversion, convoying, and intolerance of thread failure, preemption, and even page faults. *Nonblocking algorithms* avoid these limitations by ensuring that the delay or failure of a thread never prevents the system as a whole from making forward progress. We broaden the range of known nonblocking algorithms by adapting linearizability theory to support partial operations. We define *dual data structures* as concurrent objects that may hold data and requests. We present lock-free dual stacks; dual queues; exchangers, wherein participants swap data pairwise; and synchronous queues, wherein consumers explicitly acknowledge handoffs from producers. Our exchangers and synchronous queues will appear in Java 6.

Like fine-grain locks, ad hoc nonblocking algorithms are too difficult for most programmers to write. While highly valuable in libraries, they are not a general approach to simple concurrency. Transactional memory allows mechanical translation from serial algorithms to high-performance concurrent implementations. Central to transactional systems is the need for *contention management*, which determines, when transactions conflict, which will continue, wait, or abort. We introduce several contention management policies, and evaluate their performance on a variety of benchmarks. We further demonstrate proportional-share prioritizing managers, and identify a candidate default policy that performs well with each benchmark. Finally, we perform a case study analyzing randomization in our "Karma" manager.

By furthering the state of the art in locks, nonblocking algorithms, and transactional memory, we expand the options available to programmers for preemption-tolerant synchronization in user-level software applications.

Table of Contents

List of Algorithms

List of Figures

1 Introduction

1.1 Motivation

Concurrency in user applications is on the rise. Multithreading and multiprogramming are becoming increasingly prevalent in user-level applications and end-user computational environments. GUI-based applications are frequently multithreaded, and concurrency is a standard part of undergraduate computer science curricula. Meanwhile, chip manufacturers have hit a power wall with processor speed, so they are increasingly turning to multiple threads per core and to multiple cores per chip to satisfy users' demands for greater computational power. To developers producing applications for such environments, synchronization and concurrency are no longer sub-fields for specialists; they are routine.

Of course, programming for multiprocessors has been extensively studied since they became widely available for scientific and commercial computing. Bounding destructive inter-processor contention and improving the scalability of applications and data structures in particular have been researched for decades.

However, much of this prior research is predicated on the assumption that a single application will have dedicated access to some subset of the computational hardware. This is generally true for scientific and commercial computing. By comparison, the multithreaded applications that end users are now running must provide good performance even in a highly multiprogrammed environment. Further, since end users typically run several applications concurrently, the well-known synchronization strategy of limiting thread creation to one per processor does not suffice to avoid preemption. Worse, the average programmer has only been taught one technique for dealing with synchronization: mutual exclusion via locking.

To update existing applications and to produce the next generation of high-performance concurrent systems, we thus have two choices: Fix locks to make them more resilient to preemption; or switch to other synchronization strategies. In the remainder of this Section, we characterize the shortcomings of lock-based synchronization and discuss alternatives. This dissertation focuses on extending the scope and availability of "fixed locks" and of lock alternatives.

1.1.1 Disadvantages of Locks

A *concurrent* object is a data object shared by multiple threads of control within a concurrent system. Coarse-grained locking implementations of concurrent objects, though easy to create, are not scalable: The loose granularity of locking inhibits concurrency by forcing strict serialization among operations. Alternatively, implementations built with fine-grained locks may be scalable; however, they suffer from several important drawbacks:

- **Deadlock:** Any cyclic blocking between threads in a system that requires acquisition of multiple locks prevents these threads from making any forward progress unless the system is forcibly restarted. Avoiding deadlock makes implementing concurrent data structures particularly hard for highly complex data structures such as red-black trees.

- **Non-composability:** Because of the risk of deadlock, programmers cannot compose multiple lock-synchronized operations to form compound operations without careful analysis of lock access patterns.

- **Priority inversion:** In a system with multiple run-time priorities for threads, if a high-priority thread H blocks while waiting for a low-priority thread L to finish using a lock, H is effectively downgraded in priority until L finishes using it. Worse, on a uniprocessor or overloaded system, scheduling H instead of L can lead to deadlock.

- **Convoying:** A common pathology in lock-based systems, convoying occurs when several threads all block waiting for the same lock in a common execution sequence. Once this happens, they tend to follow each other in lock-step from one lock to the next, unbalancing the system at each stop.

- **Fault intolerance:** Should any thread fail while holding a lock, mutual exclusion prevents any other thread from ever again acquiring it, absent some fancy repair mechanism built using application-specific knowledge.

- **Preemption intolerance:** Even absent permanent failure, a thread that is preempted — or even takes a page fault — while holding a lock prevents any other thread from utilizing resources guarded by the lock.

1.1.2 Alternatives to Locks

Due to the drawbacks associated with locks, the last two decades have seen increasing interest in *nonblocking* synchronization algorithms, in which the temporary or permanent failure of a thread can never prevent the system from making forward progress. These *ad hoc* nonblocking implementations of concurrent objects avoid the semantic problems of locks and can match or exceed the performance of fine grain locking, but

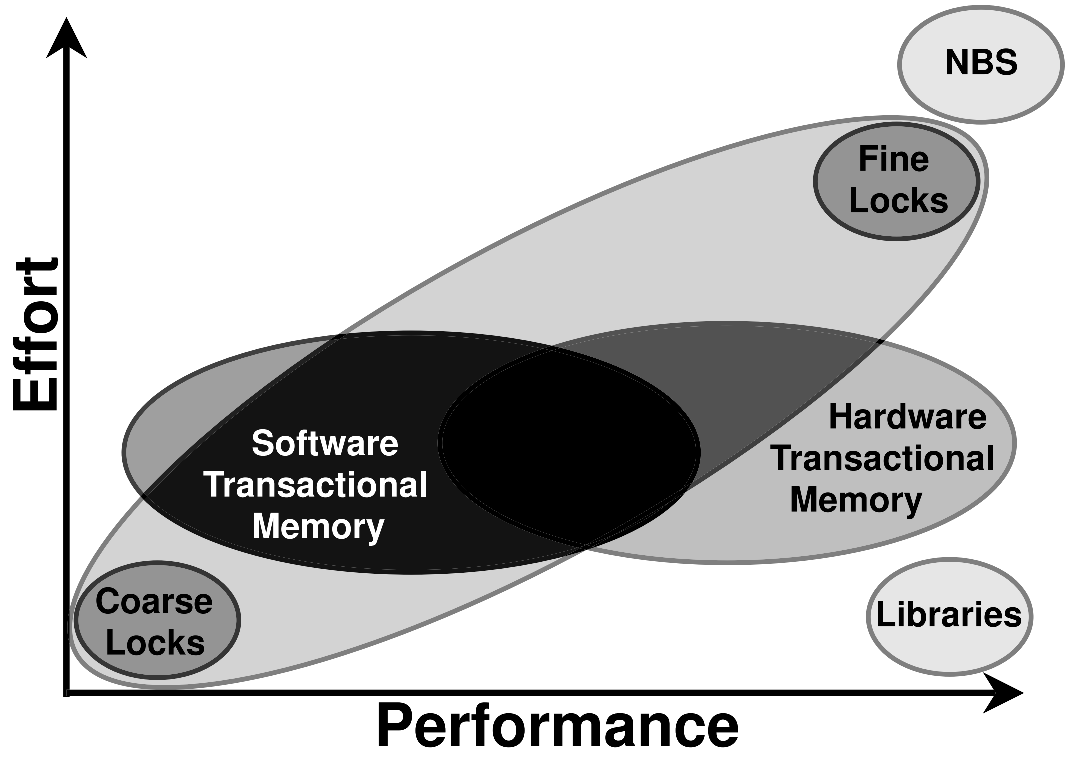


Figure 1.1: Synchronization from a user's perspective

they are at least as difficult to write. Fortunately for application programmers, high-performance fine-grained locking and nonblocking implementations of commonly-used data structures are often collected together into libraries such as NOBLE [SuT02] or the *java.util.concurrent.** [Lea05] collection. As shown in Figure 1.1 using such a library is usually fairly straightforward, though determining atomicity for composite operations remains problematic.

What can a programmer do, lacking time and/or capability to create either a fine-grained locking or an ad hoc nonblocking implementation of a data structure not available in a library? One option that has emerged in recent years is to use a general purpose *universal construction* that allows them to be created mechanically. The term *software transactional memory* (STM)[1] was coined by Shavit and Touitou [ShT97] as a software-only implementation of a hardware-based scheme proposed by Herlihy and Moss [HeM93]. Although early STM systems were primarily academic curiosities, more modern systems [Fra04; HaF03; HLM03b] have reduced runtime overheads sufficiently to outperform coarse-grained locks when several threads are active.

STM-based algorithms can generally be expected to be slower than either ad hoc nonblocking algorithms or highly-tuned fine-grained lock-based code. At the same time, they are as easy to use as coarse-grain locks: One simply brackets the code that needs to be atomic. In fact, STM systems allow correct sequential code to be converted, mechanically, into highly concurrent correct code. [2]

1.2 Background Information

1.2.1 Read-modify-write Atomic Instructions

In this work we make reference to several atomic operations. `Swap`(address, value) atomically writes a memory location and returns its original contents. `Compare_and_swap`(address, expected_value, new_value) atomically checks the contents of a memory location to see if it matches an expected value and, if so, replaces it with a new value. In either event it returns the original contents. We also use an alternative form, `compare_and_store`, that returns a boolean value indicating whether the comparison succeeded. `Fetch_and_increment`(address) atomically increments the contents of a memory location and returns the original contents.

[1]Although we hereafter tend to use these terms interchangeably, strictly speaking, a universal construction is a mechanical procedure for converting sequential code to concurrent code. An STM is a library that supports the result of applying a particular (highly straightforward) universal construction.

[2]This is a slight oversimplification: Open nesting of transactions, uncaught exceptions, and non-idempotent subroutine calls (such as for I/O) all complicate matters, though we do not discuss these further in this dissertation.

`Compare_and_swap` first appeared in the IBM 370 instruction set. `Swap` and `compare_and_swap` are provided by SPARC V9. Several recent processors, including Alpha, MIPS, and PowerPC, provide a pair of instructions, `load_linked` and `store_conditional`, that can be implemented naturally and efficiently as part of an invalidation-based cache-coherence protocol, and which provide the rough equivalent of `compare_and_swap`. Both `compare_and_swap` and `load_linked/store_conditional` are *universal* atomic operations, in the sense that they can be used without locking (but at the cost of some global spinning) to implement any other atomic operation [Her91]. `Fetch_and_increment`, together with a host of other atomic operations, is supported directly but comparatively inefficiently on the x86.

The Nonblocking Hierarchy

In the beginning of this chapter, we characterized nonblocking algorithms as those for which the pausing or failure of any thread cannot prevent another thread from getting work done. In fact, there are several variants of nonblocking synchronization, differentiated by the strength of progress guarantees that they offer:

- In a *wait-free* implementation, every contending thread is guaranteed to complete its operation within a bounded number of its own time steps [Her91].
- In a *lock-free* implementation, *some* contending thread is guaranteed to complete its operation within a bounded number of steps (from any thread's point of view) [Her91].
- In an *obstruction-free* implementation, a thread is guaranteed to complete its operation within a bounded number of steps in the absence of contention, i.e. if no other threads execute competing methods concurrently [HLM03a].
- In a *probabilistically obstruction-free* implementation, there exists some bound B such that for each B steps a thread completes in the absence of contention, it

is guaranteed to complete its operation with non-zero probability. By completing sufficient steps in the absence of contention, it thus completes its operation with probability one.

1.3 Statement of Thesis

The increasing importance and prevalence of concurrency in user-mode applications demands new techniques and new algorithms to support preemption-tolerant, high-performance computing. Improved scalable user-mode locks, extended libraries of nonblocking synchronization algorithms, and high-performance software transactional memories constitute practical options for improving performance and preemption tolerance in existing applications and for more simply constructing new ones.

1.4 Organization

The remainder of this dissertation is organized as follows. Chapter 2 moves toward "fixing" locks by addressing the problem of preemption tolerance in scalable user-mode locking. We extend standard queue-based locks to allow timeout, and then present a series of refinements in which we remove windows of vulnerability to preemption. First, we eliminate preemption vulnerability in the timeout protocol, and then we eliminate virtually all vulnerability in being queued behind a preempted waiter and enable recovery from threads preempted in their critical sections. Experimental evaluation demonstrates that our algorithms scale well on very large machines (including a 512-processor Cray T3E supercomputer) and that our preemption tolerance strategies yield very good utilization of queue-based locks even in a heavily multiprogrammed environment.

In Chapter 3, we focus on nonblocking synchronization in order to expand the body of standard implementations of concurrent objects that can be collected into libraries. We propose *dual data structures*, an adaptation of standard linearizability theory that supports concurrent objects with condition synchronization. We then develop a series of algorithms for nonblocking dual data structures, including queues, stacks, exchange channels, and synchronous queues. Empirical evaluation confirms that they provide considerable advantages over previously known implementations.

Chapter 4 is concerned with the case where high levels of concurrency are needed, but no appropriate library routines are available and the programmer lacks the time and/or capability to create a novel fine-grained locking or nonblocking solution. In such cases, we argue that *transactional memory* provides a viable alternative. We identify the problem of *contention management* as being central to extracting good performance from transactional memory systems and evaluate a variety of policies for managing conflict between transactional operations.

Finally, Chapter 5 summarizes our conclusions and suggests several avenues for future work in this area.

2 Lock-based Synchronization

2.1 Introduction

Of all the forms of synchronization in use today, by far the most widely employed is lock-based mutual exclusion. Spin locks in particular are widely used for mutual exclusion on shared-memory multiprocessors. Traditional TATAS spin locks (based on `test_and_set`) are vulnerable to memory and interconnect contention, and do not scale well to large machines. Queue-based spin locks [And90; Cra93a; GrT90; MLH94; MeS91] avoid contention by arranging for every waiting thread to spin on a separate, local flag in memory.

Traditionally spin locks have been used primarily for operating systems and for scientific computing on dedicated servers. This is not an accident: Spin locks do not handle preemption well. If the thread that holds a lock is suspended before releasing it, any CPU time given to waiting threads will be wasted on fruitless spinning.

More recently, however, developers have increasingly turned to spins locks for user-level applications. Central to this use is the ability to time out and return (unsuccessfully) from a lock acquisition attempt that takes "too long":

1. A thread in a soft real-time application may need to bound the time it spends waiting for a lock. If the timeout expires, the thread can choose to announce an error or to pursue an alternative code path that does not require the lock.

2. If a thread is preempted while holding a lock, timeout allows other threads waiting for the lock to give up, yield the processor, and try again when rescheduled. (This assumes there are enough non-waiting threads to keep the processor busy.)

3. In a parallel database system, timeout provides a viable strategy for deadlock recovery. A thread that waits "too long" for a lock can assume that deadlock has occurred, abort the current transaction, yield the processor, and retry when rescheduled.

Timeout-capable spin locks are sometimes known as *try locks*. We are aware of commercially significant signal processing applications that use try locks for reason (1) above, and parallel database servers that use them for reasons (2) and (3). In the latter case, timeout may be *the* deciding factor in making spin locks acceptable for user-level code.

Unfortunately, while timeout is trivial in a test-and-set lock—threads are mutually anonymous, and can simply give up and return—it is far from trivial in a queue based lock. To preserve the integrity of the queue data structure, a timed-out thread must arrange for its neighbors to find each other, even if they, too, are in the process of timing out. And since the goal is to build a lock, the mechanism used to modify the queue must not itself require locks.

The most obvious solution, suggested by Craig in a 1993 technical report [Cra93a], is for a timed-out thread to simply mark its node as abandoned, and for the thread that releases a lock to skip over and reclaim abandoned nodes. While simple, this solution has the disadvantage of introducing overhead on the program's critical path. In the worst case, the number of nodes that must be passed over to find a new lock holder is linear in the number of threads—higher if a timed-out thread may attempt to acquire a lock again before the point at which it would have succeeded if it had stayed in line. We therefore take the position that abandoned nodes should, whenever possible, be removed from the queue before they would be seen by the lock releaser.

Sections 2.3 presents new queue-based try-locks (MCS-try and CLH-try) that extend the MCS lock of Mellor-Crummey and Scott [MeS91] and the CLH lock of Craig [Cra93a] and Landin and Hagersten [MLH94]. These new locks provides tight space bounds and good performance in practice. Subsequent to their development, Scorr published variants (MCS-NB-try and CLH-NB-try) that provide striuct adherence to requested time bounds (timeout is nonblocking) and good space usage in practice [Sco02]. Section 2.4 presents an alternative realization of the MCS-NB-try lock that requires only atomic `swap` and features a simpler design. Depending on the machine architecture and the relative importance of time and space guarantees, each of these locks (MCS-try, CLH-try, [the new] MCS-NB-try, and CLH-NMB-try) are viable candidates for soft real-time or database applications.

2.1.1 Preemption Tolerance

With the increasing use of multiprogrammed multiprocessors for complex server applications, parallel programs cannot in general count on the dedicated use of any specific number of processors: Spawning one thread per processor does not suffice to avoid preemption in a multiprogrammed environment. Scheduler-based locks avoid wasting time on fruitless spinning if a lock holder is preempted, but the high cost of context switches can significantly reduce performance in the common, no-preemption case. So-called "spin-then-block" locks strike a compromise by spinning for a while and then blocking if unsuccessful [KLM91; Ous82]. Try locks serve a similar purpose, allowing an individual thread to move on to other work.

But what if there is no more work? Although timeout or blocking avoids wasting time waiting for a preempted peer, it does nothing to improve system-wide throughput if the lock is squarely on the application's critical path.

Moreover, in a queue-based lock, a thread preempted while waiting in the queue will block others once it reaches the head; strict FIFO ordering is generally a disadvantage in the face of preemption. And in try locks, any timeout protocol that requires explicit handshaking among neighboring threads will block a timed-out thread if its neighbors are not active.

The problem of preemption in critical sections has received considerable attention over the years. Alternative strategies include avoidance [ELS88; FRK02; KWS97; MSL91; Ous82], recovery [ABL92; Bla90; TaS97; ZhN04], and tolerance [KLM91; Ous82]. The latter approach is appealing for commercial applications because it does not require modification of the kernel interface: If a thread waits "too long" for a lock, it assumes that the lock holder has been preempted. It abandons its attempt, yields the processor to another thread (assuming there are plenty) and tries again at a later time. In database systems timeout serves the dual purpose of deadlock recovery and preemption tolerance; it also potentially allows a preempted lock holder to be rescheduled.

In a different vein, *nonblocking* algorithms avoid preemption problems by eliminating the use of locks [Her91]. Unfortunately, while excellent nonblocking implementations exist for many important data structures (see Chapter 3), general-purpose mechanisms remain elusive. Several groups (including our own) are working on this topic [HaF03; HLM03b; MSS04; ScS04a; MSS05], but it still seems unlikely that nonblocking synchronization will displace locks entirely soon.

Assuming, then, that locks will remain important, and that many systems will not provide an OS-level solution, how can we hope to leverage the fairness and scalability of queue-based spin locks in multithreaded user-level programs?

We answer this question with a pair of queue-based try locks that combine fair and scalable performance with good preemption tolerance: the MCS time-published lock

(MCS-TP) and the CLH time-published (CLH-TP) lock. In this context, we use the term *time-published* to mean that contending threads periodically write their wall clock time to shared memory in order to be able to estimate each other's runtime states. Given a low-overhead hardware timer with bounded skew across processors and a memory bus that handles requests in bounded time (both of which are typically available in modern multiprocessors), we can guess with high accuracy that another thread is preempted if the current system time exceeds the thread's latest timestamp by some appropriate threshold. We now have the ability to selectively pass a lock only to an active thread. Although this doesn't solve the preemption problem entirely (a thread can be preempted while holding the lock, and our heuristic suffers from a race condition in which we read a value that has just been written by a thread immediately before it was preempted), experimental results (Section 2.6) confirm that our approach suffices to make the locks *preemption tolerant*: free, in practice, from virtually all preemption-induced performance loss.

2.1.2 Subsequent Sections

The remainder of this chapter is organized as follows: After tracing a history of lock-based synchronization in Section 2.2, we describe queue-based try locks that support timeout (Section 2.3), that time out in nonblocking fashion (Section 2.4), and that are highly preemption tolerant (Section 2.5). In Section 2.6 we present performance results on several large machines, including a 56-processor Sun Wildfire, a 144-processor Sun E25K, a 32-processor IBM p690, and a 512-processor Cray T3E. In experiments with more threads than processors, we also demonstrate clearly the performance advantage of nonblocking timeout. We return in Section 2.7 to a summary of conclusions and directions for future work.

2.2 Literature Review

Locks are by far, the most prevalent form of synchronization today. A *lock* is an implementation of an algorithm that solves the well-known mutual exclusion problem. Although not all practitioners know the implementation of any particular lock, understanding when and how to use one is standard fare for undergraduate computer science education.

There are two general categories of locks: scheduler-based and spin locks. Scheduler-based locks rely on the operating system's (or runtime system's) context switcher to manage lock handoff. Although this prevents performance loss from scheduling a blocked thread, it induces significant overhead, particularly if the operating system is involved in context switches. Consequently, we do not discuss these locks further.

Spin locks are the main alternative to scheduler-based locks. They make use of protocols that implement mutual exclusion. Although these protocols can become somewhat complex, the availability of powerful read-modify-write atomic operations simplifies things somewhat. Also, mutual exclusion has been studied for a very long time, so the design of such protocols is well understood.

In the remainder of this section, we survey historical and modern lock-based synchronization.

2.2.1 Early Locks and Terminology

The mutual exclusion problem may be defined as the construction of a pair of concurrent protocols that we will name *acquire* and *release* and which guarantee the property that should a thread (or process) t_0 complete the *acquire* protocol, no other thread will do so until t_0 has started to execute the *release* protocol. The code that t_0 executes between completion of the *acquire* protocol and the beginning of the *release* protocol is referred to as a *critical section*. In essence, then, mutual exclusion is a general technique for ensuring in multiprogrammed and multiprocessor environments that only one thread is executing the code in a critical section. This in turn is a basic building block that can be used for all manner of concurrent systems. Throughout this paper, we will refer to synchronization algorithms that solve the mutual exclusion problem as locks, owing to their ability to lock out all but one thread.

In this section, we highlight some of the mutual exclusion algorithms that are of particular historical significance. This is typically due to their being the first algorithm to provide guarantees of some important theoretical property that we will discuss later in this chapter in conjunction with try locks. Readers interested in a more detailed review of early synchronization are referred to Raynal's analysis [Ray86] for locks up to about 1986 and to Anderson and Kim [AnK01] for more recent algorithms.

The mutual exclusion problem was originally described 40 years ago by Dijkstra [Dij65]. In this ground-breaking paper, he extended an algorithm due to Dekker from support for precisely 2 threads to support for an arbitrary number of threads. In addition to being the first solution to the mutual exclusion problem, Dijkstra's solution also guarantees a property known as *progress*: If a lock is not currently held and any arbitrary group of k threads attempts to acquire it, one of the k is guaranteed to acquire the lock within a finite number of steps of execution. (Mutual exclusion in the absence of this requirement is trivially solved by a protocol in which no thread ever acquires the lock.)

Less than a year after the first publication of Dijkstra's lock, Knuth published a new solution to the mutual exclusion problem [Knu66]. Knuth's lock improves upon Dijkstra's earlier solution by guaranteeing *fairness*. The fairness property guarantees that every thread that attempts to acquire a lock eventually does so (though fairness does not require any bound on the amount of time that the acquiring thread must wait). The fairness property is also known as *starvation freedom*, after the Dining Philosophers problem [Dij72].

Lamport's Bakery algorithm [Lam74] strengthened the fairness property to *FIFO* fairness. The FIFO fairness property holds that threads acquire the lock (complete the acquisition protocol) in the same order that they begin the acquisition attempt. Although Lamport's algorithm does not provide perfect FIFO ordering, it is FIFO after a *wait-free* prologue. (A wait-free protocol is one in which every thread is guaranteed to complete within a bounded (finite) number of steps.)

Especially because early lock algorithms were relatively complex, the Peterson 2P lock [Pet81] is notable for requiring just three lines of code! Further, although the algorithm is presented for two threads, it extends easily to arbitrary numbers of threads [Lyn96]: Nested "copies" of a Peterson 2P lock each hold back one thread, so $n - 1$ instances are sufficient to provide mutual exclusion for n threads.

A major concern for early mutual exclusion research was the number of shared variables required for mutual exclusion and the number of values required of each shared variable. Burns [Bur81] presents a new lock typical of such efforts to minimize the required numbers. His algorithm requires $n + 1$ variables for n threads, n of which are strictly binary and one which has n values.

Although results such as the previous one are theoretically important, the algorithms that represent this line of research are typically of little immediate commercial use: Modern hardware has access to advanced atomic primitives that greatly simplify algorithmic complexity. By way of comparison, Lamport published in 1987 a *fast* algorithm [Lam87] that was considerably more practical than any of its predecessors. By *fast*, we mean an algorithm that requires constant ($O(1)$) time for the acquisition protocol in the absence of contention, though the space overhead is still linear. This is important because it is widely held that in well-tuned systems, contention is rare.

All of the algorithms discussed so far in this section are based on simple reads and writes to memory. When we consider advanced atomic read-modify-write instructions, additional locks become possible. For example, repeating `test_and_set` (TAS) instructions until a thread itself does the "set" is one simple lock that is enabled with atomic instructions. Similarly, one can repeat atomic `swap` instructions with a one un-

```
typedef unsigned long bool;
typedef volatile bool tatas_lock;

void tatas_acquire(tatas_lock *L) {
    if (test_and_set(L)) {
        int b = BACKOFF_BASE;
        do {
            for (i = rand() % b; i; i--); // back off
            b = min(b * BACKOFF_FACTOR, BACKOFF_CAP);
            if (*L) continue;             // spin
        } while (test_and_set(L));
    }
}

void tatas_release(tatas_lock *L) {
    *L = 0;
}
```

Listing 2.1: The test-and-`test_and_set` (TATAS) lock with exponential backoff. Parameters `BACKOFF_BASE`, `BACKOFF_FACTOR`, and `BACKOFF_CAP` must be tuned by trial and error for each individual machine architecture.

til the value swapped out is a zero to achieve a similar lock (such a protocol is described in, but probably not original to Raynal's analysis of early locks [Ray86]).

The simple `test_and_set`-based lock has very poor scalability because every TAS operation requires operations in memory and the bus interconnect between processors. An enhanced version of the lock named TATAS (for test-and-`test_and_set`) adds a read spin on the lock between TAS attempts; this reduces wasteful traffic. Anderson proposes a further improvement (shown in Listing 2.1) to the TATAS lock that adds exponential backoff to the acquisition protocol [And90]. Backoff prevents a large spike of message traffic in the processor interconnect by effectively "staggering" threads so that not all threads see that the lock is available at the same moment when it is released.

As we shall see in Section 2.6, exponential backoff extends the scalability of TATAS locks out through about 16 to 20 processors. Beyond this limit, however, they degrade rapidly. Recently, Radović and Hagersten proposed an extension [RaH03] for non-uniform communication architecture (NUCA) hardware. In this scheme, acquiring a lock from a "close" processor costs the same as in other `test_and_set`-based locks.

To acquire the lock from a more distant processor, however, requires an additional atomic operation. Coupled with different sets of backoff constants for use when the lock holder is close or distant, this results in a scheme where locks are preferentially transferred to nearby processors. In highly NUCA hardware, this can considerably reduce lock handoff overhead and improve utilization.

2.2.2 A History of Queue-Based Locks

An important paradigm for implementation of mutual exclusion may be found in queue-based locks. These locks all share a common theme: A thread waits in line until it reaches the head of the queue; once there, it has acquired the lock. Most of the research completed on queue-based locks has been very recent compared to that spent on the general mutual exclusion problem. However, the first description of a queue-based approach dates back to 1966, in one of the first papers on mutual exclusion. Specifically, in his discussion of the concept of fairness [Knu66], Knuth observes that FIFO fairness can be achieved if one has access to (using current terminology) atomic implementations for adding to a queue, retrieving the head of a queue, and removing the head of a queue. His algorithm (in Listing 2.2) is fundamentally unchanged over the years; it differs only in implementation details from, say, the MCS lock [MeS91]. (Typically, however, modern implementations combine `add-to-queue()` and spinning on calls to `head-of-queue()` in a single `acquire()` procedure.)

In the remainder of this subsection, we discuss a selection of queue-based locks that follow in Knuth's footsteps.

Hardware Approaches

Before the development of software queue-based locks, synchronization in parallel programs was considered a performance nadir to be avoided at all costs. Some studies found that standard TATAS locks added very high levels of contention in memory

```
begin:
  processor i;
  queue q;
L0:
  add-to-queue(q, i);
L1:
  if (i != head-of-queue(q)) then goto L1;
  /* critical section */
  remove-from-queue(i);
  /* remainder */
  goto L0;
end.
```

Listing 2.2: Knuth's queue-based lock

interconnects, adding up to 49% messaging overhead [PfN85]. This is due to a high frequency of cache invalidation as each of several threads repeatedly attempts to access a common memory address.

Because of this overhead, many multiprocessors were designed to provide direct hardware support for locking. Several architectures had built-in support for queue-based locking. For example, the BBN Butterfly's [BBN87] firmware included "dual queues" of thread-specific "events" that worked in conjunction with the operating system to automatically reschedule threads once at the head of the queue.

Another hardware-based scheme appeared in Kendall Square Research's KSR-1 multiprocessor [Ken92]. This scheme exploited the unidirectional ring topology of the hardware in order to implement a token ring-like locking scheme that, unfortunately, never worked quite right and was outperformed by an ordinary TATAS lock.

The Stanford DASH project [LLG92] provided hardware-assisted queue-based spin locks that again exploited the machine's clustered topology. In this scheme, rather than invalidating all caches' copies of lock variables when the lock is released, an extension to the cache coherence protocol [LLG90] allows the lock to be passed to a random waiting cluster. This is accomplished by only invalidating one cluster's cached copies of the lock variables; [1] other clusters never notice that the lock was ever freed.

A more hardware-agnostic approach, the `QOSB` instruction [GVW89] was designed to allow *local spinning*: spinning in which each thread spins on a separate cache line. Now, by calling `QOSB` on a synchronization address, a thread could add itself to a queue for the address. The `test_and_set` operation is modified in this scheme to additionally fail if there is a queue in place for the target address and the issuing thread is not at the head of the queue. Finally, a special `unset` instruction resets the flag bit to zero in addition to dequeuing the current thread (if it is queued). By virtue of these modifications, locking-induced remote cache accesses are limited to a small constant.

A small modification to the interface for `QOSB` was later added. At the same time, the instruction was renamed to `QOLB` [KBG97]. In the new interface, a single `EnQOLB` instruction enqueues the calling thread, but repeated `EnQOLB` instructions can be used to spin until the current thread gets the lock. Also, an `EnQOLB` instruction can be inserted ahead of time in the instruction stream to allow pre-fetching of the cache-line for the word of interest. Finally, a `DeQOLB` instruction releases the lock and enables the next waiting thread.

Empirical tests performed by Kägi et al. show that the `QOLB` instructions allow for highly scalable, FIFO waiting, significantly outperforming test-and-`test_and_set` locks. (It also outperforms the MCS lock, described in the next section, by roughly a factor of two when there is moderate contention for the lock; this is because protected data can be collocated in the same cache line as the lock that guards it.)

In part due to these performance benefits, the `QOLB` primitive remains in use today in such multiprocessors as the HP/Convex Exemplar and Sequent/IBM NUMA-Q. A

[1]Strictly speaking, this approach reduces fairness by abandoning the usual FIFO semantics of queue-based locks; however, the expected gains in throughput from increased lock utilization outweigh any risk of starvation.

current description for it in may be found in the Scalable Coherent Interface (SCI) standard [P1590].

Early Queue-based Locks

The first published (and fully implemented) software queue-based lock that we are aware of is due to Anderson [And90]. As with the hardware solutions embodied in `QOLB`, Anderson's lock scales linearly for lock acquisition time under heavy contention (beyond a few threads). Unlike the hardware approaches, however, Anderson's lock has high overhead in the low-contention case.

Another early queue-based lock is due to Graunke and Thakkar [GrT90]. Their lock is very similar to Anderson's, differing mainly in that where Anderson's uses `fetch_and_increment`, Graunke and Thakkar's lock uses `swap`. Their lock (as originally published) also requires brief access to a hardware mutual exclusion algorithm while enqueuing a thread, though the hardware lock is released before the spin. Performance characteristics for Graunke and Thakkar's lock are similar to those for Anderson's lock.

Both Anderson's and Graunke and Thakkar's locks use an array-based queue. This has three implications. First, for t threads and n locks that coexist in a system, the space overhead is $O(n \times t)$. Second, since space must be reserved at the time the lock is created, there is little flexibility to adapt to varying numbers of threads: An expected maximum must be used. This limits the locks' usability for user-level synchronization in multiprogrammed systems. Finally, because space in arrays is typically allocated contiguously, on machines with cache line sizes of more than one word it is (at least theoretically) possible for the spin in these locks to still exhibit cache-invalidation-based memory interconnect contention if one does not pad the array elements.

Inspired by `QOSB`, Mellor-Crummey and Scott's MCS lock [MeS91] (shown in Listing 2.3) addresses many of these limitations by using a linked list for its queue. Consequently, it has space overhead of $O(n + t)$. Also, because the individual queue

```
typedef struct mcs_qnode {
    volatile bool waiting;
    volatile struct mcs_qnode *volatile next;
} mcs_qnode;

typedef volatile mcs_qnode *mcs_qnode_ptr;
typedef mcs_qnode_ptr mcs_lock; // initialized to nil

void mcs_acquire(mcs_lock *L, mcs_qnode_ptr I) {
    I->next = nil;
    mcs_qnode_ptr pred = swap(L, I);
    if (pred == nil) return;    // lock was free
    I->waiting = true;          // word on which to spin
    pred->next = I;             // make pred point to me
    while (I->waiting);         // spin
}

void mcs_release(mcs_lock *L, mcs_qnode_ptr I) {
    mcs_qnode_ptr succ;
    if (!(succ = I->next)) {    // I seem to have no succ.
        // try to fix global pointer
        if (compare_and_store(L, I, nil)) return;
        do {
            succ = I->next;
        } while (!succ);        // wait for successor
    }
    succ->waiting = false;
}
```

Listing 2.3: The MCS queue-based spin lock. Parameter `I` points to a qnode record allocated (in an enclosing scope) in shared memory locally-accessible to the invoking processor.

nodes are allocated separately, the chance of them occupying adjacent memory (and thus the same cache line) is greatly reduced. This benefit comes at a price, however: The MCS lock, like Graunke and Thakkar's, requires access to a `compare_and_swap` instruction. The performance characteristics of the MCS lock are similar to those of Anderson's and Graunke and Thakkar's locks on machines that cache remote memory locations. However, on non-cache-coherent non-uniform memory access (NCC-NUMA) hardware, MCS still performs well because it arranges for each thread to spin on a node in its own memory space. By comparison, the other locks do not share this property; their performance suffers in such environments.

Doing More (Or at Least, Almost as Much) with Less

The requirement of such exotic [2] instructions as `compare_and_swap` for the MCS lock has provoked many people to ask whether simpler, or at least fewer, atomic instructions could be used. Recognizing this need, Mellor-Crummey and Scott also present a variant that requires only `swap`, but this variant lock admits the possibility of starvation [Hua99].

Another reduced-primitives lock, requiring only `swap`, is due to Landin and Hagersten [MLH94], and independently, Craig [Cra93a]. This lock (shown in Listing 2.4) uses a doubly-indirect pointer, and the node that a thread leaves the queue with is actually different than the one it entered with. As a result, the lock is of limited use for NCC-NUMA machines: The node on which a thread spins is not guaranteed to be local to the thread's address space. However, by adding an extra level of indirection (as per Listing 2.5), it can be made to work perform well in such environments.

The M lock, due to Magnussen [MLH94], adds considerable complexity (a third more code) compared to the MCS lock in order to save a single read access to a global variable. Huang [Hua99] also gives a queue-based lock — both `swap`-only and with `compare_and_swap` —that is optimal in terms of memory contention and remote references. Finally, Krieger et al. also require just `swap` in their scalable fast reader-writer lock [KSU93] (one which allows multiple concurrent readers, but only a single writer).

Adding Properties to Locks

Besides reducing the complexity of atomic instructions needed, another research path for queue-based spin locks has been to add additional properties, based on shortcomings of earlier locks. For example, they tend to perform badly in environments

[2]for the day, though `compare_and_swap` (or its rough equivalent, `load_linked/store_conditional`) are even today not available for all multiprocessors

```
typedef struct clh_qnode {
    volatile bool waiting;
    volatile struct clh_qnode *volatile prev;
} clh_qnode;

typedef volatile clh_qnode *clh_qnode_ptr;
typedef clh_qnode_ptr clh_lock;
    // initialized to point to an unowned qnode

void clh_acquire(clh_lock *L, clh_qnode_ptr I) {
    I->waiting = true;
    clh_qnode_ptr pred = I->prev = swap(L, I);
    while (pred->waiting);      // spin
}

void clh_release(clh_qnode_ptr *I) {
    clh_qnode_ptr pred = (*I)->prev;
    (*I)->waiting = false;
    *I = pred;                  // take pred's qnode
}
```

Listing 2.4: The CLH queue-based spin lock. Parameter `I` points to qnode record or, in `clh_release`, to a pointer to a qnode record. The qnode "belongs" to the calling thread, but may be in main memory anywhere in the system, and will generally change identity as a result of releasing the lock.

```
typedef struct clh_numa_qnode {
    volatile bool *w_ptr;
    volatile struct clh_qnode *volatile prev;
} clh_numa_qnode;

typedef volatile clh_numa_qnode *clh_numa_qnode_ptr;
typedef clh_numa_qnode_ptr clh_numa_lock;
    // initialized to point to an unowned qnode

const bool *granted = 0x1;

void clh_numa_acquire(clh_numa_lock *L,
                      clh_numa_qnode_ptr I) {
    volatile bool waiting = true;
    I->w_ptr = nil;
    clh_numa_qnode_ptr pred = I->prev = swap(L, I);
    volatile bool *p = swap(&pred->w_ptr, &waiting);
    if (p == granted) return;
    while (waiting);            // spin
}

void clh_numa_release(clh_numa_qnode_ptr *I) {
    clh_numa_qnode_ptr pred = (*I)->prev;
    volatile bool *p = swap(&((*I)->w_ptr), granted);
    if (p) *p = false;
    *I = pred;                  // take pred's qnode
}
```

Listing 2.5: Alternative (NUMA) version of the CLH lock, with an extra level of indirection to avoid remote spinning on a non-cache-coherent machine.

where the total number of threads exceeds the number of processors. Here, the busy-spinning behavior of the lock becomes a detriment: Passing the lock to a swapped-out thread will cause all waiting threads to spin uselessly until that thread is rescheduled.

Kontothanassis, Wisniewski, and Scott [WKS94; KWS97; WKS95] address this problem in their Smart-Q lock by exploiting a special kernel interface that allows a lock holder to determine whether a candidate successor is currently swapped out. If the successor is in fact swapped out, the lock holder flags its acquisition attempt as failed and moves on to the next candidate. The former successor then re-enters the queue. Hence, by relaxing absolute FIFO ordering of lock acquisitions (and adding a very slim possibility for starvation), the Smart-Q lock eliminates much preemption-induced overhead. Their handshaking lock uses a standard kernel interface, but slows down the common case.

Another property that has been added to queue-based locks is support for timeout. Here, a thread can specify a maximum amount of time that it is willing to wait in order to acquire the lock. An early mention of this property in relation to queue-based locks and some detailed comments on the implementation of a timeout-based lock are due to Craig [Cra93b]. The first published implementations of a timeout-supporting queue-based lock are the MCS try and CLH try locks [ScS01].

MCS try is vulnerable to preemption issues related to those the original MCS algorithm suffers from. In particular, when a thread wishes to leave the queue, it needs to handshake with both of its neighbors in order to leave safely. Should a neighbor be swapped out, this protocol again results in useless spinning. In order to address this problem, Scott [Sco02] derives variants of the MCS try and CLH try locks that time out in nonblocking fashion. Again, this comes at a cost: The space overhead for the locks is no longer bounded (though in practice it is unlikely that worse than $O(n+p^2)$ would ever be observed). By using a priority queue, Jayanti [Jay03] establishes a bound on worst-case space overhead; however, his algorithm incurs far too much overhead for practical use.

A final property that has been added to queue-based locks is the ability to recover from a crashed thread. By considering a crashed thread as one that is swapped out indefinitely, one can see that all of the issues we have discussed thus far in this section apply doubly in the case of thread death: A swapped out thread eventually comes back, so preemption is "just" a time delay. To quote a popular environmentalist slogan, "extinction is forever".

In order to add support for recovery (the ability to go on despite a crashed thread), Bohannon [BLS96] adds a separate cleanup procedure invoked periodically by a thread guaranteed to never wait for the lock. Although cleanup overhead is high (all lock activity is blocked while checking to see if cleanup is necessary), the frequency with which cleanups occur can be tuned to amortize this cost to arbitrarily low amounts. Nonetheless, this facility must be used carefully; it is difficult to determine that a thread is truly dead, and careful analysis of application-specific knowledge is generally necessary to restore invariants when a thread dies halfway through an operation.

Other Data Structures

Besides the locks previously mentioned, a few other variants make use of non-queue data structures, yet are still derived from the MCS lock. The first of these locks is due to Lim and Agarwal [LiA94]. Noticing that the MCS lock is outperformed by a simple test-and-`test_and_set` lock in the absence of contention, they set out to get the best of both worlds by having a lock that uses different protocols under different conditions. For example they might create a lock that switches between test-and-`test_and_set` locks and MCS depending on whether contention is low or high. In order to effect a clean transfer, they introduce a new data structure, the *consensus object*, that can only be accessed by threads that hold the lock. Consensus objects contain boolean flags for whether they are currently valid, and one consensus object is globally shared for each protocol, for each instance of the reactive lock. Changing protocols is a simple matter of invalidating the flag for the old protocol and validating the flag for the new one;

the trick is deciding *when* to switch. It turns out that fairly simple decision protocols are sufficient to achieve overall performance comparable to the low-contention performance of test-and-`test_and_set` locks and the high-contention performance of the MCS lock.

Another instance of a changed data structure may be found in the work of Fu and Tzeng [Fu97; HuS98], who use a circularly linked list as the foundation for a queue-based lock. In their analysis of the MCS lock, they observe that the base lock pointer itself is a single place in memory that all threads access, hence a hot spot. By using a circularly linked list, they spread out the initial memory accesses across multiple memory locations. Notably, their lock outperforms the MCS lock in all instances in their tests; however, these tests were performed on simulated hardware.

The final alternate data structure we consider is the priority queue. Locks of this form relax absolute FIFO fairness to level-specific FIFO fairness within multiple service levels. Generally, a thread in a higher priority level is allowed to acquire the lock ahead of every thread at lower levels; to avoid starvation, some authors bound the number of times the longest-waiting thread at a lower priority level can be passed over to some constant factor k per priority level.

The first published priority queue-based lock, which is based on MCS with a doubly-linked list, is due to Markatos and LeBlanc [Mar91]. This lock has a serious drawback, however: Priority queue maintenance is performed at release time, where it adds to the wait time for all threads. The PR-lock, due to Johnson and Harathi [JoH97], corrects this problem by moving lock maintenance procedures into the enqueue operation (thereby folding maintenance costs into wait time). This yields a large performance gain in lock acquisition, and converts from linear to constant time overhead in lock release. Another priority lock due to Craig [Cra93a] uses `swap` instead of `compare_and_swap` and also allows nesting locks with only one lock record.

A standard problem with prioritized threads and waiting is known as *priority inversion*. Essentially, when a high-priority thread is forced to wait for a lower-priority thread that has the lock it wants, it needs to wait at the lower priority. A standard way to solve this problem is to bump up the lower thread's priority to match that of the waiting thread. Wang, Takada, and Sakamura [WTS96] give a pair of algorithms that use exactly this approach to solve the priority inversion problem. The main difference between the two is that the second algorithm is designed to avoid non-local spins, which result in heavy interconnect traffic on hardware that does not cache remote memory locations.

2.2.3 Lock-based Synchronization in Java

Because monitors and `synchronized` methods are particularly prevalent in Java, many groups have sought to reduce the cost of locks in this environment. For example, Bacon et al. [BKM98] propose an "inflatable" lock that allows a thread to "own" an object in the absence of contention and cheaply lock or unlock it. When contention occurs, the lock expands into a queue to support multiple waiters. Once expanded, however, the lock remains expanded forever. Meta-locks [ADG99] use a secondary lock to explicitly manage synchronization data, thereby avoiding some space overhead. Relaxed locks [Dic01] address the space concern by speculatively deflating the lock when the contention appears to end; if contention has not actually ended, then the overhead of inflation recurs.

A different approach to reducing the cost of locking in Java is to remove locks that are never contended in the first place. If compile-time program analysis can determine that a particular lock will never be contended for, the overhead of synchronization can be deleted altogether. Many groups have applied this approach [ACS99; BoH99; DiR99; FKR00; GRS00; Ruf00].

```
// return value indicates whether lock was acquired
bool tatas_try_acquire(tatas_lock *L, hrtime_r T) {
    if (tas(L)) {
        hrtime_t start = gethrtime();
        int i, b = BACKOFF_BASE;
        do {
            if (gethrtime() - start > T) return false;
            for (i = rand() % b; i; i--);  // back off
            b = min(b * BACKOFF_FACTOR, BACKOFF_CAP);
            if (*L) continue;             // spin
        } while (tas(L));
    }
}
```

Listing 2.6: The standard TATAS-try lock. Type definitions and `release` code are the same as in Figure 2.1.

2.3 Queue-based Locks with Timeout

As noted in Section 2.1, a thread may wish to bound the time it may wait for a lock in order to accommodate soft real-time constraints, avoid waiting for a preempted peer, or recover from transaction deadlock. Such a bound is easy to achieve with a TATAS lock (see Figure 2.6): Threads are anonymous and compete with one another chaotically. Things are not so simple, however, in a queue-based lock: A waiting thread is linked into a data structure on which other threads depend; it cannot simply leave.

Previous work in scheduler-conscious synchronization [KWS97] arranged to mark the queue node of a preempted thread so that the thread releasing the lock would simply pass it over. Upon being rescheduled, a skipped-over thread would have to reenter the queue. A thread that had yet to reach the head of the queue when rescheduled would retain its original position. Several years before, Craig proposed (in narrative form) a similar "mark the node" strategy for queue-based try locks [Cra93a]. Specifically, he suggested that a timed-out thread leave its queue node behind, where it would be reclaimed by another thread when it reached the head of the queue.

Unfortunately, this strategy does not work nearly as well for timeout as it does for preemption. The problem is that a timed-out thread is not idle: It may want to acquire other locks. Because we have no bound on how long it may take for a marked queue node to reach the head of the queue, we cannot guarantee that the same queue node will be available the next time the thread tries to acquire a lock. As Craig notes, the total space required for T threads and L locks rises from $O(T + L)$ with the original CLH lock to $O(T \times L)$ in a mark-the-node try lock. Further, this approach requires dynamic memory allocation of nodes. We also note that the need to skip over abandoned queue nodes increases the worst case lock release time from $O(1)$ to $O(T)$.

We have addressed these space and time problems in new timeout-capable versions of both the CLH and MCS locks. In fairness to Craig, the code presented here requires a `compare_and_swap` (CAS) operation; his work was predicated on the assumption that only `swap` was available. A different version of the MCS-try lock, also using only `swap`, has been developed by Vitaly Oratovsky and Michael O'Donnell of Mercury Computer Corp. [OrO00]. Their lock has the disadvantage that newly arriving threads that have not specified a timeout interval (i.e. that are willing to wait indefinitely) will bypass any already-waiting threads whose patience is more limited. A thread that specifies a timeout may thus fail to acquire a lock—may in fact never acquire a lock, even if it repeatedly becomes available before the expiration of the timeout interval—so long as threads that have not specified a timeout continue to arrive.

CLH-try Lock

In the standard CLH lock, a thread leaves its queue node behind when releasing the lock. In its place it takes the node abandoned by its predecessor. For a try lock we prefer to arrange for a timed-out thread to leave with its own queue node. Otherwise, as noted above, we might need $O(T \times L)$ queue nodes in the system as a whole.

It is relatively easy for a thread B to leave the middle of the queue. Since B's intended successor C (the thread behind it in the queue) is already spinning on B's queue node, B can simply mark the node as "leaving". C can then dereference the node to find B's predecessor A and mark B's node as "recycled", whereupon B can safely reclaim its node and leave. (B's wait for notification from C is an example of

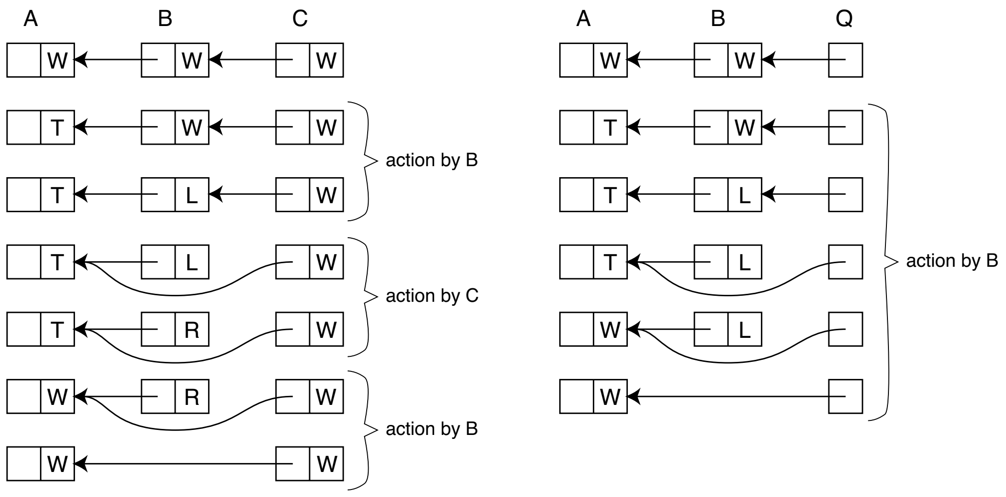


Figure 2.1: Timeout in the CLH-try lock (two principal cases). In the figure at left thread B can leave the middle of the queue as soon as it receives confirmation from its successor, C, that no pointer to its queue node remains. In the figure at right, B can leave the end of the queue once it has updated the tail pointer, Q, using `compare_and_swap`. The transitions from `waiting` to `leaving` and from `waiting` to `available` (not shown) must also be made with `compare_and_swap`, to avoid overwriting a `transient` flag.

handshaking.) There is no race between A and B because A never inspects B's queue node.

Complications arise when the departing thread B is the last thread in the queue. In this case B must attempt to modify the queue's tail pointer to refer to A's queue node instead of its own. We can naturally express the attempt with a CAS operation. If the CAS fails we know that another thread C has arrived. At this point we might hope to revert to the previous (middle-of-the-queue) case. Unfortunately, it is possible that C may successfully leave the queue after B's CAS, at which point B may wait indefinitely for a handshake that never occurs. We could protect against the indefinite wait by repeatedly re-checking the queue's tail pointer, but that would constitute spinning on a non-local location, something we want to avoid.

Our solution is to require C to handshake with B in a way that prevents B from trying to leave the queue while C is in the middle of leaving. (B, likewise, must handshake with A so A cannot leave while B is leaving.) The handshake is effected by defining an additional "transient" state for a queue node. Prior to marking its own node `leaving`, B uses a CAS to mark A's node `transient`. After B handshakes with C (or successfully updates the tail pointer to point to A's node), it changes the status of A's node back to `waiting`. The two principal cases (B in the middle of the queue and at the end) are illustrated in Figure 2.1.

Like the standard CLH lock, the CLH-try lock depends on cache coherence to avoid remote spinning. In the CLH-try lock it is actually possible for two threads to end up spinning on the same location. In the fourth line of the right-hand side of Figure 2.1, if thread A calls `clh_release`, it will spin until the `transient` flag reverts to `waiting`. If a new thread C arrives at about this time, it, too, will begin to spin on the flag in A's queue node, hoping it will change to `available`. When B finally updates the flag, its write will terminate A's spin and cause a reload of C's cached copy. A's immediately subsequent CAS will terminate C's spin.

Code for the CLH-try lock can be found in Listing 2.7. It includes two optimizations that are not obviously necessary in a naive implementation of the algorithm illustrated in Figure 2.1.

First, a lock releaser might become "stuck" if its successor were preempted while timing out, leaving the lock holder's node in `transient` state. Even absent preemption, a pathological timing sequence could allow a series of successors to time out and change the lock holder's node's status word to `transient` just as it was reset to `waiting`. Since these successors are (prior to initiating their time-out sequences) spinning on the same node status word as the lock holder, they will be notified of the update in the same bus transaction as the lock holder; whether the lock holder subsequently releases the lock or the successor begins its timeout sequence is then a question of which update to this same status word happens to win the bus arbitration.

We improve this situation by introducing a new state for the status word that means "transient, but available". Now, the lock holder can alternate between CASing the status word from waiting to available and CASing the status word from `transient` to

```
typedef enum {
   waiting,       // lock is held
   available,     // lock is free
   preleaving,    // thread intends to leave
   leaving,       // node owner is giving up
   transient,     // successor is giving up
   transvailable, // predecessor leaving, lock free
   recycled       // no pointers to node remain
} clh_status;

typedef struct clh_qnode {
   volatile clh_status status;
   volatile struct clh_qnode *volatile prev;
} clh_qnode;

typedef volatile clh_qnode *clh_qnode_ptr;
typedef clh_qnode_ptr clh_try_lock;

bool clh_try_acquire(clh_try_lock *L, clh_qnode_ptr I, hrtime_t T)
{
   clh_status stat;

   I->status = waiting;
   clh_qnode_ptr pred = swap(L, I);

   if (pred->status == available) {
      I->prev = pred;
      return true;
   }
   hrtime_t start = gethrtime();
   bool timeout = false;

   while (1) {
      stat = pred->status;
      if (stat == available) {
         I->prev = pred;
         return true;
      }
      if (stat == leaving) {
         clh_qnode_ptr temp = pred->prev;
         pred->status = recycled;
         pred = temp;
         continue;
      }
      // stat might also be transient, if somebody left the queue just
      // before I entered; or preleaving, if predecessor is timing out
      if (timeout || CUR_TIME - start > T) {
         // if either I or prev == trans, a neighbor
         // is timing out, so I must wait
         timeout == true;
         if (stat == preleaving || stat == transient ||
             stat == transvailable || I->status == transient) {
               continue;
            } else if (!compare_and_store(&I->status, waiting, preleaving)) {
               continue;
            } else break;
        }
   }
```

Listing 2.7: Source code for the CLH-try lock.

```
   // timed out
   while (1) {
      // Spin while predecessor is transient or getting available
      stat = pred->status;
      if (stat == available) {
         I->prev = pred;
         I->status = waiting;
         return true;
      }
      if (stat == waiting) {
         if (compare_and_store(&pred->status, waiting, transient))
            break;
         else continue;
      }
      if (stat == leaving) {
         clh_qnode_ptr temp;
         temp = pred->prev;
         pred->status = recycled;
         pred = temp;
      }
   }

   if (!compare_and_store(L, I, pred)) {
      while (I->status != recycled); // spin
   }
   if (!compare_and_store(&pred->status, transient, waiting))
      pred->status = available;
   return false;
}

void clh_try_release(clh_qnode_ptr *I)
{
    clh_qnode_ptr pred = (*I)->prev;
    while (1) {
        if (compare_and_store(&(*I)->status, waiting, available))
            break;
        if (compare_and_store(&(*I)->status, transient, transvailable))
            break;
    }
    *I = pred;
}
```

Listing 2.7: (continued)

“`transvailable`”. If the successor is preempted then one of these will certainly succeed and the lock holder (though no one else) is free to proceed. Finally, when a timing-out thread sees that its node has become `recycled`, rather than unconditionally mark its predecessor’s status word as `waiting`, it must first attempt to CAS it from `transient` to `waiting`. If the CAS fails then the word must have been `transvailable` and the timing-out thread can simply mark it `available` with a simple write.

Our second optimization introduces a new status value (`preleaving`) that gives preference to older threads when two adjacent waiters in the queue time out at approximately the same time. Without this optimization, younger threads got priority, so an older thread trying to time out could block the queue and not let any other thread reach the head to acquire the lock. At the very beginning of the new time-out sequence, a thread sets its own node's status word to `preleaving`. A successor will decline to change this to `transient`.

MCS-try Lock

A feature of the standard MCS lock is that each thread spins on its own queue node, which may be allocated in local memory even on a machine that does not cache remote locations. To leave the queue, therefore, a thread B must update the successor pointer in the queue node of its predecessor A so that it points to B's successor C, if any, rather than to B. If C later chooses to leave the queue as well, it will again need to update A's queue node, implying that B must tell it where A's queue node resides. Pointers to both predecessors and successors must therefore reside in the queue nodes in memory, where they can be read and written by neighboring threads. These observations imply that an MCS-try lock must employ a doubly linked queue, making it substantially more complex than the CLH-try lock. The benefit of the complexity is better memory locality on a machine that does not cache remote memory locations.

As in the CLH-try lock there are two principal cases to consider for the MCS-try lock, depending on whether the departing thread B is currently in the middle or at the end of the queue. These cases are illustrated in Figure 2.2. While waiting to be granted the lock, a thread ordinarily spins on its predecessor pointer. In the middle-of-the-queue case, departing thread B first replaces the four pointers into and out of its queue node, respectively, with `leaving_other` and `leaving_self` flags (shown as LO and LS in the figure). It then updates C's predecessor pointer, and relies on C to update A's successor pointer. In the end-of-the-queue case, B "tags" A's nil successor pointer to

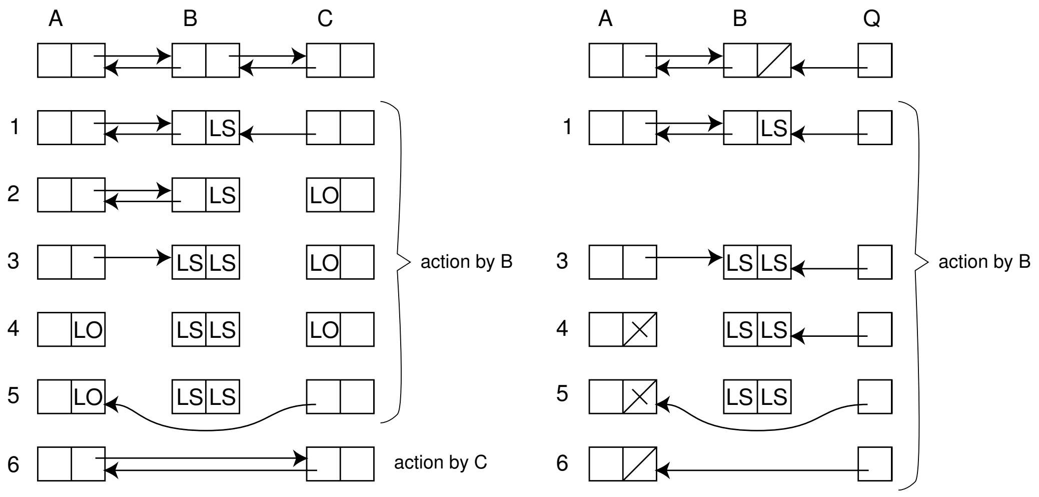


Figure 2.2: Timeout in the MCS-try lock (two principal cases). In the figure at left thread B uses atomic `swap` operations to replace the pointers into and out of its queue node, in a carefully chosen order, with `leaving_other` and `leaving_self` flags. It then updates the pointer from C to A, and relies on C to update the pointer from A to C. In the figure at right, B has no successor at step 2, so it must perform the final update itself. In the meantime it leaves a "tag" on A's (otherwise nil) successor pointer to ensure that a newly arriving thread will wait for B's departure to be finalized.

indicate that additional changes are pending. Absent any race conditions, B eventually clears the tag using CAS.

Unfortunately, there are many potential races that have to be resolved. Thread A at the head of the queue may choose to grant the lock to its successor B while B is attempting to leave the queue. In this case, the race is between A's attempt to change its next pointer to `granted` and B attempt to change this pointer to `leaving_other` in step 4; if A wins the race then B obtains the lock serendipitously and must restore its own next pointer and, if it has a successor C, C's predecessor pointer to it. For its part, C will find that B is again its predecessor when it relinks itself at step 6 of B's departure. Alternatively, if A loses the race, B will have already removed any reference to itself from A's next pointer, so A must wait for B's successor (if present) or B (otherwise) to update its next pointer before it can hand off the lock.

Another race condition occurs when B's predecessor A decides to leave the queue at approximately the same time as B. Generally, the timeout protocol is designed to favor B's departure ahead of A. The crucial decision point is again in B's step 4: If B successfully updates A's next pointer to `leaving_other` before A updates it to `leaving_self` as its step 1, then B is committed to leaving and A must wait for it to do so. In particular, A must here again wait for B's successor (if present) or B (otherwise) to update its next pointer before it can resume its attempt to leave. Alternatively, if A changes its next pointer to `leaving_self` in step 1 before B can change it to `leaving_other`, A has priority over B in leaving, so B must revert its attempt to leave (as before) and wait until A has finished departing.

In yet a third race, a thread B that is initially at the end of the queue in step 2 may discover in step 5 that it now has a successor. Handling this case is straightforward, and mirrors the spin in release for the original MCS lock: B waits until its (new) successor C updates B's next pointer to C.

In general, the order of updates to pointers is chosen to ensure that (1) no thread ever returns from `mcs_try_acquire` until we are certain that no pointers to its queue node remain, and (2) if two adjacent threads decide to leave concurrently, the one closer to the front of the queue leaves first.

Handing off the MCS-try lock is very similar to handing off the original MCS lock. To do so, the lock holder A first marks its next pointer `granted`. This guarantees that if A's successor B should attempt to time out and leave, it will fail step 4 and realize that it has obtained the lock. A then marks B's predecessor pointer `granted` to indicate that B now owns the lock. If this is successful, A waits for B to acknowledge receipt (by changing A's next pointer to nil) before departing.

If A doesn't have a successor, then it attempts to update the lock pointer from itself to nil. If successful, it is free to leave. Otherwise, exactly as in the original MCS lock, a new successor is in the process of enqueuing itself and A must wait to discover that successor's identity so it can hand off the lock.

Finally, in the event that A loses the race to a successor B that is timing out (the first race condition described earlier), A must wait for B to be fully gone and then repeat the entire release cycle with its new successor (if any is present).

Code for the MCS-try lock appears in Listing 2.8.

2.4 Queue-based Locks with Nonblocking Timeout

The algorithms of the previous section have the advantage of cleanly removing a timed-out thread's node from the lock queue. Unfortunately they require handshaking with neighbor threads, which may not currently be running on a multiprogrammed system. Further, the MCS-try lock is extraordinarily complex. In this section we describe MCS-NB-try, [3] a queue-based spin lock that supports timeout and in which the timeout protocol is *wait-free*: A thread can time out and leave the queue in a bounded number of time steps regardless of any action or inaction on the part of other threads.

Where the CLH, CLH-try, MCS, and MCS-try locks all support static memory allocation for queue nodes, MCS-NB-try, like Scott's earlier nonblocking timeout locks, requires dynamic memory management for them. Following Scott's example, nodes are allocated as part of the `acquire` method for MCS-NB-try; this requires writing a pointer to the lock holder's node in a field of the lock itself in order that the `release` method might find the node to reclaim it. As Scott points out, this has the side benefit of allowing a standardized API for the locks. The MCS-NB-try lock reuses Scott's fast memory allocator (see Listing 2.9).

[3]Though it shares the name, this algorithm is distinct from, and supersedes the MCS-NB-try lock previously published by Scott [Sco02]: After we published MCS-try and CLH-try, Scott published MCS-NB-try and CLH-NB-try [Sco02]. In my efforts to create a CLH-NB-NUMA-try lock, we discovered that adding NUMA indirection to CLH yields was very similar to what Scott acheived from eliminating the spin in release in MCS, so CLH-NB-NUMA-try ended up as an improved version of MCS-NB-try. I present it here.

```
typedef struct mcs_qnode {
   volatile struct mcs_qnode *volatile prev;
   volatile struct mcs_qnode *volatile next;
} mcs_qnode;
typedef volatile mcs_qnode *mcs_qnode_ptr;
typedef mcs_qnode_ptr mcs_try_lock;

/* We assume that all valid qnode pointers are word- aligned addresses
(so the two low-order bits are zero) and are larger than 0x10.  We use
low-order bits 01 to indicate restoration of a prev pointer that had
temporarily been changed.  We also use it to indicate that a next
pointer has been restored to nil, but the thread that restored it has
not yet fixed the global tail pointer.  We use low-order bits of 10 to
indicate that a lock has been granted (If it is granted to a thread
that had been planning to leave, the thread needs to know who its
[final] predecessor was, so it can synch up and allow that predecessor
to return from release).  Those two bits are mutually exclusive,
though we don't take advantage of the fact.  We use bogus pointer
values with low-order 0 bits to indicate that a thread or its neighbor
intends to leave, or has recently done so. */

#define nil              ((qnode_ptr) 0x0)
#define leaving_self     ((qnode_ptr) 0x4)
#define leaving_other    ((qnode_ptr) 0x8)
#define gone             ((qnode_ptr) 0xc)
#define restored_tag     ((unsigned long) 0x1)
#define transient_tag   restored_tag
#define granted_tag      ((unsigned long) 0x2)
#define is_restored(p)   ((unsigned long) (p) & restored_tag)
#define is_transient(p) is_restored(p)
#define is_granted(p)    ((unsigned long) (p) & granted_tag)
#define restored(p)      ((qnode_ptr)
                         ((unsigned long) (p) | restored_tag))
#define transient(p)    restored(p)
#define granted(p)       ((qnode_ptr)
                         ((unsigned long) (p) | granted_tag))
#define cleaned(p)       ((qnode_ptr)
                         (((unsigned long) (p)) & ~0x3))
#define is_real(p)       (!is_granted(p)
                        && ((unsigned long) (p) > 0x10))
/* prev pointer has changed from pred to temp, and we know temp is a
real (though possibly restored) pointer. If temp represents a new
predecessor, update pred to point to it, clear restored flag, if any
on prev pointer, and set next field of new predecessor (if not
restored) to point to me.  In effect, RELINK is step 6.  */
#define RELINK                                       \
      do {                                           \
         if (is_restored(temp)) {                    \
            (void) compare_and_swap(&I->prev,        \
                  temp, cleaned(temp));              \
               /* remove tag bit from pointer,       \
                  if it hasn't been changed          \
                  to something else */               \
            temp = cleaned(temp);                    \
```

Listing 2.8: Source code for the MCS-try lock.

```
         } else {                                   \
            pred = temp;                            \
            temp = swap(&pred->next, I);            \
            if (temp == leaving_self) {             \
               /* Wait for predecessor to notice    \
                  I've set its next pointer         \
                  and move on to step 2             \
                  (or, if I'm slow, step 5). */     \
               do {                                 \
                  temp == I->prev;                  \
               } while (temp == pred);              \
               continue;                            \
            }                                       \
         }                                          \
         break;                                     \
      } while (is_real(temp));

bool mcs_try_acquire(mcs_try_lock *L, qnode_ptr I, hrtime_t T)
{
   qnode_ptr pred;
   qnode_ptr succ;
   qnode_ptr temp;

   I->next = nil;
   pred = swap(L, I);
   if (pred == nil) {            // lock was free
      return true;
   }
   // else queue was non-empty
   hrtime_t start = gethrtime();
   I->prev = transient(nil);     // word on which to spin
   // Make predecessor point to me, but preserve transient tag if set.
   do {
      temp = pred->next;
   } while (!compare_and_store(&pred->next, temp,
      (is_transient(temp) ? transient(I) : I)));
   // Old value (temp) can't be granted, because predecessor sets that
   // only via CAS from a legitimate pointer value.  Might be nil
   // (possibly transient) or leaving_self, though.
   if (is_transient(temp)) {
      while (I->prev == transient(nil));  // spin
         // Wait for thread that used to occupy my slot in the queue
         // to clear the transient tag on our (common) predecessor's
         // next pointer.  The predecessor needs this cleared so it
         // knows when it's safe to return and deallocate its qnode.
         // I need to know in case I ever time out and try to leave:
         // The previous setting has to clear before I can safely set
         // it again.
      // Store pred for future reference, if predecessor has not
      // modified my prev pointer already.
      (void) compare_and_swap(&I->prev, nil, pred);
   } else if (temp == leaving_self) {
      // I've probably got the wrong predecessor.  Must wait to learn
      // id of real one, *without timing out*.
      do {
         temp = I->prev;
      } while (temp == transient(nil));
      if (is_real(temp)) {
         RELINK
      }
   } else {
      // Store pred for future reference, if predecessor has not
      // modified my prev pointer already.
      (void) compare_and_swap(&I->prev, transient(nil), pred);
   }
```

Listing 2.8: (continued)

```
  while (1) {
     do {
        if (gethrtime() - start > T) {
           goto timeout;        // 2-level break
        }
        temp = I->prev;
     } while (temp == pred);    // spin
     if (is_granted(temp)) {    // (optimization)
        break;
     }
     if (temp == leaving_other) {
        // Predecessor is leaving; wait for identity of new predecessor.
        do {
           temp = I->prev;
        } while (temp == leaving_other);
     }
     if (is_granted(temp)) {
        break;
     }
     RELINK
  }
  // Handshake with predecessor.
  pred = cleaned(temp);
  pred->next = nil;
  return true;

  // Step 1: Atomically identify successor and indicate intent to leave
timeout:
  while (1) {
     do {
        do {
           succ = I->next;
        } while (is_transient(succ));
     } while (!compare_and_store(&I->next, succ, leaving_self));
        // don't replace a transient value
     if (succ == gone)
        succ = nil;
     if (succ == nil) {
        // No visible successor; we'll end up skipping step 2 and
        // doing an alternative step 6.
        break;
     }
     if (succ == leaving_other) {
        // Wait for new successor to complete step 6.
        do {
           temp = I->next;
        } while (temp == leaving_self);
        continue;
     }
     break;
  }
  // Step 2: Tell successor I'm leaving
  if (succ != nil) {
     temp = swap(&succ->prev, leaving_other);
     if (temp == leaving_self) {
        // Successor beat me to the punch.  Must wait until it tries
        // to modify my next pointer, at which point it will know I'm
        // leaving, fail step 4, and wait for me to finish.
        do {
           temp = I->next;
        } while (temp != succ);
     }
  }
```

Listing 2.8: (continued)

```
// Step 3: Identify predecessor and indicate intent to leave.
while (1) {
   qnode_ptr n;
   temp = swap(&I->prev, leaving_self);
   if (is_granted(temp)) {
      goto serendipity;
   }
   if (temp == leaving_other) {
      // Predecessor is also trying to leave; it takes precedence.
      do {
         temp = I->prev;
      } while (temp == leaving_self || temp == leaving_other);
      if (is_granted(temp)) {
         // I must have been asleep for a while.  Predecessor won
         // the race, then experienced serendipity itself, restored
         // my prev pointer, finished its critical section, and
         // released.
         goto serendipity;
      }
      RELINK
      continue;
   }
   if (temp != pred) {
      RELINK
      // I'm about the change pred->next to leaving_other or
      // transient(nil), so why set it to I first?  Because
      // otherwise there is a race.  I need to avoid the
      // possibility that my (new) predecessor will swap
      // leaving_self into its next field, see leaving_other (left
      // over from its old successor), and assume I won the race,
      // after which I come along, swap leaving_other in, get
      // leaving_self back, and assume my predecessor won the race.
   }

   // Step 4: Tell predecessor I'm leaving.
   if (succ == nil) {
      n = swap(&pred->next, transient(nil));
   } else {
      n = swap(&pred->next, leaving_other);
   }
   if (is_granted(n)) {
      // Predecessor is passing me the lock; wait for it to do so.
      do {
         temp = I->prev;
      } while (temp == leaving_self);
      goto serendipity;
   }
   if (n == leaving_self) {
      // Predecessor is also trying to leave.  I got to step 3
      // before it got to step 2, but it got to step 1 before I got
      // to step 4.  It will wait in Step 2 for me to acknowledge
      // that my step 4 has failed.
      pred->next = I;
      // Wait for new predecessor (completion of old predecessor's step 5)
      do {
         temp = I->prev;
      } while (temp == leaving_self    // what I set
           || temp == leaving_other);  // what pred will set in step 2
      if (is_granted(temp))
         goto serendipity;
      RELINK
      continue;
   }
   break;
}
```

Listing 2.8: (continued)

```
   if (succ == nil) {
      // Step 5: Try to fix global pointer.
      if (compare_and_store(L, I, pred)) {
         // Step 6: Try to clear transient tag.
         temp = compare_and_swap(&pred->next, transient(nil), gone);
         if (temp != transient(nil)) {
            pred->next = cleaned(temp);
            // New successor got into the timing window, and is now
            // waiting for me to signal it.
            temp = cleaned(temp);
            (void) compare_and_swap(&temp->prev, transient(nil), nil);
         }
      } else {
         // Somebody has gotten into line.  It will discover that I'm
         // leaving and wait for me to tell it who the real
         // predecessor is (see pre-timeout loop above).
         do {
            succ = I->next;
         } while (succ == leaving_self);
         pred->next = leaving_other;
         succ->prev = pred;
      }
   } else {
      // Step 5: Tell successor its new predecessor.
      succ->prev = pred;
   }
   // Step 6: Count on successor to introduce itself to predecessor.
   return false;

serendipity:   // I got the lock after all; temp contains a granted
               // value read from my prev pointer.

   // Handshake with predecessor:
   pred = cleaned(temp);
   pred->next = nil;
   if (succ == nil) {
      // I don't think I have a successor.  Try to undo step 1.
      if ((temp = compare_and_swap(&I->next,
             leaving_self, succ)) != leaving_self) {
         // I have a successor after all.  It will be waiting for me
         // to tell it who its real predecessor is.  Since it's me
         // after all, I have to tag the value so successor will
         // notice change.  Undo step 2:
         temp->prev = restored(I);
      }
   } else {
      // I have a successor, which may or may not have been trying to
      // leave.  In either case, I->next is now correct.  Undo step 1:
      I->next = succ;
      // Undo step 2:
      succ->prev = restored(I);
   }
   return true;
}
```

Listing 2.8: (continued)

```
void mcs_release(mcs_try_lock *L, qnode_ptr I)
{
   qnode_ptr succ;
   qnode_ptr temp;

   while (1) {
      succ = I->next;
      if (succ == leaving_other || is_transient(succ)) {
         // successor is leaving, but will update me.
         continue;
      }
      if (succ == gone) {
         if (compare_and_store(&I->next, gone, nil)) {
            succ = nil;
         } else {
            continue;
         }
      }
      if (succ == nil) {
         // Try to fix global pointer.
         if (compare_and_store(L, I, nil)) {
            // Make sure anybody who happened to sneak in and leave
            // again has finished looking at my qnode.
            do {
               temp = I->next;
            } while (is_transient(temp));
            return;  // I was last in line.
         }
         do {
            temp = I->next;
         } while (temp == nil);
         continue;
      }
      if (compare_and_store(&I->next, succ, granted(nil))) {
         // I have atomically identified my successor and indicated
         // that I'm releasing; now tell successor it has the lock.
         temp = swap(&succ->prev, granted(I));
         // Handshake with successor to make sure it won't try to
         // access my qnode any more (might have been leaving).
         do {
            temp = I->next;
         } while (temp != nil);
         return;
      }
      // Else successor changed. Continue loop. Note that every
      // iteration sees a different (real) successor, so there isn't
      // really a global spin.
   }
}
```

Listing 2.8: (continued)

```
// Code to manage a local but shared pool of qnodes.
// All nodes are allocated by, and used by, a given thread,
// and may reside in local memory on an NCC-NUMA machine.
// The nodes belonging to a given thread form a circular
// singly linked list.  The head pointer points to the node
// most recently successfully allocated.  A thread allocates
// from its own pool by searching forward from the pointer
// for a node that's marked "unused".  A thread (any thread)
// deallocates a node by marking it "unused".

typedef struct local_qnode {
   union {
      clh_nb_qnode cnq;           // members of this union are
      mcs_nb_qnode mnq;           // never accessed by name
   } real_qnode;
   volatile bool allocated;
   struct local_qnode *next_in_pool;
} local_qnode;

typedef struct {
   local_qnode *try_this_one;     // pointer into circular list
   local_qnode initial_qnode;
} local_head_node;

inline local_qnode *alloc_local_qnode(local_head_node *hp)
{
   local_qnode *p = hp->try_this_one;
   if (!p->allocated) {
      p->allocated = true;
      return p;
   } else {
      local_qnode *q = p->next_in_pool;
      while (q != p) {
         if (!q->allocated) {
            hp->try_this_one = q;
            q->allocated = true;
            return q;
         } else {
            q = q->next_in_pool;
         }
      }
      // All qnodes are in use.
      // Allocate new one and link into list.
      special_events[mallocs]++;
      q = (local_qnode *) malloc(sizeof(local_qnode));
      q->next_in_pool = p->next_in_pool;
      p->next_in_pool = q;
      hp->try_this_one = q;
      q->allocated = true;
      return q;
   }
}

#define free_local_qnode(p) ((p)->allocated = false)
```

Listing 2.9: Scott's routines for fast space management.

MCS-NB-Try Lock

Our MCS-NB-try lock combines nonblocking timeout with strictly local spinning. As usual, an MCS-NB lock variable takes the form of a tail pointer for a list of queue nodes, but where Scott's CLH-NB queue is linked from tail to head, the bulk of the MCS-NB queue is linked from head to tail. After swapping a reference to its own queue node into the tail pointer, a thread writes an additional reference to its node into the `next` pointer of its predecessor's node. It then spins on its own node's status, rather than the predecessor's node. As in the original MCS lock, this "backward" linking enables local spinning, even on a machine that does not cache remote locations.

To release a standard MCS lock, a thread attempts to follow its `next` pointer and update the word on which its successor is spinning. If the pointer is still nil, the thread performs a CAS on the lock tail pointer, in an attempt to replace a pointer to its own node with a nil pointer. If that attempt fails then some other thread must be in the process of linking itself into the queue. The releasing thread waits for its `next` pointer to be updated, then follows it and updates the successor's status word. As with handshaking in the locks of Section 2.4 we must eliminate the spin in `release` if we are to bound the time required by lock operations.

The solution employs a technique due to Craig, who explains how to employ an extra level of indirection in a CLH lock to obtain local-only spinning on machines that do not cache remote locations [Cra93a]. (His lock variant, which we call CLH-NUMA, is generally slower than the MCS lock, but it does not require CAS.) The idea is that when a thread A wants to change the status of its successor node, it swaps a sentinel value into its `next` pointer, rather than merely reading it. If the `swap` returns a valid pointer, it dereferences this and updates the successor's node. If the `swap` returns `nil`, however, no further action is required: A successor B, when and if it appears, will find the sentinel when it attempts to swap the address of its own node into A's `next` pointer.

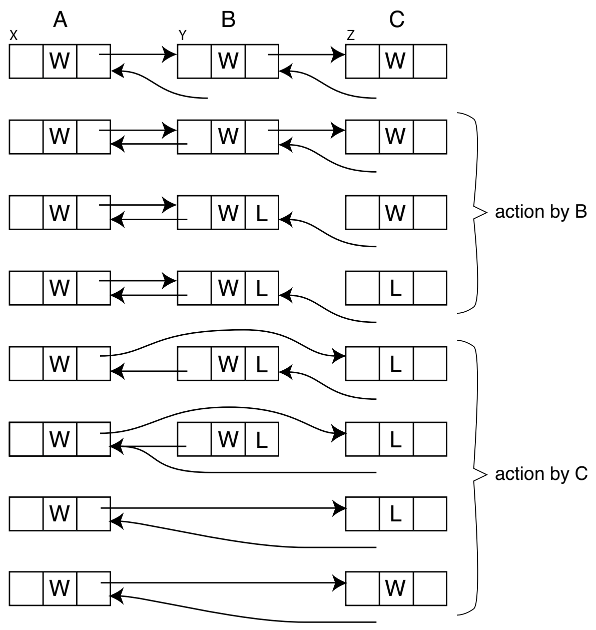


Figure 2.3: Timeout in the MCS-NB-try lock, with departing thread B in the middle. If thread C is not present, B can return after the third line of the figure; the next arriving thread will reclaim node Y. For clarity, references to predecessor nodes that are held in local variables, rather than in qnodes, are not shown.

The value of the sentinel tells B what A would have written to B's status word if A had known how to find it.

Queue nodes in the MCS-NB lock (see Figure 2.3) contain a status flag and a `pair` of pointers, used to link the queue in both directions. When nil, the backward-pointing `next` pointer indicates that no successor node is known. Two sentinel values (assumed not to be the same as any valid reference) correspond to special states. When set to `AVAILABLE`, the `next` pointer indicates that the lock is currently available. When set to `LEAVING`, it indicates that the thread that allocated the node has timed out. The forward `prev` pointer is normally nil, but when a thread times out it fills this pointer with a reference to its predecessor's node; this allows its successor to remove it from the queue.

The status word of a queue node (separate from either pointer) has three possible values. Before linking its node into the queue, a thread initializes its status word to waiting. Once the link-in operation is complete, the thread spins, waiting for the value to change. The other two possible values—available and leaving—mirror the values of the predecessor's next pointer, described in the previous paragraph.

When a thread C performs an initial swap on the tail pointer of a lock that is not currently available, it receives back a reference to the queue node Y allocated by C's predecessor, B. C swaps a reference to its own node, Z, into Y's next pointer. If the swap returns AVAILABLE, C knows that it has the lock. Moreover, C knows that no other thread has performed a swap on Y's next pointer, so B must have abandoned this node in the queue at release time and C is responsible for reclaiming Y. Otherwise, C is linked into the queue and it spins on the status word in its node (Z). When the word changes to available, C writes a reference to Z into the head pointer field of the lock, and returns successfully.

If Z's status word changes to leaving, C knows that B has timed out. It must therefore clean up the queue and find a new predecessor node X allocated by thread A. C first resets its status word to waiting, and then reads Y's prev pointer to find A's node X. C then swaps a reference to Z into X's next pointer. If the swap returns a reference to Y, C knows that Y has been removed from the queue and can safely be reclaimed; otherwise A must have changed its own next pointer.

One possibility is that A is in the process of releasing the lock, in which case the swap might return AVAILABLE. C knows that A will at some point write available to Y's status word. If C were to reclaim Y before this happens, mutual exclusion could be compromised if Y were reinserted into the queue before A's update. Hence, C *also* swaps available into Y's status field; whichever of C and A completes the swap *last* (getting available as the result of the swap) knows that it is responsible for reclaiming Y. Either way, C knows that it will be next to get the lock, so it writes a reference to Z into the head pointer field of the lock, and returns successfully.

The other possibility is that both A and B are in the process of timing out, in which case C's swap of X's next pointer might return LEAVING. Paralleling the previous case, A and C both swap `leaving` into Y's status word; whichever `swap` returns `leaving` indicates responsibility for reclaiming the node. Once this is decided, C updates its local `pred` pointer from Y to X, writes `LEAVING` back into X's `next` pointer, and branches back to the top of the `acquire` routine. On its next pass through the code it will see the sentinel and attempt to reclaim X.

While waiting for the lock, A spins on X's status word and also periodically polls the system clock. If it times out in this inner loop, A first writes its local `pred` pointer into X's `prev` pointer. It then proceeds to swap the `LEAVING` sentinel into X's `next` pointer. If this `swap` does not return a valid pointer, then A has no successor and it is free to leave; the next thread that arrives will reclaim X. Otherwise, A swaps `leaving` into Y's status word to inform B that it (A) is leaving. If this swap returns `waiting` the timeout is complete. It may also return `leaving`, however, in which case A has reached the other half of the race condition described in the previous paragraph. A knows that B has timed out, that A won the race to update X's `next` pointer, that C won the race to update Y's status word, and that A is therefore responsible for reclaiming Y.

Releasing a MCS-NB-try lock occurs in two steps. First, thread A `swap`s the sentinel `AVAILABLE` into its next pointer. If this `swap` returns nil, no successor has yet arrived and the next to do so will see the sentinel and know it has acquired the lock. Otherwise, the swap returns the node Y of A's successor B. A `swaps` `available` into Y's status word to inform B that it has now received the lock. In a manner analogous to the actions of the previous paragraph, if this second `swap` returns `available`, then A must reclaim Y. Source code for the MCS-NB-try appears in Listing 2.10.

Like the original CLH lock, the MCS-NB-try lock uses only `read`, `write`, and `swap` instructions to access memory. This settles in the affirmative an open question posed by Oratovsky and O'Donnell [OrO00]: namely, whether it is possible to construct a fair, timeout-capable queue-based lock without a universal primitive like `compare_and_swap` or `load_linked/store_conditional`.

Interestingly, both the original MCS lock and the MCS-try lock require CAS. The explanation is as follows: Where the original MCS lock guarantees that a thread's queue node will be reusable when the `release` routine returns, MCS-NB has no hope of doing so. Because it plans to leave its node behind, it can safely swap an `AVAILABLE` sentinel into its `next` pointer and avoid the need to CAS the tail pointer of the queue if no successor has yet announced itself. Of course by the same token a thread can swap a `LEAVING` sentinel into its `next` pointer when it times out, and in this case its node will not be reclaimed until some new thread joins the queue.

Scott previously described an alternative version of the MCS-NB-try lock that uses CAS to reclaim its own node when it is the last thread in the queue [Sco02]. Unfortunately, to resolve a race condition when the second-to-last thread is also timing out, this version must incorporate an extra first step in the timeout protocol, in which a thread erases its predecessor's `next` pointer. On balance, we do not believe that the desire to reclaim end-of-queue nodes immediately justifies the time and complexity introduced by the extra step (not to mention the need for CAS).

2.4.1 Space Requirements

Like Scott's CLH-NB-try lock, the MCS-NB try lock typically abandons queue nodes in an unsuccessful `acquire` operation. This means that the $O(L + T)$ space bound for L locks and T threads that the non-timeout and blocking timeout CLH and MCS locks achieve is not matched by MCS-NB-try. In fact, Scott describes a scenario observed by Victor Luchangco of Sun Microsystems' Scalable Synchronization Research Group [Luc02] in which worst-case space overhead is unbounded; the same scenario applies to the new MCS-NB-try lock as well.

```
typedef enum {available, leaving, waiting
} qnode_status;
typedef struct mcs_nb_qnode {
   struct mcs_nb_qnode *   volatile prev;
   volatile qnode_status   status;
   struct mcs_nb_qnode *   volatile next;
} mcs_nb_qnode;
typedef mcs_nb_qnode *volatile mcs_nb_qnode_ptr;
typedef struct {
   mcs_nb_qnode_ptr   volatile tail;
   mcs_nb_qnode_ptr   lock_holder;
      // node allocated by lock holder
} mcs_nb_lock;

#define AVAILABLE ((mcs_nb_qnode_ptr) 1)
#define LEAVING   ((mcs_nb_qnode_ptr) 2)

#define mqn_swap(p,v) (mcs_nb_qnode *) \
   swap((volatile unsigned long *) (p), (unsigned long) (v))
#define s_swap(p,v) (qnode_status)
   swap((volatile unsigned long *) (p), (unsigned long) (v))

#define alloc_qnode() \
   (mcs_nb_qnode_ptr)alloc_local_qnode(my_head_node_ptr())

bool mcs_nb_try_acquire(mcs_nb_lock *L, hrtime_t T)
{
   mcs_nb_qnode_ptr I = alloc_qnode();
   mcs_nb_qnode_ptr tmp, pred, pred_pred;
   I->status = waiting;
   I->prev = NULL;
   I->next = NULL;
   pred = mqn_swap(&L->tail, I);
   hrtime_t start = START_TIME;

   // Each loop, we link to a new predecessor
   while (1) {
      tmp = mqp_swap(&pred->next, I);
      if (tmp == AVAILABLE) {
         // lock was free; just return
         L->lock_holder = I;
         free_qnode(pred);
         return true;
      }
      if (!tmp) {
spin:    while (I->status == waiting)
            if (CUR_TIME - start > T)
               goto timeout;
         if (I->status == available) {
            free_qnode(pred);
            L->lock_holder = I;
            return true;
         }
         I->status = waiting; // reset status
         // fall through to predecessor-timed-out case
      }
```

Listing 2.10: Source code for the MCS-NB-try lock.

```
      // Predecessor timed out, get new predecessor
      pred_pred = pred->prev;
      tmp = mqp_swap(&pred_pred->next, I);
      if (tmp == pred) {
         free_qnode(pred);
         recovered_nodes++;
         pred = pred_pred;
         goto spin; // target is UP 20 lines
      } else if (tmp == AVAILABLE) {
         // Pred_pred will try to access pred's status;
         // we and it are in a race.
         if (s_swap(&pred->status, available) == available) {
            // Pred_pred got to the status before us.
            // We'll get the lock (next pass), but
            // must recycle pred.
            free_qnode(pred);
         } // else swap returned waiting: we won the
           // race to get to pred's status.  Pred_pred
           // will still try to access the status at
           // some point. It will know to recycle pred.

         // Recycle pred_pred and take the lock
         free_qnode(pred_pred);
         L->lock_holder = I;
         return true;
      } else {
         // Pred_pred will try to access pred's status;
         // we and it are in a race.
         if (s_swap(&pred->status, leaving) == leaving) {
            // Pred_pred got to the status before we
            // did: we must recycle pred.
            free_qnode(pred);
         } // else we won the race.  Pred_pred will
           // still try to access the status at some
           // point.  It will know to recycle pred.

         pred = pred_pred;
         pred->next = LEAVING;
            // to fool ourselves into thinking this
            // is a normal top-of-loop case.  Note
            // that pred (formerly pred_pred) has
            // already either tried to release the
            // lock or timed out, so it won't
            // subsequently try to change the field
            // we just wrote into, so the plain
            // write (not swap) is safe.
         if CUR_TIME - start > T) {
            break; /* drop to timeout code below */
         }
      }
   }
```

Listing 2.10: (continued)

```
   // Timed out
timeout:
   I->prev = pred;
   tmp = mqp_swap(&I->next, LEAVING);
   if (tmp) {
      // Tell my successor I've timed out
      if (s_swap(&tmp->status, leaving) == leaving) {
         // My successor's successor beat me to this
         //  point, so I need to recycle my successor's
         // node. This status value implies that my
         // successor is also timed out.
         free_qnode(tmp);
      }
   }
   return false;
}

void mcs_nb_try_release(mcs_nb_lock *L)
{
   mcs_nb_qnode_ptr I = L->lock_holder;
   mcs_nb_qnode_ptr succ = mqn_swap(&I->next, AVAILABLE);
   if (succ) {
      if (swap(&succ->status, available) == available) {
        // somebody beat us to succ->status; we are
        // responsible for recycling succ.
        free_qnode(succ);
      } // else swap returned waiting: clean hand-off
   }
}
```

Listing 2.10: (continued)

2.5 Time-Published Queue-based Locks

The locks of Sections 2.3 and 2.4 allow a thread to cleanly leave the queue of a CLH- or MCS-style lock. By choosing a timeout significantly smaller than a scheduling quantum, these locks can be used to tolerate preemption of predecessors in the queue, *assuming the lock is not on the application's critical path.* This assumption is safe in some applications but not in others. For the general case we need (a) a better way to prevent or recover from preemption of the lock holder and (b) a way to avoid passing a lock to an already preempted thread.

As noted in Section 2.1, some operating systems provide mechanisms that address need (a), but such mechanisms are neither universal nor standardized. To the best of our knowledge, no commercial operating system provides mechanisms sufficient to address need (b). Most modern machines and operating systems do, however, provide a fine grain, low overhead user-level clock, often in the form of a fast running cycle or nanosecond counter register, that is synchronized across processors. In this Section we explain how to use such a clock to build *preemption tolerant* queue locks—locks that are free, in practice, from virtually all preemption-induced performance loss.

Each waiting thread in our *time-published* locks, co-designed with Bijun He, periodically writes the current system time to a thread-specific shared location. If a thread A sees a stale timestamp for its predecessor thread B, A assumes that B has been preempted and removes B's node from the queue. Further any thread that acquires a lock writes the time at which it did so to a lock-specific shared location. If it sees an acquisition timestamp that is sufficiently far in the past (farther than longest plausible critical section time—the exact value is not critical), A yields the processor in the hope that a suspended lock holder might resume. Use of the acquisition timestamp allows A to distinguish, with high accuracy, between preemption of the lock holder and simple high contention, both of which manifest themselves as long delays in lock acquisition time.

There is a wide design space for time-published locks, portions of which are illustrated in Figure 2.4. Two specific points in this design space—the MCS-TP and CLH-TP locks—are described in the following two subsections. These algorithms reflect straightforward strategies consistent with the head-to-tail and tail-to-head linking of the MCS and CLH locks, respectively.

2.5.1 The MCS Time-Published Lock

Our MCS-TP try lock uses the same head-to-tail linking as other MCS variants. Unlike the MCS and MCS-NB try locks of Sections 2.3 and 2.4, however, it does not attempt to remove abandoned nodes from the queue.

We add two new states to the `waiting` and `available` of the standard MCS lock. When a waiting thread times out before acquiring the lock, it marks its node `left` and returns, leaving the node in the queue. When a node reaches the head of

Lock	MCS-TP	CLH-TP
Link Structure	Queue linked head to tail	Queue linked tail to head
Lock handoff	Lock holder explicitly grants the lock to a waiter	Lock holder marks lock available and leaves; next-in-queue claims lock
Timeout precision	Strict adherence to patience	Bounded delay from removing timed-out and preempted predecessors
Queue management	Only the lock holder removes timed-out or preempted nodes (at handoff)	Concurrent removal by all waiting threads
Space management	Semi-dynamic allocation: Waiters may reinhabit abandoned nodes until removed from the queue	Dynamic allocation: Separate node per acquisition attempt

Figure 2.4: Comparing the MCS-TP and CLH-TP time-published locks

the queue but is either marked `left` or appears to be owned by a preempted thread (i.e., has a stale timestamp), the lock holder marks it `removed`, and follows its `next` pointer to find a new candidate lock recipient, repeating as necessary. Figure 2.5 shows state transitions for MCS-TP queue nodes. Source code for the MCS-TP lock appears in Listing 2.13.

The MCS-TP algorithm allows each thread at most one node per lock. If a thread that calls `acquire` finds its previous node (which is cached on a failed acquire operation) still marked `left`, it reverts the state to `waiting`, resuming its former place in line. Otherwise, it begins a fresh attempt from the tail of the queue. To all other threads, timeout and retry are indistinguishable from an execution in which the thread was waiting all along.

To guarantee bounded-time lock handoff, we must avoid a pathological case in which waiting threads might repeatedly time out, have their nodes removed, rejoin the queue, and then time out again before obtaining the lock. In this scenario, a lock holder might see an endless treadmill of abandoned nodes, and never be able to release the

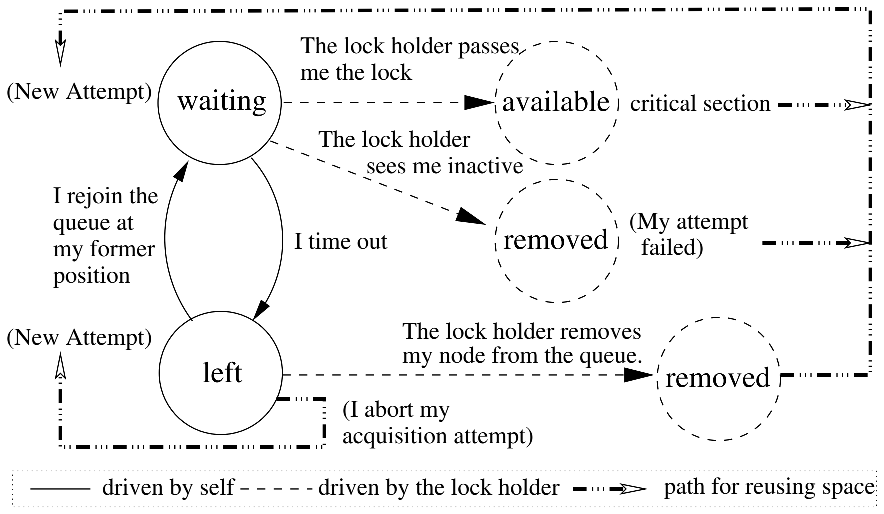

(New Attempt)
waiting
The lock holder passes me the lock
available
critical section
The lock holder sees me inactive
removed
(My attempt failed)
I rejoin the queue at my former position
I time out
(New Attempt)
left
The lock holder removes my node from the queue.
removed
(I abort my acquisition attempt)
driven by self
driven by the lock holder
path for reusing space

Figure 2.5: MCS-TP queue node state transitions

lock. We therefore arrange for the lock holder to remove only the first T `left` nodes it encounters; thereafter, it scans the list until it either reaches the end or finds a viable successor. Only then does it mark the scanned nodes `removed`. (If a scanned node's owner comes back to waiting before being marked `removed`, it will eventually see the `removed` state and quit as a failed attempt). Because skipped nodes' owners reclaim their existing (bypassed) spots in line, the length of the queue is bounded by the total number of threads T and this process is guaranteed to terminate in at most $2T$ steps. In practice, we have never observed the worst case, even in contrived "torture tests"; lock holders typically find a viable successor within the first few nodes.

2.5.2 The CLH Time-Published Lock

Our CLH-TP try lock retains the tail-to-head linking of the CLH lock, but removes nodes inserted by threads that have timed out or (appear to have) been preempted. Unlike MCS-TP, CLH-TP allows any thread to remove the node inserted by a preempted predecessor; removal is not reserved to the lock holder. Middle-of-the-queue removal adds significant complexity to CLH-TP; experience with the MCS

```
typedef struct mcstp_qnode {
  mcstp_lock *last_lock;   // lock from last attempt
  volatile hrtime_t time;  // published timestamp
  volatile qnode_status status;
  struct mcstp_qnode *volatile next;
} mcstp_qnode;

typedef struct mcstp_lock {
  mcstp_qnode *volatile tail;
  volatile hrtime_t cs_start_time;
} mcstp_lock;

#define mtp_swap(p,v) ((mcstp_qnode *) \
   swap((volatile unsigned long *)(p), (unsigned long)(v)))
#define compare_and_store(p,o,n) \
   (cas((volatile unsigned long *) (p), \
       (unsigned long) (o), (unsigned long) (n)) \
         == (unsigned long) (o))

extern int MAX_CS_TIME;  // Approximate upper bound
                         // on the length of a critical
                         // section.
extern int MAX_THREADS;  // Approximate max number
                         // of threads in the system.
extern int UPDATE_DELAY; // Approximate length of time
                         // it takes a thread to see
                         // a timestamp published on
                         // another thread, including
                         // any potential clock skew.

bool mcstp_acquire(mcstp_lock *L, mcstp_qnode *I,
                   hrtime_t T)
{
   mcstp_qnode *pred;
   hrtime_t start_time = START_TIME;

   // try to reclaim position in queue
   if (I->status != timedout || I->last_lock != L ||
      !compare_and_store(&I->status, timedout, waiting)) {
      I->status = waiting;
      I->next = 0;
      pred = swap(&L->tail, I);

      if (!pred) { // lock was free
         L->cs_start_time = gethrtime();
         return true;
      } else pred->next = I;
   }

   while (1) {
      if (I->status == available) {
         L->cs_start_time = gethrtime();
         return 1;
      } else if (I->status == failed) {
         if (CUR_TIME - L->cs_start_time > MAX_CS_TIME)
            yield();
         I->last_lock = L;
         return false;
      }
```

Listing 2.11: Source code for the MCS-TP lock.

```
      while (I->status == waiting) {
         I->time = gethrtime();
         if (CUR_TIME - start_time <= T)
            continue;
         if (!compare_and_store(&I->status,
            waiting, timedout)) {
            I->last_lock = L;
            break;
         }
         if (CUR_TIME - L->cs_start_time > MAX_CS_TIME)
            yield();
         return false;
      }
   }
}

void mcstp_release (mcstp_lock *L, mcstp_qnode *I)
{
   int scanned_nodes = 0;
   mcstp_qnode *succ, *curr = I, *last = NULL;

   while (1) {
      succ = curr->next;
      if (!succ) {
         if (compare_and_store(&L->tail, curr, 0)) {
            curr->status = failed;
            return; // I was last in line.
         }
         while (!succ)
            succ = curr->next;
      }
      if (++scanned_nodes < MAX_THREADS)
         curr->status = failed;
      else if (!last)
         last = curr; // handle treadmill case
      if (succ->status == waiting) {
         hrtime_t succ_time = succ->time;
         if ((CUR_TIME - succ_time <= UPDATE_DELAY) &&
            compare_and_store(&succ->status, waiting,
            available)) {
               for ( ; last && last != curr; last = last->next)
                  last->status = failed;
               return;
         }
      }
      curr = succ;
   }
}
```

Listing 2.11: (continued)

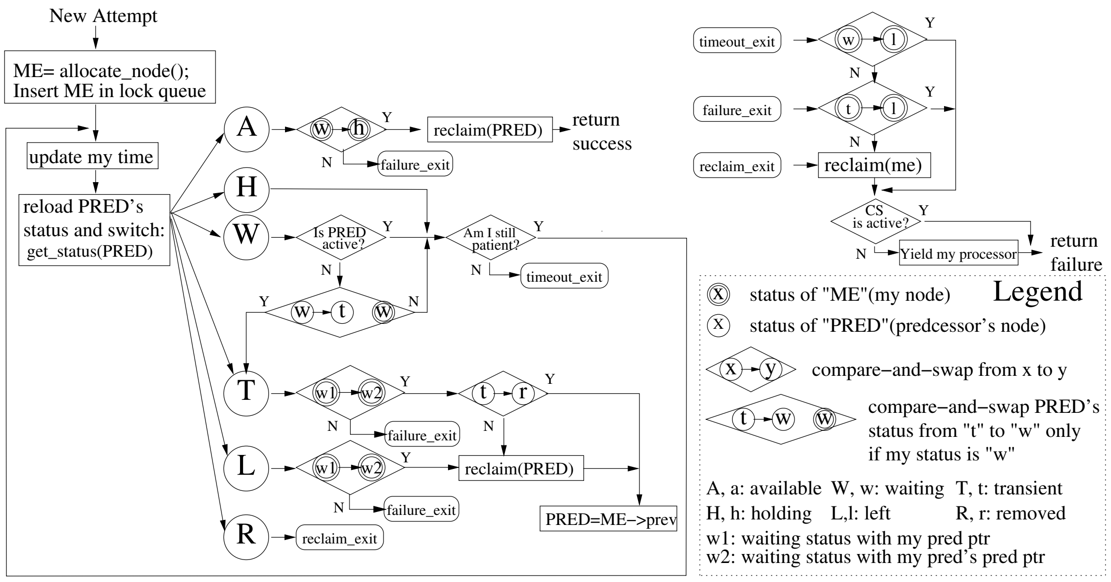

New Attempt
ME= allocate_node(); Insert ME in lock queue
update my time
reload PRED's status and switch: get_status(PRED)
A
H
W
T
L
R
reclaim(PRED)
return success
failure_exit
Is PRED active?
Am I still patient?
timeout_exit
PRED=ME->prev
reclaim_exit
Y
N
timeout_exit
failure_exit
reclaim_exit
reclaim(me)
CS is active?
Yield my processor
return failure
Legend
status of "ME"(my node)
status of "PRED"(predcessor's node)
compare–and–swap from x to y
compare–and–swap PRED's status from "t" to "w" only if my status is "w"
A, a: available W, w: waiting T, t: transient
H, h: holding L,l: left R, r: removed
w1: waiting status with my pred ptr
w2: waiting status with my pred's pred ptr

Figure 2.6: Control flow for the CLH-TP lock

and MCS-NB try locks suggests that it would be very difficult to add to MCS-TP. Source code for the CLH-TP lock appears in Listing 2.13. (Although the code shown uses a `compare_and_swap` operation, this is trivial to emulate via `load_linked/store_conditional`; only `load_linked/store_conditional` – on an emulation of it via CAS – is needed to implement this algorithm.) Figure 2.6 provides an overview of control flow for the CLH-TP algorithm.

We use low-order bits in a CLH-TP node's `prev` pointer to store the node's state, allowing us to modify the state and the pointer together, atomically. If `prev` is a valid pointer, its two lowest-order bits specify one of three states: `waiting`, `transient`, and `left`. Alternatively, `prev` can be a `nil` pointer with low-order bits set to indicate three more states: `available`, `holding`, and `removed`. Figure 2.7 shows the state transition diagram for CLH-TP queue nodes.

In each lock acquisition attempt, thread B dynamically allocates a new node Y and links it to predecessor X as before. While waiting, B handles three events. The simplest occurs when X's state changes to `available`; B atomically updates Y's

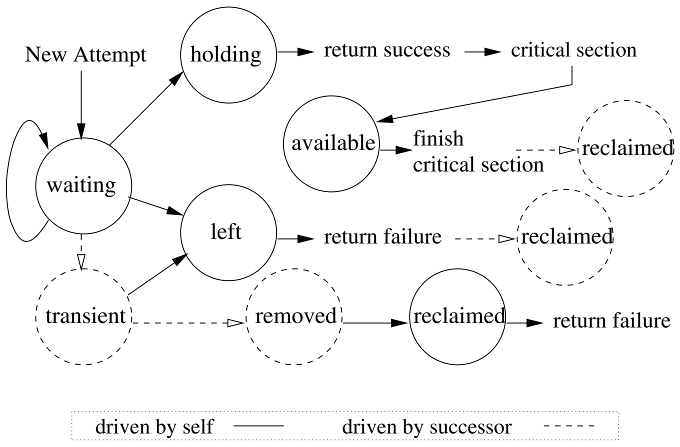


Figure 2.7: CLH-TP queue node state transitions

state to `holding` to claim the lock. The other two are described in the paragraphs below.

The second event occurs when B sees that X has been marked `left` (implying it has timed out), or that its timestamp has grown stale (implying it has been preempted). If X is marked `left`, B knows that A has left the queue and will no longer access X. B therefore links X out of the queue and reclaims it. If X is still marked `waiting`, but A's timestamp is stale, B performs a three-step *removal sequence* to unlink A's node from the queue. First, B atomically changes X's state from `waiting` to `transient`, to prevent A from acquiring the lock or from reclaiming and reusing X if it is removed from the queue by some successor of B (more on this below). Second, B removes X from the queue, simultaneously verifying that B's own state is still `waiting` (since Y's `prev` pointer and state share a word, this is a single `compare_and_swap`). Hereafter, X is no longer visible to other threads in the queue, and B spins on A's predecessor's node. Finally, B marks X as safe for reclamation by changing its state from `transient` to `removed`.

The third event occurs when B times out or when it notices that Y is `transient`. In either case, it attempts to atomically change Y's state from `transient` or `wait-`

```
int cas_w_waiting(node_t * volatile *addr,
                  unsigned long     oldv,
                  unsigned long     newv,
                  node_t * volatile *me)
{
    do {
        unsigned long tmp = LL(addr);
        if (tmp != oldv || !is_waiting(me))
            return 0;
    } while(!SC(addr, newv));
    return 1;
}
```

Listing 2.12: Conditional updates in CLH-TP

`ing` to `left`. If the attempt (a `compare_and_swap`) succeeds, B has delegated responsibility for reclamation of Y to a successor. Otherwise, B has been removed from the queue and must reclaim its own node. In both cases, whichever of B and its successor is the last to notice that Y has been removed from the queue handles the memory reclamation; this simplifies memory management.

A corner case occurs when, after B marks X `transient`, Y is marked `transi-ent` by some successor thread C before B removes X from the queue. In this case, B leaves X for C to clean up; C recognizes this case by finding X already `transient`.

The need for the `transient` state derives from a race condition in which B decides to remove X from the queue but is preempted before actually doing so. While B is not running, successor C may remove both Y and X from the queue, and A may reuse its node in this or another queue. When B resumes running, we must ensure that it does not modify (the new instance of) A. The `transient` state allows us to do so, if we can update X's state and verify that Y is still `waiting` as a single atomic operation. The custom atomic construction shown in Figure 2.12 implements this operation, assuming the availability of `load_linked/store_conditional`. Since C must have removed Y before modifying X, if B reads X's state before C changes Y, then the value read must be X's state from before C changed X. Thereafter, if X is changed, the *store-conditional* will force B to recheck Y. Alternative solutions might rely on a tracing garbage collector (which would decline to recycle X as long as B has a reference), on RCU-style grace periods [MAK01], or on manual reference-tracking methodologies [HLM02; Mic04].

2.5.3 Time and Space Bounds

This subsection provides an informal analysis of time and space requirements for the MCS-TP and CLH-TP locks. Figure 2.8 provides an overview summary of worst- and common case processor time steps for timing out and lock handoff, as well as per-lock queue length and total memory requirements.

MCS-TP bounds

We first consider space management in MCS-TP. Because no thread can ever have more than one node in the queue, the queue length is trivially linear in the number of threads T. A thread cannot reuse a node for another lock until that node is first removed from the previous lock's queue. This gives a worst-case space consumption for L locks of $O(T \times L)$. However, since lock holders clean up timed-out nodes during lock handoff, a thread will rarely have more than a small constant number of allocated nodes; this suggests that the space requirement will be closer to $O(T+L)$ in the common case.

To time out, a waiting thread must update its node's state from `waiting` to `left`. It must also reclaim its node if removed from the queue by the lock holder. Both operations require a constant number of steps, so the overall time requirement for leaving is $O(1)$.

As discussed in Section 2.5.1, the MCS-TP lock holder removes at most T nodes from the queue before switching to a scan. Since each removal and each step of the scan can be done in $O(1)$ time, the worst case is that the lock holder removes T nodes and then scans through T more timed-out nodes before reaching the end of the queue. It then marks the queue empty and re-traverses the (former) queue to remove each node,

```
// atomic operation which saves the old value of
// swap_addr in set_addr, and swaps the new_ptr
// into the swap_addr.
bool clhtp_swap_and_set(
   clhtp_qnode *volatile *swap_addr,
   clhtp_qnode *new_ptr,
   clhtp_qnode *volatile *set_addr)
{
  unsigned long pred;
  repeat
    pred = LL(swap_addr);
    (*set_addr) = (clhtp_qnode *) pred;
  while (0 == SC(swap_addr, new_ptr));
  return (clhtp_qnode *)pred;
}

// atomic compare and swap the tag in the pointer.
bool clhtp_tag_cas_bool(clhtp_qnode * volatile * p,
        unsigned long oldtag, unsigned long newtag)
{
   unsigned long oldv, newv;
   do {
      oldv = LL(p);
      if (get_tagbits(oldv) != oldtag)
         return false;
      newv = replace_tag(oldv, newtag);
   } while (!SC(p, newv));
   return true;
}

bool clhtp_rcas_bool(
   clhtp_qnode *volatile *stateptr,
   clhtp_qnode *volatile *ptr,
   clhtp_qnode *oldp,
   unsigned long newv)
{
   unsigned long oldv = (unsigned long)oldp;
   do {
      unsigned long tmp = LL(ptr);
      if (get_tagbits(*stateptr) != WAITING)
         return false;
      if (tmp != oldv)
         return true;
      } while (0 == SC(ptr, newv));
   return true;
}

void clhtp_failure_epilogue(
   clhtp_lock *L, clhtp_qnode *I)
{
   if (I->prev == SELFRC ||
      !clhtp_tag_cas_bool(&I->prev, PTIMEOUT, SUCRC)) {
      free_clhtp_qnode(I);
   }
}
```

Listing 2.13: Source code for the CLH-TP lock.

```
void clhtp_success_epilogue(clhtp_lock *L,
   clhtp_qnode *I, clhtp_qnode *pred)
{
   L->lock_holder = I;
   L->cs_start_time = CUR_TIME;
   free_clhtp_qnode(pred);
}

bool clhtp_acquire(clhtp_lock *L, hrtime_t T)
{
   clhtp_qnode *I = alloc_qnode();
   clhtp_qnode *pred;

   I->time = CUR_TIME;
   pred = swap_and_set(&L->tail, I, &I->prev);
   if (pred->prev == AVAILABLE) {
      if (compare_and_store(&I->prev, pred, HOLDING)) {
         clhtp_success_epilogue(L, I, pred);
         return true;
      } else {
         clhtp_failure_epilogue(L, I);
         if (CUR_TIME - L->cs_start_time > MAX_CSTICKS)
            yield();
         return false;
      }
   }

   bool result = clhtp_acquire_slow_path(L, T, I, pred);
   if (!result)
      if (CUR_TIME - L->cs_start_time > MAX_CSTICKS)
         yield();
   return result;
}

bool clhtp_acquire_slow_path(clhtp_lock *L, hrtime_t T,
        clhtp_qnode * I,clhtp_qnode * pred)
{
   hrtime_t my_start_time, current, pred_time;

   my_start_time = I->time;
   pred_time = pred->time;

   while (true) {
     clhtp_qnode *pred_pred;
     current = gethrtime();
     I->time = current;
     pred_pred  = pred->prev;

     if (pred_pred == AVAILABLE) {
        if (compare_and_store(&I->prev, pred, HOLDING))
           goto label_success;
        goto label_failure;
     } else if (pred_pred == SELFRC)
        goto label_self_rc;
     else if (pred_pred == HOLDING or INITIAL)
        goto check_self;
```

Listing 2.13: (continued)

```
     else {
       clhtp_qnode *pp_ptr = get_ptr(pred_pred);
       unsigned int pred_tag = get_tagbits(pred_pred);

       if (pred_tag == SUCRC) {
          if (!CAS_BOOL(&I->prev, pred, pp_ptr))
             goto label_failure;
          free_clhtp_qnode(pred);
          pred = pp_ptr;
          pred_time = pred->time;
          continue;
       }

       else if (pred_tag == PTIMEOUT) {
          if (!compare_and_store(&I->prev, pred, pp_ptr))
             goto label_failure;
          if (!compare_and_store(&pred->prev,
             pred_pred, SELFRC))
             free_clhtp_qnode(pred);
          pred = pp_ptr;
          pred_time = pred->time;
          continue;
       }

       else if (pred_tag == WAITING) {
          if (CUR_TIME - pred_time - UPDATE_DELAY >
             current) {
             if (pred->time != pred_time) {
                pred_time = pred->time;
                continue;
             } else if (clhtp_rcas_bool(
                &I->prev, &pred->prev, pred_pred,
                tagged_wptr(pred_pred, PTIMEOUT)))
                continue;
             }
          }
       }

check_self:
      unsigned int my_tag;
      pred = I->prev;
      if (pred == SELFRC)
         goto label_self_rc;
      my_tag = get_tagbits(pred);
      if (my_tag == PTIMEOUT)
         goto label_failure;
      else if (my_tag == WAITING) {
         if (CUR_TIME - my_start_time - T > current)
            goto label_self_timeout;
      }
   }
```

Listing 2.13: (continued)

```
label_success:
   clhtp_success_epilogue(L, I, pred);
   return true;

label_failure:
label_self_rc:
   clhtp_failure_epilogue(L, I);
   return false;

label_self_timeout:
   if (!compare_and_store(&I->prev, pred,
      tagged_wptr(pred, SUCRC))) {
      clhtp_failure_epilogue(L, I);
      return false;
   }
}

void clhtp_try_release(clhtp_lock *L)
{
   clhtp_qnode *I = L->lock_holder;
   I->prev = AVAILABLE;
}
```

Listing 2.13: (continued)

for a total of $O(T)$ steps. In the common case a thread's immediate successor is not preempted, allowing handoff in $O(1)$ steps.

CLH-TP bounds

In our implementation, the CLH-TP lock uses a timeout protocol in which it stops publishing updated timestamps $k\mu s$ before its patience has elapsed, where k is the staleness bound for deciding that a thread has been preempted. [4] Further, so long as a thread's node remains `waiting`, the thread continues to remove timed-out and preempted predecessors. In particular, a thread only checks to see if it has timed out if its predecessor is active.

A consequence of this approach is that thread A cannot time out before its successor B has had a chance to remove it for inactivity. If B is itself preempted, then any successor active before it is rescheduled will remove B's and then A's node; otherwise,

[4]For best performance, $k\mu s$ should be greater than the round-trip time for a memory bus or interconnect transaction on the target machine, plus the maximal pairwise clock skew observable between processors.

	Worst-case		Common case
	MCS-TP	CLH-TP	(both locks)
Timeout	$O(1)$	$O(T)$	$O(1)$
Lock handoff	$O(T)$	$O(1)$	$O(1)$
Queue length	$O(T)$	$O(T^2)$	$O(T)$
Total space	$O(T \times L)$	$O(T^2 \times L)$	$O(T + L)$

Figure 2.8: Time/space bounds for TP locks: T threads and L locks

B will remove A's node once rescheduled. This in turn implies that any pair of nodes in the queue abandoned by the same thread have at least one node between them that belongs to a thread that has not timed out. In the worst case, $T - 1$ nodes precede the first, suspended, "live" waiter, $T - 2$ precede the next, and so on, for a total of $O(T^2)$ total nodes in the queue.

As in MCS-TP, removing a single predecessor can be performed in $O(1)$ steps. As the queue length is bounded, so, too, is the timeout sequence. In contrast to MCS-TP, successors in CLH-TP are responsible for actively claiming the lock; a lock holder simply updates its state to show that it is no longer using the lock, clearly an $O(1)$ operation.

Since all waiting threads concurrently remove inactive nodes, it is unlikely that an inactive node will remain in the queue for long. In the common case, then, the queue length is close to the total number of threads currently contending for the lock. Since a thread can only contend for one lock at a time, we can expect common case space $O(T + L)$. Similarly, the average timeout delay is $O(1)$ if most nodes in the queue are actively waiting.

2.5.4 Scheduling and Preemption

TP locks publish timestamps to enable a heuristic that guesses whether the lock holder or a waiting thread is preempted. This heuristic admits a race condition wherein a thread's timestamp is polled just before it is descheduled. In this case, the poller will mistakenly assume the thread to be active. In practice (see Section 2.6), the timing window is too narrow to have a noticeable impact on performance. Nevertheless it is instructive to consider modified TP locks that use a stronger scheduler interface to completely eliminate preemption vulnerabilities.

Extending previous work [KWS97], we distinguish three levels of APIs for user-level feedback to the kernel scheduler implementation:

(I) Critical section protection (CSP): A thread can bracket a block of code to request that it not be preempted while executing inside.

(II) Runtime state check: A thread can query the status (running, preempted) of other threads.

(III) Directed preemption avoidance: A thread can ask the scheduler not to preempt a specified peer thread.

Several commercial operating systems, including Solaris and AIX 5L (which we use in our experiments [IBM01]) provide Level I APIs. Level II and III APIs are generally confined to research systems [ABL92; ELS88; MSL91]. The Mach scheduler [Bla90] provides a variant of the Level III API that includes a directed yield of the processor to a specified thread.

For TATAS locks, a Level I API is sufficient [KWS97] to create a variant that avoids preemption during the critical section of a lock holder. By comparison, a thread contending for a (non-timeout-capable) queue-based lock is sensitive to preemption in two additional timing windows—windows not addressed by the preemption tolerance of the MCS-TP and CLH-TP locks. The first window occurs in MCS-style locks between swapping a node into the queue's tail and connecting links with the remainder of the queue. The second occurs between when a thread is granted the lock and when it starts actually using the lock. We say that a lock is *preemption-safe* only if it prevents all such *timing windows*.

Previous work proposed two algorithms for preemption-safe MCS variants: the Handshaking and SmartQ locks [KWS97]. Both require a Level I API to prevent preemption in the critical section and in the first (linking-in) window described above. For the second (lock-passing) window, the lock holder in the Handshaking lock exchanges messages with its successor to confirm that it has invoked the Level I API. In practice, this transaction has undesirably high overhead (two additional remote coherence misses on the critical path), so SmartQ employs Level II and III APIs to replace it. We characterize the preemption safety of the Handshaking lock as *heuristic*, in the sense that a thread guesses the status of a successor based on the speed of its response, and may err on the side of withholding the lock if, for example, the successor's processor is momentarily busy handling an interrupt. By contrast, the preemption safety of the SmartQ lock is *precise*.

Our MCS-TP lock uses a one-way handoff transaction similar to, but simpler and faster than, that of the Handshaking lock. However, because of the reduced communication, the lock cannot be made preemption safe with a Level I API. By contrast, a preemption-safe CLH variant can be built efficiently from the CLH-TP lock. The tail-to-head direction of linking eliminates the first preemption window. The second is readily addressed if a thread invokes the Level I API when it sees the lock is available, but before updating its state to grab the lock. If the lock holder grants the lock to a preempted thread, the first active waiter to remove all inactive nodes between itself and the queue's head will get the lock. We call this clock CLH-CSP (critical section protection). Like the Handshaking lock, it is heuristically preemption safe. For precise preemption safety, one can use a Level II API for preemption monitoring (CLH-PM).

Note that TATAS locks require nothing more than a Level I API for (precise) preemption safety. The properties of the various lock variants are summarized in Figure 2.9. The differences among families (TATAS, MCS, CLH) stem mainly from the

This table lists minimum requirements for implementing NB/PT/PS capabilities. The Handshaking and SmartQ locks are from Kontothanassis et al. [KWS97]. "CSP" indicates use of a Level I API for critical section protection; "PM" indicates preemption monitoring with a Level II API; "try" indicates a timeout-capable lock. **NB**: Non-Blocking; **PT**: Preemption-Tolerant; **PS**: Preemption-Safe; **—**: unnecessary.

Support	TATAS	MCS	CLH
Atomic instructions	PT NB-try (TAS-yield)	standard lock (MCS)	standard lock (CLH)
NB timeout algorithms	—	NB-try (MCS-NB)	NB-try (CLH-NB)
TP algorithms	—	PT NB-try (MCS-TP)	PT NB-try (CLH-TP)
Level I API	precise PS (TAS-CSP)	heuristic PS (Handshaking)	heuristic PS (CLH-CSP)
Level II API	—	—	precise PS (CLH-PM)
Level III API	—	precise PS (SmartQ)	—

Figure 2.9: Preemption tolerance in families of locks (using the classification from Section 2.5.4 for required kernel support)

style of lock transfer. In TATAS locks, the opportunity to acquire an available lock is extended to all comers. In the MCS locks, only the current lock holder can determine the next lock holder. In the CLH locks, waiting threads can pass preempted peers to grab an available lock, though they cannot bypass active peers.

2.6 Experimental Results

Using the atomic operations available on three different platforms, we have implemented nine different lock algorithms: TATAS, TATAS-try, CLH, CLH-NUMA, CLH-try, CLH-NB-try [Sco02], MCS, MCS-try, and MCS-NB-try. We have tested on large multiprocessors based on three different processor architectures: Sun E25000 and Sun-Fire 6800 servers based on the SPARC V9 instruction set, an IBM p690 based on the Motorola Power4 instruction set, and a Cray T3E based on the Alpha 21264 instruction set and Cray `shmem` libraries. On the IBM p690, we have additionally implemented the CLH-TP and MCS-TP locks (these locks require access to the `load_linked/ store_conditional` instruction pair, so require additional software infrastructure on the other platforms). Our main test employs a microbenchmark consisting of a tight loop containing a single acquire/release pair and optional timed "busywork" inside and outside the critical section. In this benchmark, a "fuzz factor" optionally adds a small random factor to the busywork and to the patience used for try locks.

`Acquire` and `release` operations are implemented as in-line subroutines wherever feasible. Specifically: For CLH, CLH-NUMA, and MCS we in-line both `acquire` and `release`. For TATAS, TATAS-try, and CLH-try we in-line `release` and the "fast path" of `acquire` (with an embedded call to a true subroutine if the lock is not immediately available). For MCS-try we in-line fast paths of both `acquire` and `release`. For CLH-NB-try, MCS-NB-try, CLH-TP, and MCS-TP, the need for dynamic memory allocation forces us to implement both `acquire` and `release` as true subroutines.

We present performance results from a diverse collection of machines: a 144-way Sun E25000 multiprocessor with 1.35 GHz UltraSPARC III processors, a 32-way IBM p690 multiprocessor with 1.3 GHz Power4 processors, and a 512-way Cray T3E with 600 MHz Alpha 21264 processors. We additionally present limited results from a 64-way Sun E10000 and a 16-way SunFire 6800. In each case, assignment of threads to processors was left to the operating system, and the machine was otherwise unloaded. Code was compiled with the maximum level of optimization supported by the various compilers, but was not otherwise hand-tuned.

2.6.1 Single-processor results

We can obtain an estimate of lock overhead in the absence of contention by running the microbenchmark on a single processor, and then subtracting out the loop overhead. Results, calculated in hardware cycles, appear in Figure 2.10; these were collected on a 64-processor Sun Enterprise 10000. In an attempt to avoid perturbation due to other activity on the machine (such as invocations of kernel daemons), we have reported the minima over a series of 8 runs. Data in this chart is reproduced from Scott's 2002 PODC paper [Sco02]; the newer version of MCS-NB-try that we present here did not exist yet. The times presented for the CLH-NB-try and MCS-NB-try locks include dynamic allocation and deallocation of queue nodes.

	cycles	atomic ops	reads	writes
TATAS	19	1	0	1
TATAS-try	19	1	0	1
CLH	35	1	3	4
CLH-try	67	2	3	3
CLH-NB try	75	2	3	4
MCS	59	2	2	1
MCS-try	59	2	2	1
MCS-NB-try (original)	91	3	3	4
MCS-NB-try (current)	na	4	2	3

Figure 2.10: Single-processor spin-lock overhead: Sun E10000

As one might expect, none of the queue-based locks is able to improve upon the time of the TATAS lock. The plain CLH lock, which in our early studies was able to match TATAS, now performs almost twice as slowly, due to the increasingly disparity in run-time overhead between atomic and regular instructions. The relatively high overhead we see for nonblocking timeout locks reflects the lack of inlined acquire and release functions.

The importance of single-processor overhead can be expected to vary from application to application. It may be significant in a database system that makes heavy use of locks, so long as most threads inspect independent data, keeping lock contention low. For large scientific applications, on the other hand, Kumar et al. [KJC99] report that single-processor overhead—lock overhead in general—is dwarfed by waiting time at contended locks, and is therefore not a significant concern.

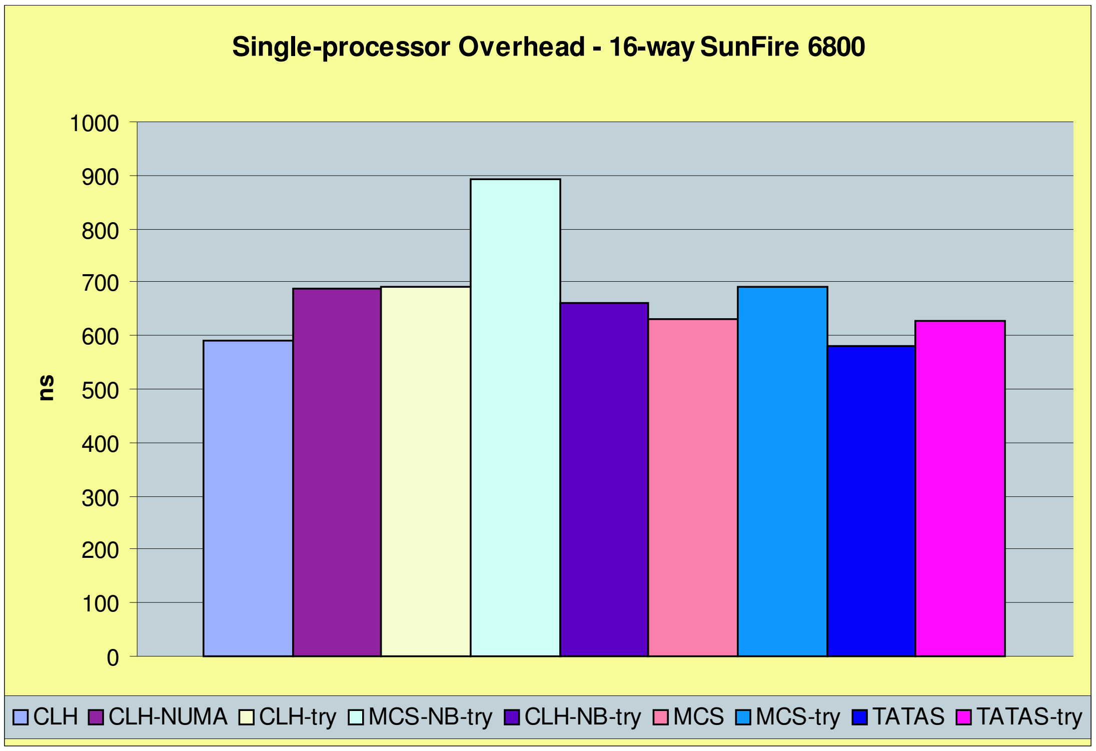


Figure 2.11: Single processor lock overhead on a SunFire 6800

Although the single-processor results in Figure 2.10 show wide disparity, this is largely because all non-lock overhead has been removed. By way of comparison, we present in Figure 2.11 single-processor overhead in combination with modest critical and non-critical sections (25 and 50 repetitions of an idle loop, respectively) on our SunFire 6800. The same general performance trends manifest as before, but the differences in magnitude are greatly diminished. That the performance advantage of the TATAS lock is considerably reduced relative to the E10000 results is probably attributable to the deprecation of the `test_and_set` instruction that we used in the SPARC v9 architecture.

2.6.2 Overhead on multiple processors

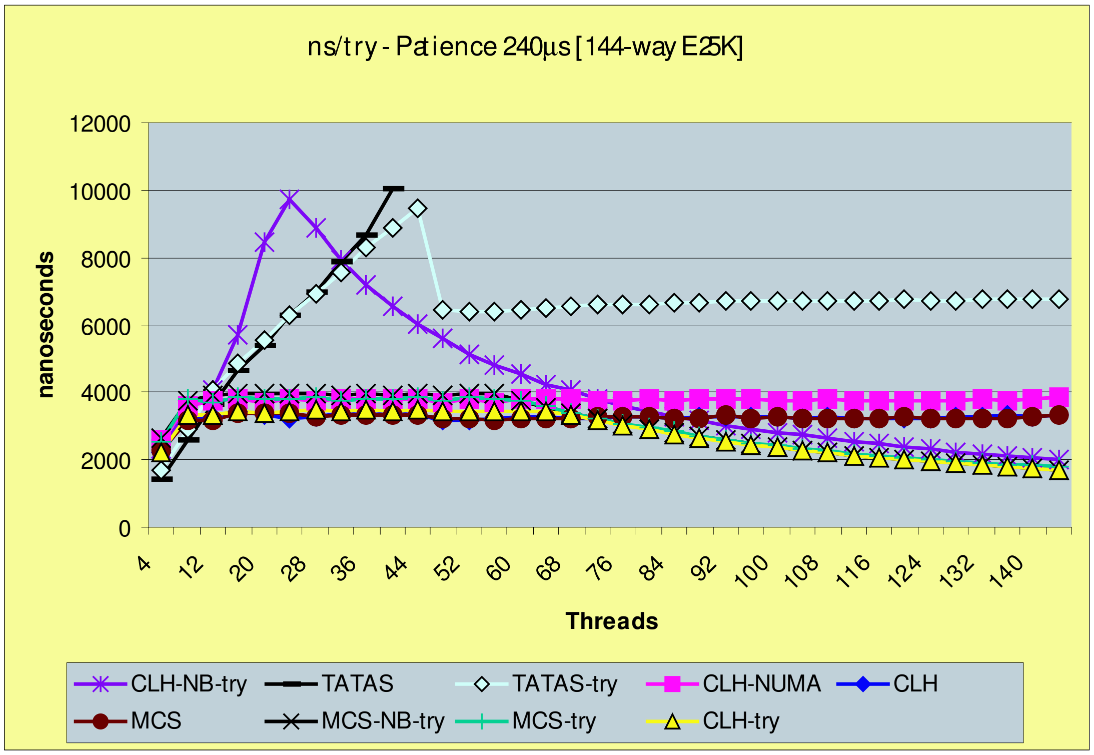
We can obtain an estimate of the time required to pass a lock from one processor to another by running our microbenchmark on a large collection of processors. This *passing time* is not the same as total lock overhead: As discussed by Magnussen, Landin, and Hagersten [MLH94], queue-based locks tend toward heavily pipelined execution, in which the initial cost of entering the queue and the final cost of leaving it are overlapped with the critical sections of other processors.

Figure 2.12 shows lock behaviors on the Sun E25000, with timeout threshold (patience) of $240\mu s$, non-critical busywork corresponding to 50 iterations of an empty loop, and critical section busywork corresponding to 25 iterations of the loop.

With this patience level, the various try locks exhibit distinctly bimodal behavior. Up through 68 active processors, timeout almost never occurs, and lock behavior mimics that of the non-try CLH and MCS locks. With more active processors, timeouts begin to occur.

For higher processor counts, as for lower patience levels, the chance of a processor getting a lock is primarily a function of the number of processors that are in the queue ahead of it minus the number of those that time out and leave the queue before obtaining the lock. As is evident in these graphs, when patience is insufficient, this chance drops off sharply. The average time per try also drops, because giving up is cheaper than waiting to acquire a lock.

Comparing individual locks, we see that the best performance up through 12 processors is obtained with the TATAS locks, though the gap is fairly small. Thereafter, MCS and CLH give roughly equivalent performance, with CLH-try close behind. CLH-NUMA and the remaining try locks all cluster just behind. Finally, the TATAS-try and TATAS locks display a considerable performance penalty at these processor counts. (The poor showing of CLH-NB-try at these processor counts is an artifact of a bug that we were unable to track down before losing access to the test machine.)



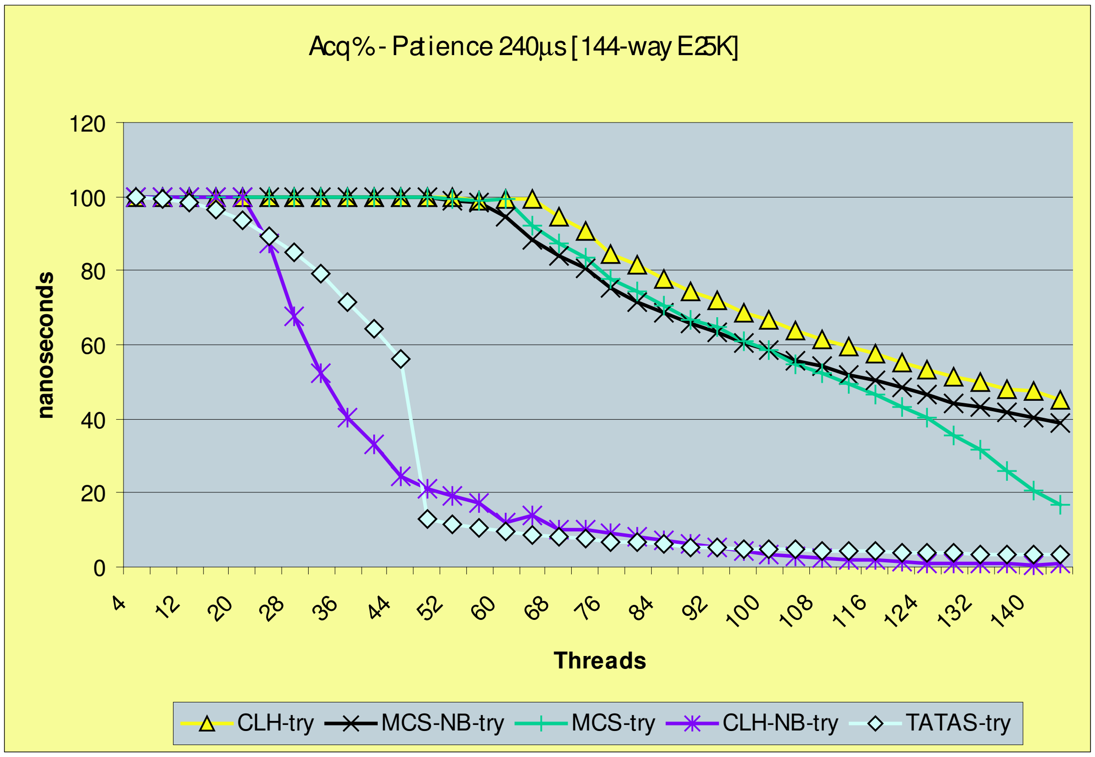


Figure 2.12: Queue-based lock performance: 144-processor Sun E25000. Microbenchmark net iteration time (top) and success rate (bottom) with patience $240\mu s$, non-critical busywork of 50 repetitions of an idle loop, and critical busywork of 25 repetitions of an idle loop

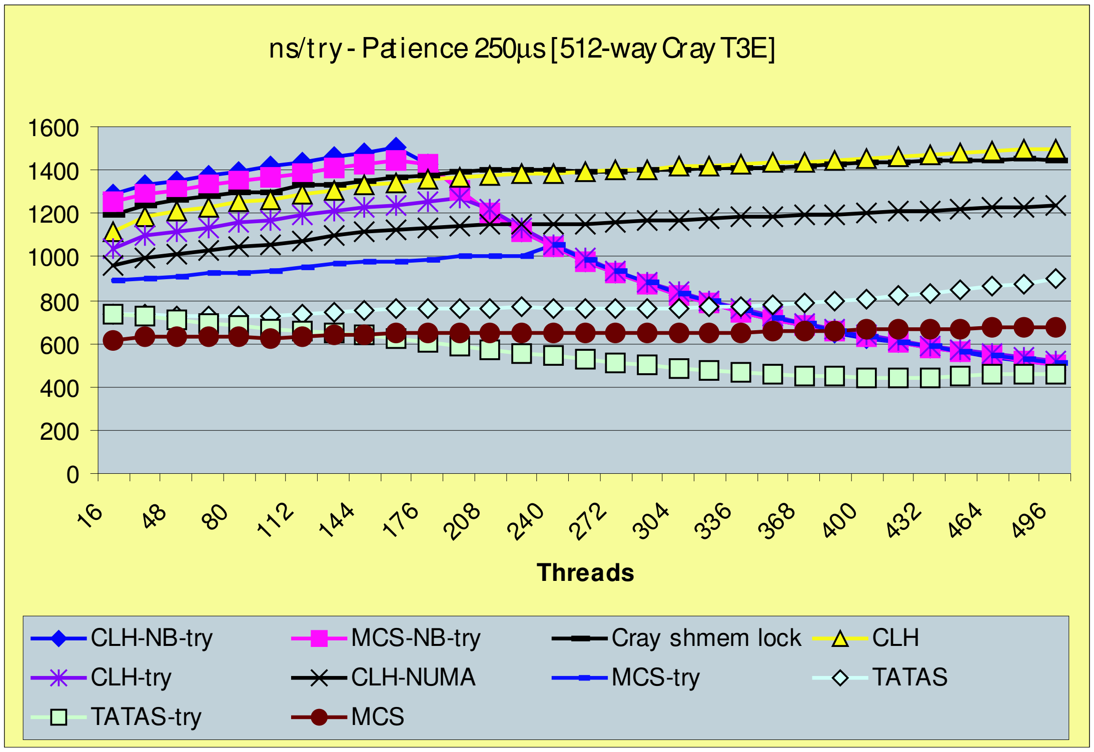
The tradeoffs among MCS-NB-try, MCS-try, and plain MCS are as expected: At the cost of a higher average iteration time (per attempt), the plain MCS lock always manages to successfully acquire the lock. At the cost of greater complexity, the MCS-try lock provides the option of timing out. Dynamic allocation of queue nodes incurs a small amount of additional overhead in the MCS-NB-try lock. Similar tradeoffs hold among the CLH, CLH=try, and CLH-NB-try locks.

The tradeoffs between try locks and TATAS are also interesting. While iteration time is consistently higher for the queue-based locks, the acquisition (success) rate depends critically on the ratio between patience and the level of competition for the lock. When patience is high, relative to competition, the queue-based locks are successful all the time. Once the expected wait time exceeds the timeout interval in the queue-based locks, however, the TATAS lock displays a higher acquisition rate. In these executions, however, the same processor tends to repeatedly reacquire a TATAS lock; the queue-based locks are much more consistently fair.

Figure 2.13 shows lock behaviors on the Cray T3E, with patience of $250\mu s$, non-critical busywork corresponding to 50 iterations of an empty loop), and critical section busywork corresponding to 25 iterations of the loop. Figure 2.14 shows that (with sufficient patience of $750\mu s$) the locks scale well to the full extent of the Cray machine. (The gradual increase in acquisition time reflects the increasing diameter of the hypertorus interconnect between processors.)

Where the E25000 shows relatively little difference between the MCS and CLH lock families, the Cray T3E results are more pronounced. For example, the plain MCS lock is twice as fast as the plain CLH. This is because the T3E, while cache-coherent, does not cache remote memory locations. [5] Hence, locks in the MCS family, which

[5]In fact, by default, pointers on the Cray T3E refer explicitly to memory in the local node. Porting the various locks to it required creating higher-level pointers that added information about the host node, and implementing a "portability layer" that invoked calls to Cray's shmem libraries with these global pointers. We were fortunate that the addressing limits of the Alpha processors left room for us to steal enough bits to represent an operand node.



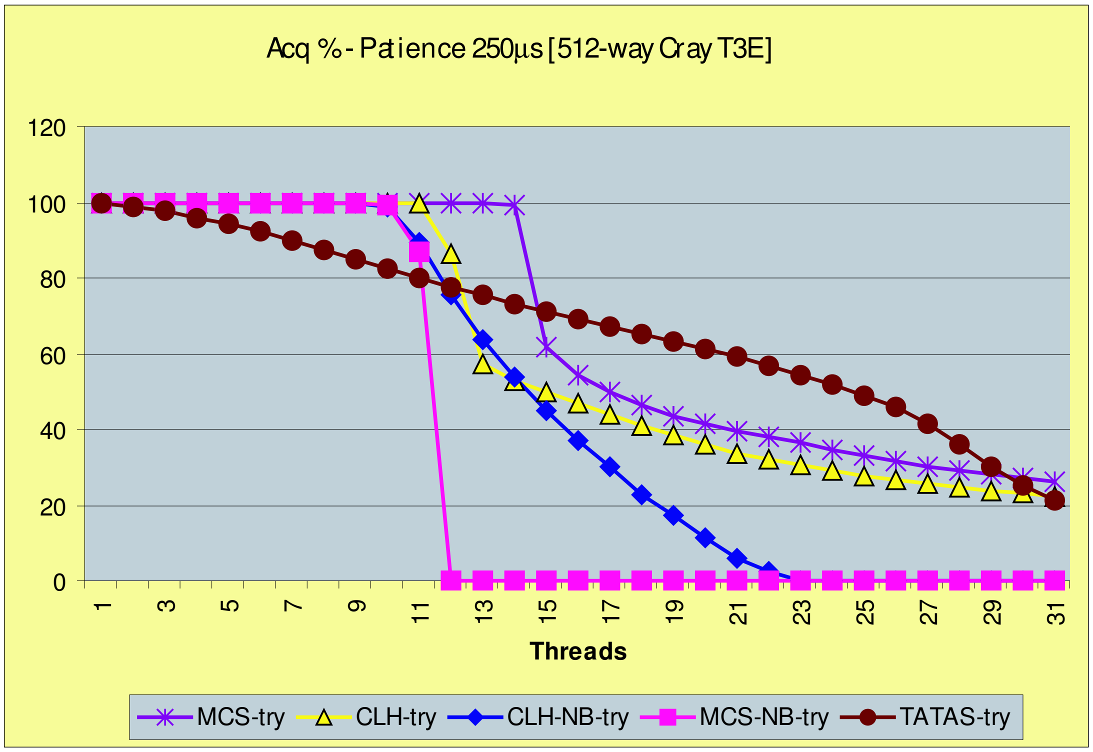


Figure 2.13: Queue-based lock performance: 512-processor Cray T3E. Microbenchmark net iteration time (top) and success rate (bottom) with patience $250\mu s$, non-critical busywork of 50 repetitions of an idle loop, and critical busywork of 25 repetitions of an idle loop

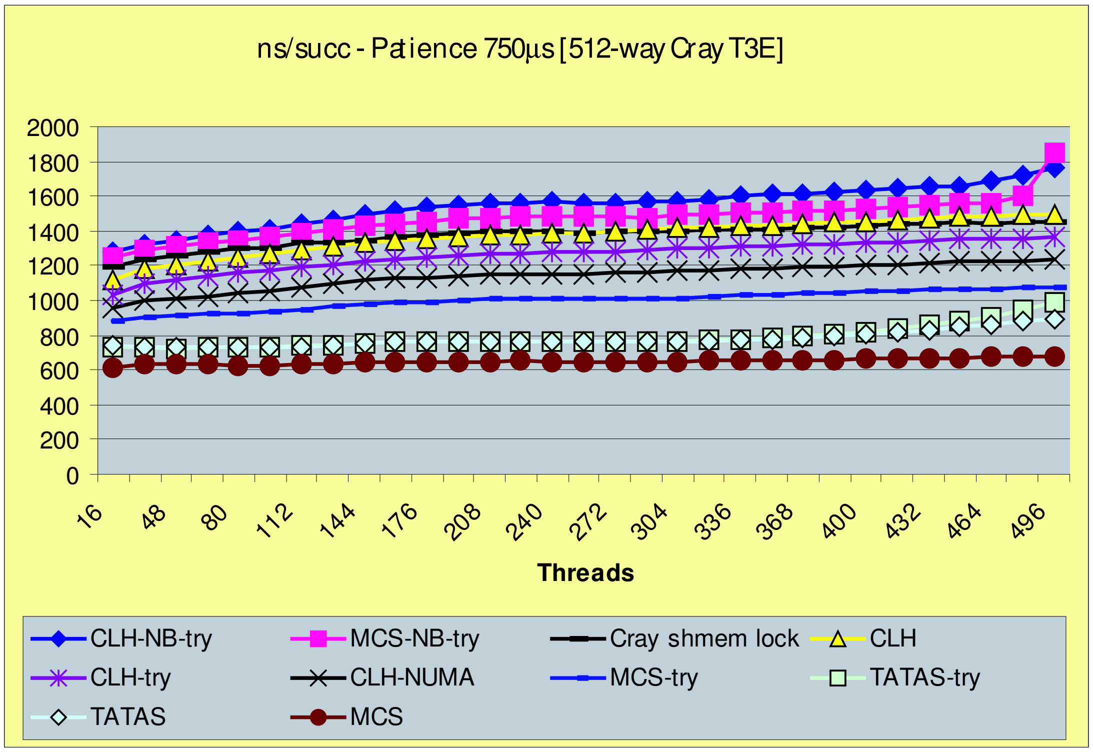
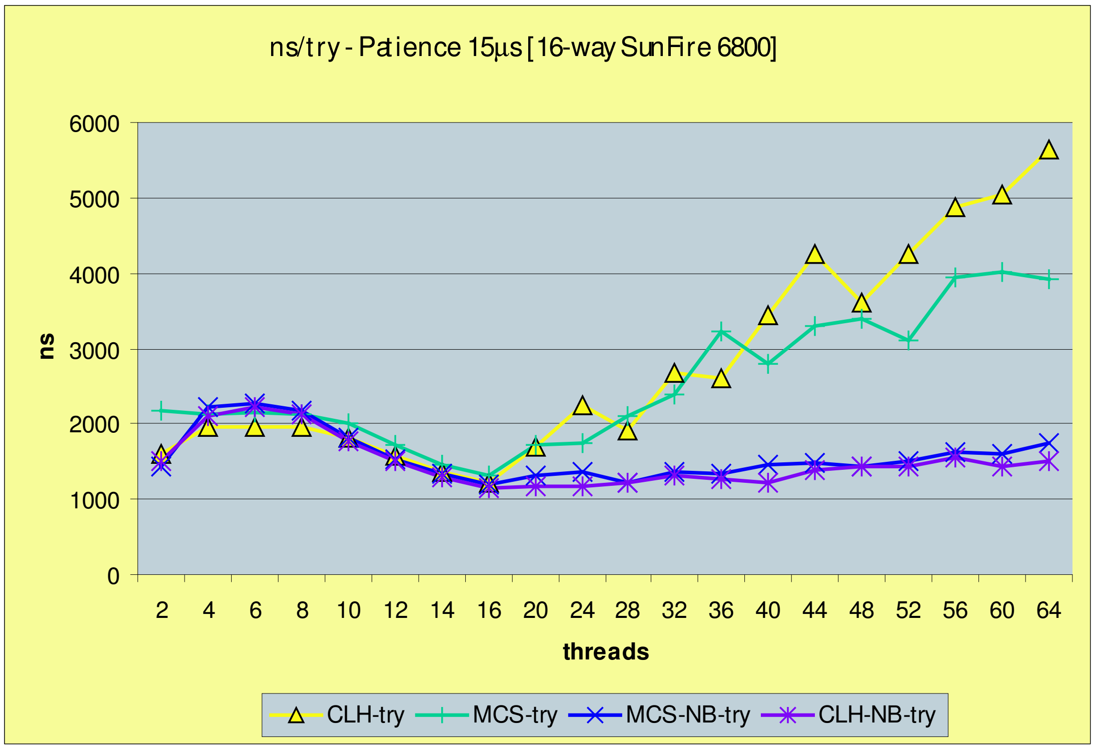


Figure 2.14: Scalability of queue-based locks on the Cray T3E at patience $750\mu s$.

arrange to spin on a memory location local to the waiting processor, incur far less interconnect traffic and outperform their CLH counterparts by considerable margins. Given this characterization, it seems surprising that Cray's default shmem lock appears to be based on an algorithm that mirrors the performance of the CLH algorithm.

2.6.3 Adding preemption

Following Scott's example [Sco02], we present results on a heavily multiprogrammed system: up to 64 threads on the 16-processor SunFire 6800. By increasing the total number of threads, the various windows of preemption vulnerability discussed in Section 2.5 – the critical section, in the queue, during timeout – are increasingly likely to be hit; not surprisingly, the nonblocking timeout locks give far better preemption tolerance than do their blocking-timeout counterparts.



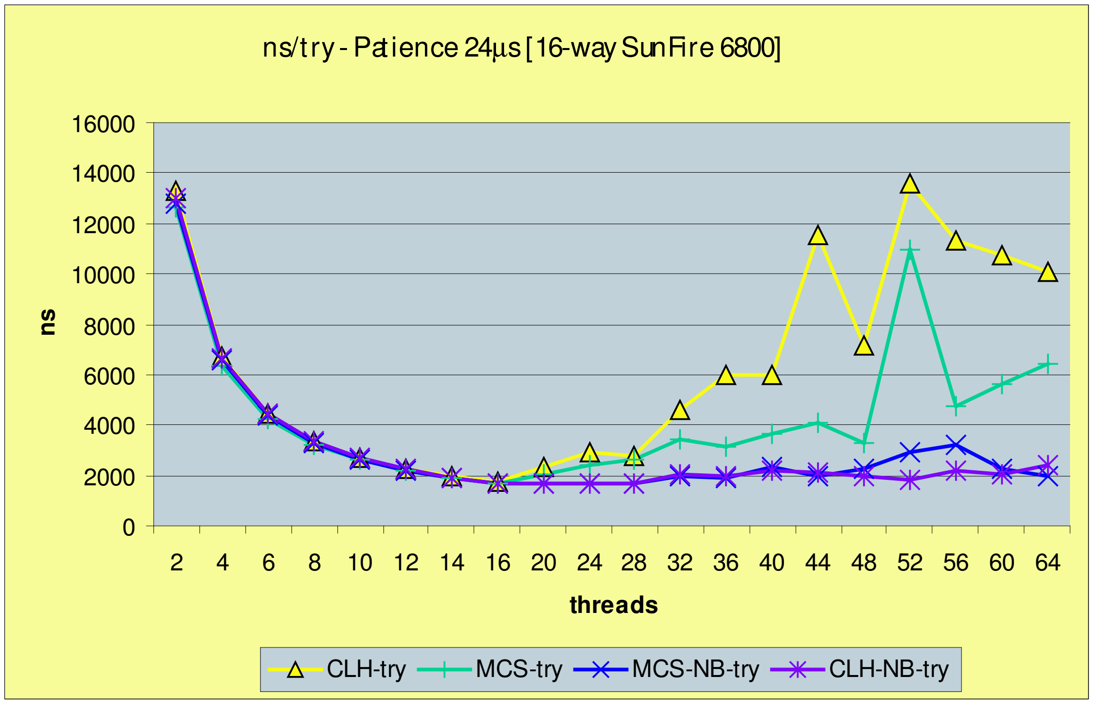


Figure 2.15: The impact of preemption in queue-based locks on a 16-processor SunFire 6800: Acquirable (top) and unacquirable (bottom) locks.

Figure 2.15 plots iteration times for our preemption sensitivity experiment. Results were averaged over three runs. As can be clearly seen, the cumulative effect of hitting various windows of preemption vulnerability directly affects the time overhead spent on each attempt to acquire the lock; locks with nonblocking timeout suffer from fewer such windows and thus maintain better performance.

2.6.4 Time-published locks

Although nonblocking timeout helps get threads out of the queue for a lock, it does not tend to improve overall lock utilization with high levels of preemption. As discussed in Section 2.5, our TP locks attack the core of this problem. We tested these TP locks on an IBM pSeries 690 (Regatta) multiprocessor. For comparison purposes, we included a range of user-level spin locks: TATAS, MCS, CLH, MCS-NB, and CLH-NB. We also tested spin-then-yield variants [KLM91] of each lock in which threads yield the processors after exceeding a wait threshold.

Finally, we tested preemption-safe locks dependent on scheduling control APIs: CLH-CSP, TAS-CSP, Handshaking, CLH-PM, and SmartQ. TAS-CSP and CLH-CSP are TATAS and CLH locks augmented with critical section protection (the Level I API discussed in Section 2.5.4). The Handshaking lock [KWS97] also uses CSP. CLH-PM adds the Level II preemption monitoring API to assess the preemptive state of threads. The SmartQ lock [KWS97] uses all three API Levels.

Our p690 runs AIX 5.2. Since this system provides no scheduling control APIs, we have also implemented a synthetic scheduler similar to that used by Kontothanassis et al. [KWS97]. This scheduler runs on one dedicated processor, and sends Unix signals to threads on other processors to mark the end of each $20\,ms$ quantum. If the receiving thread is preemptable, that thread's signal handler spins in an idle loop to simulate execution of a compute-bound thread in some other application; otherwise, preemption

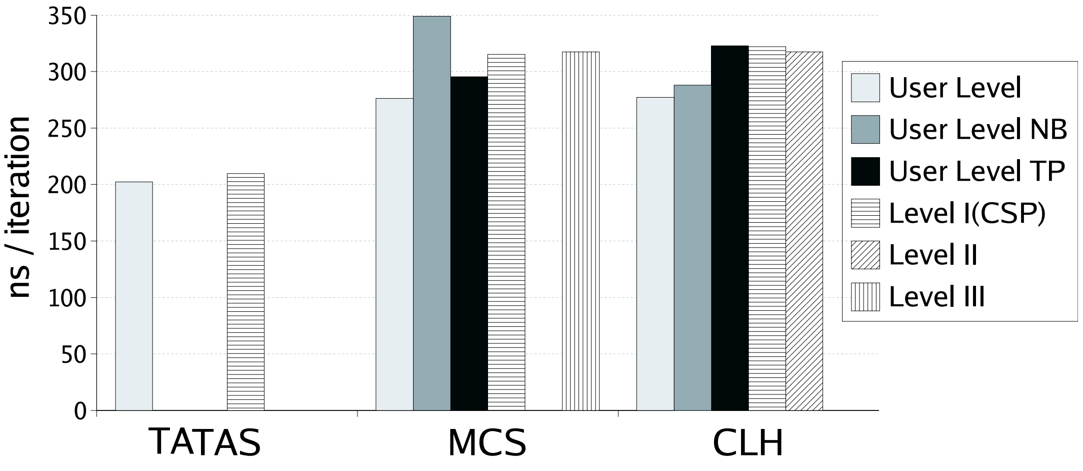


Figure 2.16: Single-processor spin-lock overhead: IBM p690

is deferred until the thread is preemptable. Our synthetic scheduler implements all three Levels of scheduling control APIs.

Our microbenchmark application has each thread repeatedly attempt to acquire a lock. We simulate critical sections (CS) by updating a variable number of cache lines; we simulate non-critical sections (NCS) by varying the time spent spinning in an idle loop between acquisitions. We measure the total throughput of lock acquisitions and we count successful and unsuccessful acquisition attempts, across all threads for one second, averaging the results of 6 runs. For try locks, we retry unsuccessful acquisitions immediately, without executing a non-critical section. We use a fixed patience of $50\,\mu s$.

Single Thread Performance

Because low overhead is crucial for locks in real systems, we assess it by measuring throughput absent contention with one thread and empty critical and non-critical sections. We organize the results by lock family in Figure 2.16.

As expected, the TATAS variants are the most efficient for one thread. MCS-NB has one `compare_and_swap` more than the base MCS lock; this appears in its single-thread overhead. Similarly, other differences between locks trace back to the operations in their `acquire` and `release` methods. One difference between the results for this machine and those for the Sun E10000 in Figure 2.10 is that the CLH and MCS are comparable in cost on the p690 where MCS was half again as expensive on the Sun. This suggests that the difference in cost between an atomic instruction and a remote read that misses in cache is far smaller on the p690 than on the E10000. We note that time-publishing functionality adds little overhead to locks.

A single-thread atomic update on our p690 takes about $60\,ns$. Adding additional threads introduces delays from memory and processor interconnect contention, and from cache coherence overhead when transferring a cache line between processors. We have measured overheads for an atomic update at 120 and $420\,ns$ with 2 and 32 threads.

Comparison to User-Level Locks

Under high contention, serialization of critical sections causes application performance to depend primarily on the overhead of handing a lock from one thread to the next; other overheads are typically subsumed by waiting. We present two configurations for critical and non-critical section lengths that are representative of large and small critical sections.

Our first configuration simulates contention for a small critical section with a 2-cache-line-update critical section and a $1\,\mu s$ non-critical section. Figure 2.17 plots the performance of the user-level locks with a generic kernel interface (no scheduler control API). Up through 32 threads (our machine size), queue-based locks outperform TATAS; however, only the TP and TATAS locks maintain throughput in the presence of preemption. MCS-TP's overhead increases with the number of preempted threads because it relies on the lock holder to remove nodes. By contrast, CLH-TP distributes cleanup work across active threads and keeps throughput more steady. The right-hand graph in Figure 2.17 shows the percentage of successful lock acquisition attempts for the try locks. MCS-TP's increasing handoff time forces its success rate below that

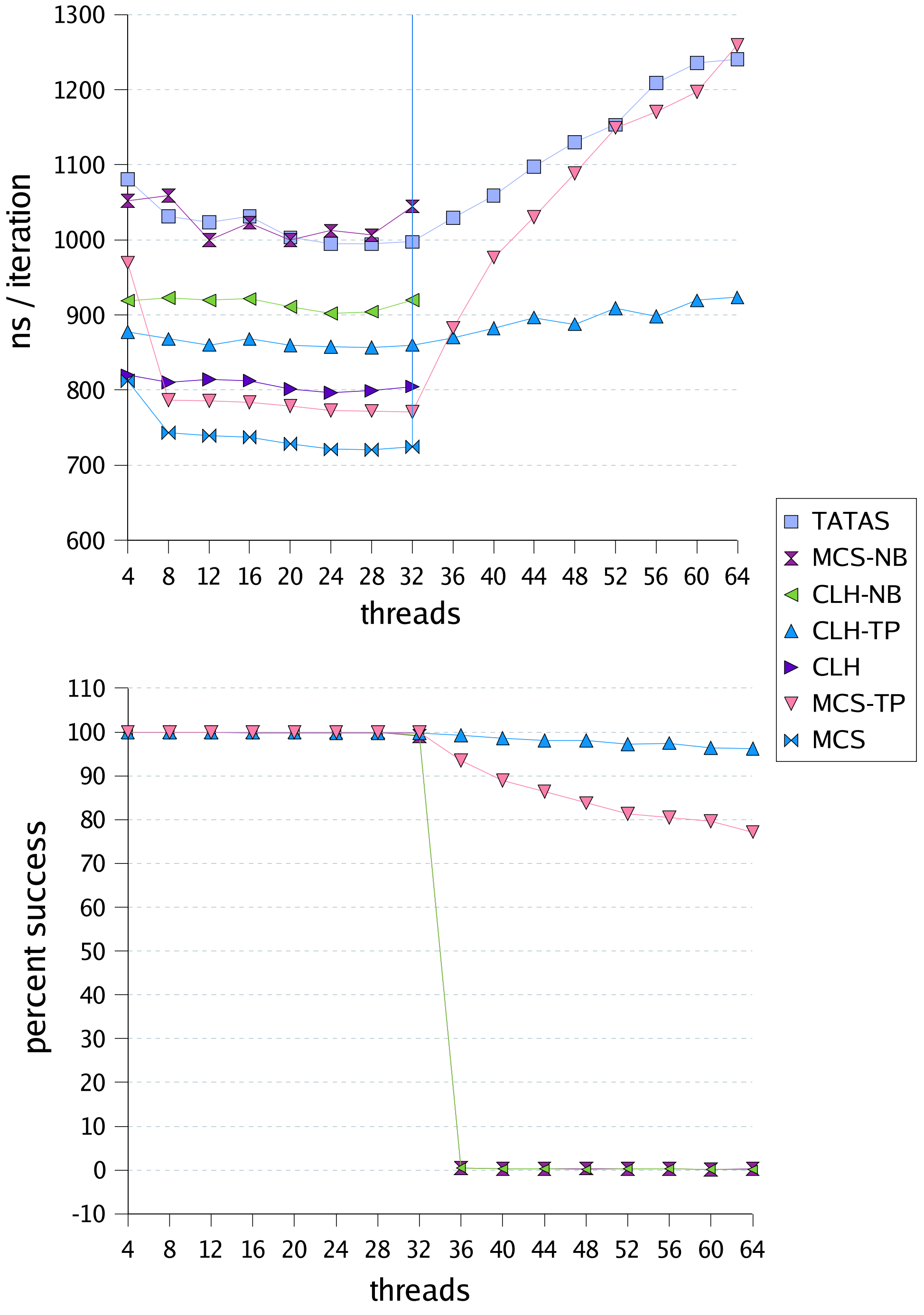


Figure 2.17: Queue-based lock performance: IBM p690 (small critical sections).
2 cache line-update critical section (CS); 1 μs non-critical section (NCS).
Critical section service time (top) and success rate (bottom)

of CLH-TP as the thread count increases. CLH-NB and MCS-NB drop to almost no successes due to preemption while waiting in the queue.

Our second configuration uses 40-cache-line-update critical sections and $4\,\mu s$ non-critical sections. It models longer operations in which preemption of the lock holder is more likely. Figure 2.18 shows the behavior of user-level locks with this configuration. That the TP locks outperform TATAS demonstrates the utility of cooperative yielding for preemption recovery. Moreover, the gap in performance between CLH-TP and MCS-TP is smaller here than in our first configuration: The relative importance of removing inactive queue nodes goes down compared to that of recovering from preemption in the critical section.

In Figure 2.18, the success rates for try locks drop off beyond 24 threads. Since each critical section takes about $2\,\mu s$, our $50\,\mu s$ patience is just enough for a thread to sit through 25 predecessors. TP locks adapt better to insufficient patience.

One might expect a spin-then-yield policy [KLM91] to allow other locks to match TP locks in preemption tolerance. In Figure 2.19 we test this hypothesis with a $50\,\mu s$ spinning time threshold and a 2 cache line critical section. (Other settings produce similar results.) Although yielding improves the throughput of non-TP queue-based locks, they still run off the top of the graph. TATAS benefits enough to become competitive with MCS-TP, but CLH-TP still outperforms it. These results confirm that targeted removal of inactive queue nodes is much more valuable than simple yielding of the processor.

Comparison to Preemption-Safe Locks

For this section, we used our synthetic scheduler to compare TP and preemption-safe locks. Results for short critical sections are shown in Figure 2.20, both with (multiprogrammed mode) and without (dedicated mode) an external 50% load.

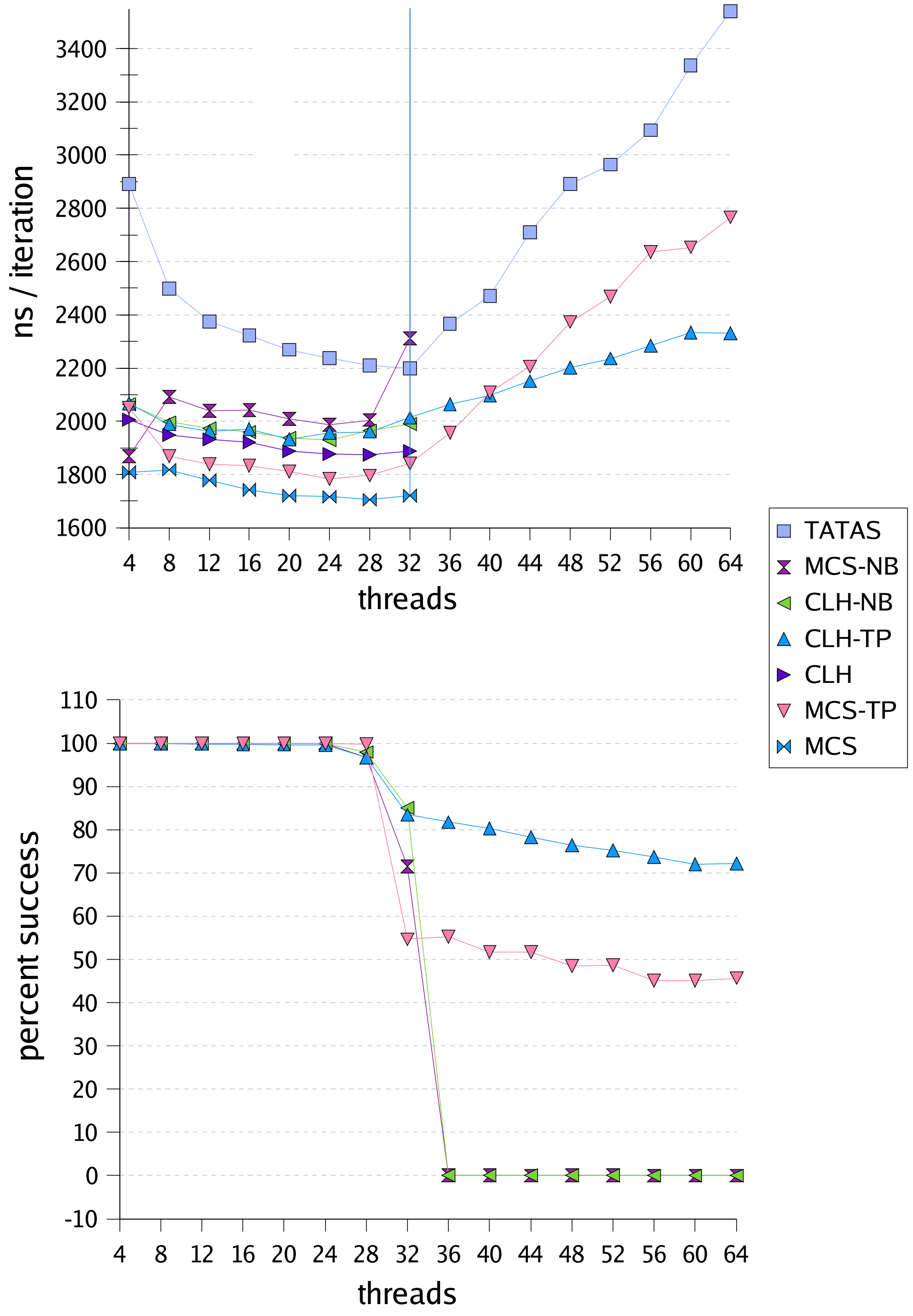


Figure 2.18: Queue-based lock performance: IBM p690 (large critical sections). 40 cache line CS; 4 μs NCS. Critical section service time (top) and success rate (bottom)

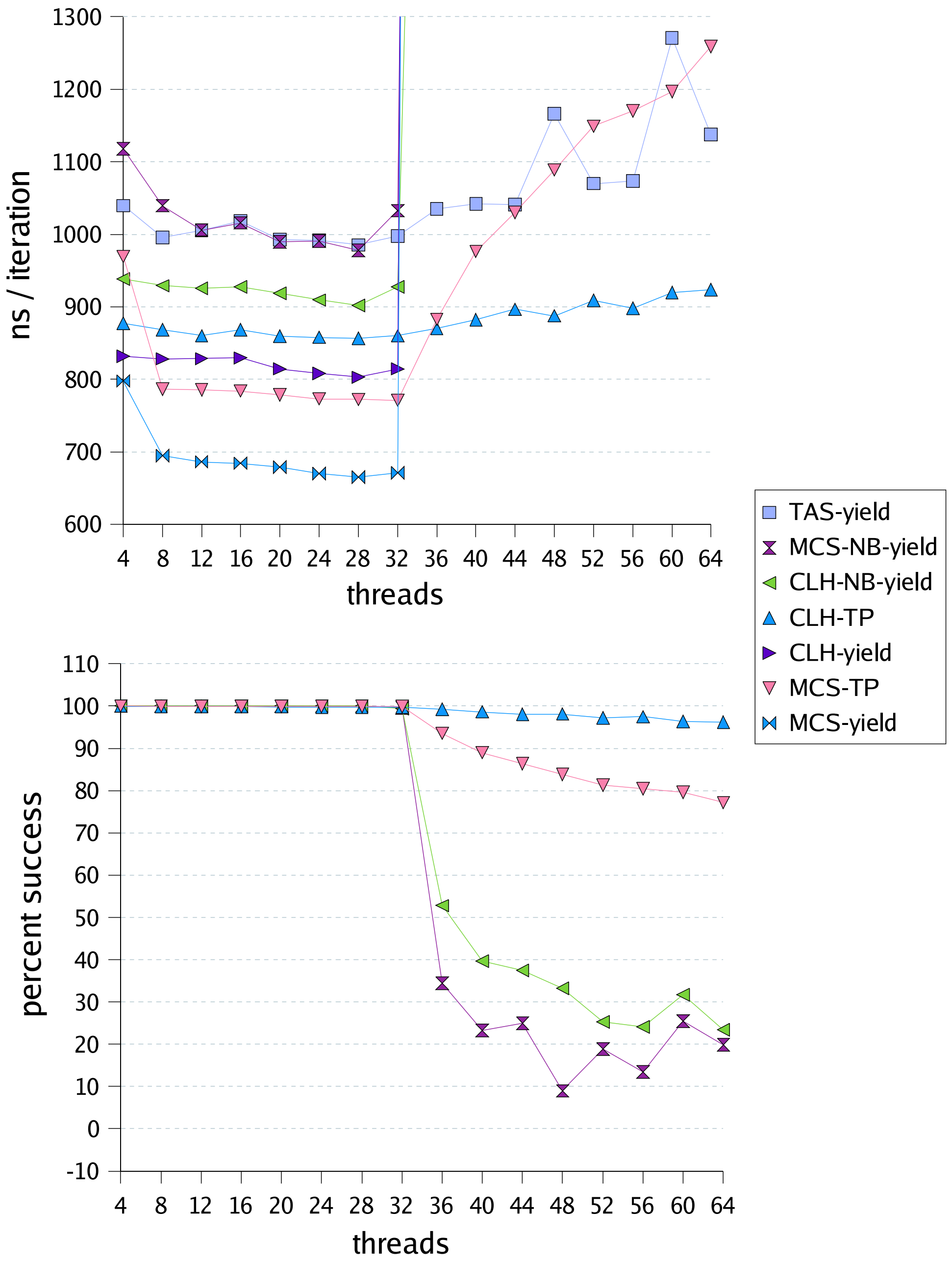
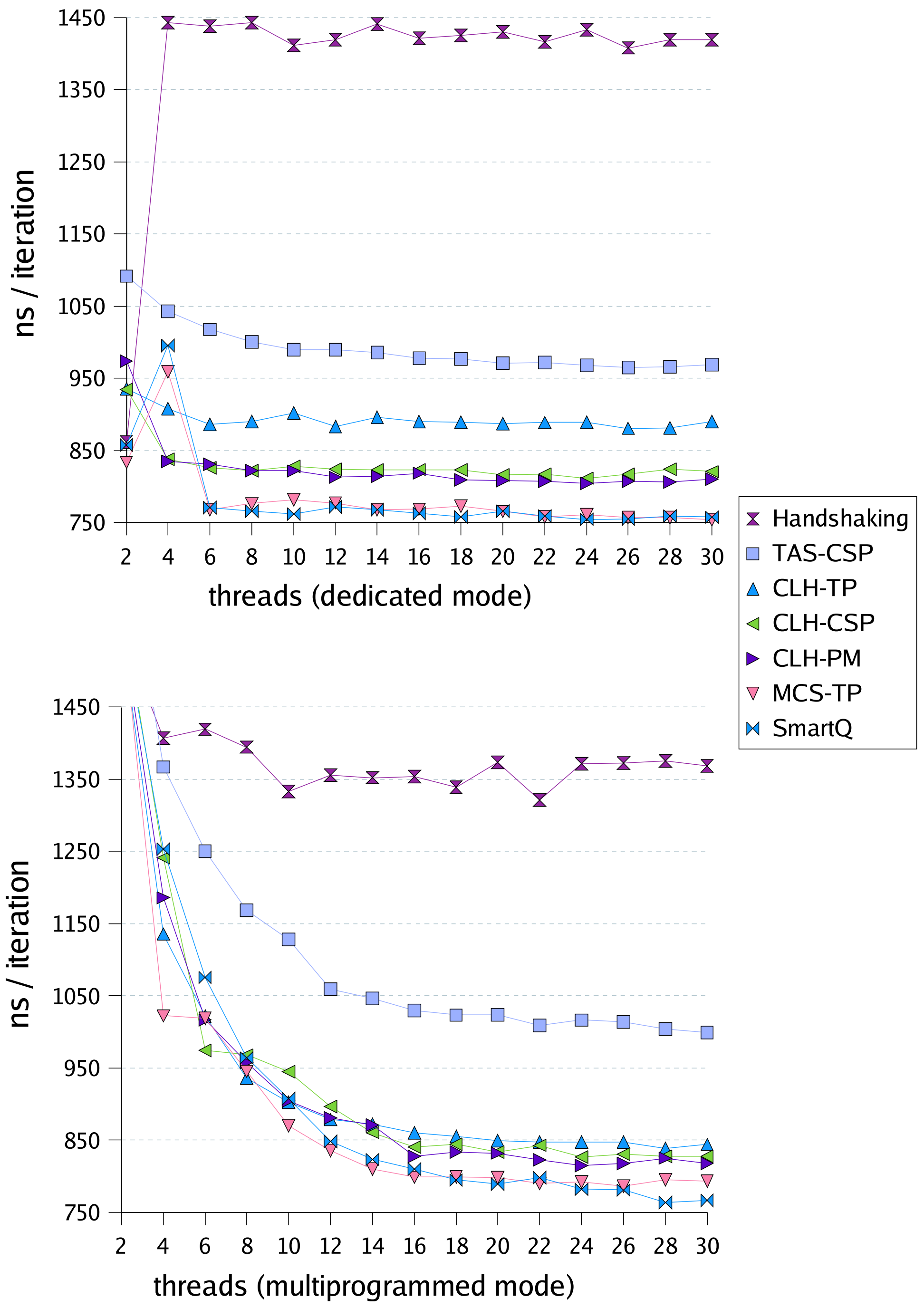


Figure 2.19: TP locks vs. spin-then-yield: 2 cache line CS; 1 μs NCS. Critical section service time (top) and success rate (bottom)



Critical section service time with (top) and without (bottom) 50% load on processors. **Top**, the curves of SmartQ and MCS-TP overlap with each other as do those of CLH-CSP and CLH-PM. **Bottom**, the close clustering of five curves suggests that they have similar tolerance for preemption.

Figure 2.20: Preemption-safe lock performance (small critical sections): 2 cache line CS; 1 μs NCS.

Overall, TP locks are competitive with preemption-safe locks. The modest increase in performance gained by locks that use high levels of scheduling control is comparatively minor. In dedicated mode, CLH-TP is 8–9% slower than the preemption-safe CLH variants, but it performs comparably in multiprogrammed mode. MCS-TP closely matches SmartQ (based on MCS) in both modes. Both TP locks clearly outperform the Handshaking lock.

In dedicated mode, CLH-TP incurs additional overhead from reading and publishing timestamps. In multiprogrammed mode, however, overhead from the preemption recovery mechanisms dominates. Since all three CLH variants handle preemption by removing inactive predecessor nodes from the queue, their performance is very similar.

Among preemption-safe locks, CLH-PM slightly outperforms CLH-CSP because it can more accurately assess whether threads are preempted. SmartQ significantly outperforms the Handshaking lock due to the latter's costly round-trip handshakes and its use of timeouts to confirm preemption.

Sensitivity to Patience

Timeout patience is an important parameter for try locks. Insufficient patience yields low success rates and long average critical section service times [Sco02; ScS01]. Conversely, excessive patience can delay a lock's response to bad scenarios. Our experiments show TP locks to be highly robust to changes in patience. Figure 2.21 shows the case with a large critical section; for smaller critical sections, the performance is even better. Overall, TP locks are far less sensitive to tuning of patience than other locks; with very low patience, the self-timeout and removal behaviors of the locks help to maintain critical section service time even as the acquisition rate plummets.

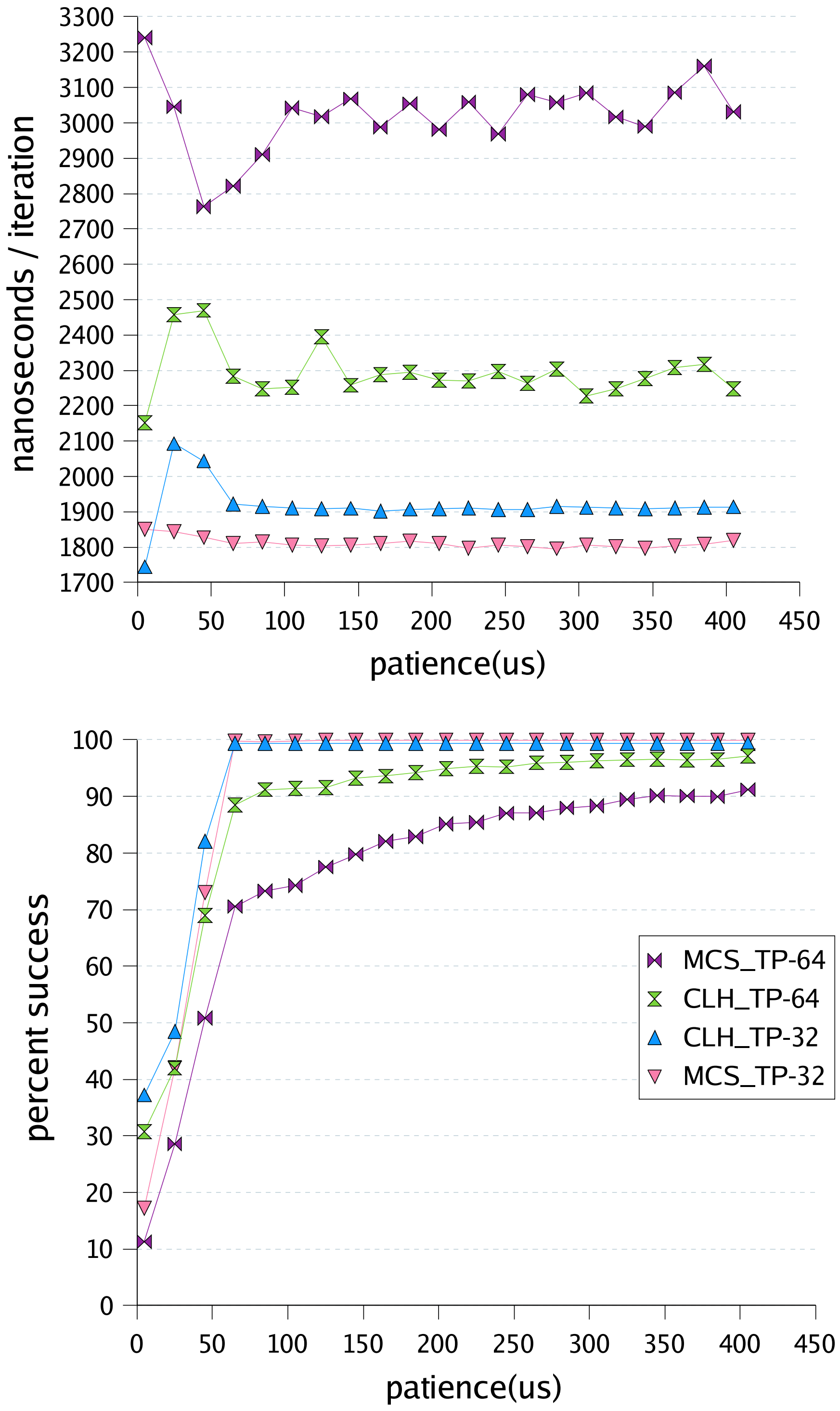


Figure 2.21: Sensitivity to patience in TP locks: 40 cache line CS; $4\,\mu s$ NCS.

Time and Space Bounds

As a final experiment, we measure the time overhead for removing an inactive node. On our Power4 p690, we calculate that the MCS-TP lock holder needs about 200–350 ns to delete each node. Similarly, a waiting thread in CLH-TP needs about 250–350 ns to delete a predecessor node. By combining these values with our worst-case analysis for the number of inactive nodes in the lock queues (Section 2.5.3), one can estimate an upper bound on delay for lock handoff when the holder is not preempted.

In our analysis of the space bounds for the CLH-TP lock (Section 2.5.3) we show a worst-case bound quadratic in the number of threads, but claim an expected linear value. Two final tests maximize space consumption to gather empirical evidence for the expected case. One test maximizes contention via empty critical and non-critical sections. The other stresses concurrent timeouts and removals by presetting the lock to held, so that every contending thread times out.

We ran both tests 6 times, for 5 and 10 second runs. We find space consumption to be very stable over time, getting equivalent results with both test lengths. With patience as short as 15 μs, the first test consumed at most 77 queue nodes with 32 threads, and at most 173 nodes with 64 threads. The second test never used more than 64 or 134 nodes with 32 or 64 threads. Since our allocator always creates a minimum of $2T$ nodes, 64 and 128 are optimal. The results are far closer to the expected linear than the worst-case quadratic space bound.

Application results

In a final set of tests we measure the performance of the TP locks on Raytrace and Barnes, two lock-based benchmarks from the SPLASH-2 suite [ABD92].

Application Features: Raytrace and Barnes spend much time in synchronization [KJC99; WOT95]. Raytrace uses no barriers but features high contention on a

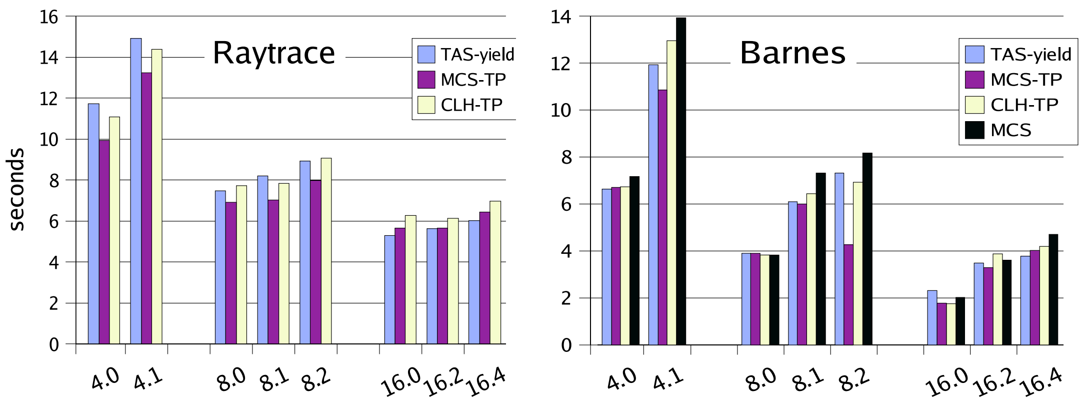


Figure 2.22: TP Locks: Evaluation with Raytrace and Barnes. Configuration *M.N* means M application threads and $(32 - M) + N$ external threads on the 32-way SMP.

small number of locks. Barnes uses limited barriers (17 for our test input) but numerous locks. Both offer reasonable parallel speedup.

Experimental Setup: We test each of the locks in Section 2.6 plus the native `pthread_mutex` on our p690, averaging results over 6 runs. We choose inputs large enough to execute for several seconds: 800×800 for Raytrace and 60K particles for Barnes. We limit testing to 16 threads due to the applications' limited scalability. External threads running idle loops generate load and force preemption.

Raytrace: The left side of Figure 2.22 shows results for three preemption tolerant locks: TAS-yield, MCS-TP and CLH-TP. Other spin locks give similar performance absent preemption; when preemption is present, non-TP queue-based locks yield horrible performance (Figures 2.17, 2.18, and 2.19). The `pthread_mutex` lock also scales very badly; with high lock contention, it can spend 80% of its time in kernel mode. Running Raytrace with our input size took several hours for 4 threads.

Barnes: Preemption tolerance is less important here than in Raytrace because Barnes distributes synchronization over a very large number of locks, greatly reducing the impact of preemption. We demonstrate this by including a highly preemption-sensitive lock, MCS, with our preemption tolerance locks in the right side of Figure 2.22; MCS “only” doubles its execution time with heavy preemption.

With both benchmarks, we find that our TP locks maintain good throughput and tolerate preemption well. With Raytrace, MCS-TP in particular yields 8-18$\%$ improvement over a yielding TATAS lock with 4 or 8 threads. Barnes is less dependent on lock performance in that the various preemption-tolerant locks have similar performance.

2.7 Conclusions

We have shown that it is possible, given standard atomic operations, to construct queue-based locks in which a thread can time out and abandon its attempt to acquire the lock. Our bounded-space algorithms (MCS-try and CLH-try) guarantee immediate reclamation of abandoned queue nodes, but require that a departing thread obtain the explicit cooperation of its neighbors. Our nonblocking timeout algorithm (MCS-NB-try) greatly improves tolerance for preemption, provided that alternative useful work exists that can be executed while waiting for the lock holder to resume. Our time-published algorithms ensure very high levels of throughput even in the presence of preemption, and even when contended locks are on the application’s critical path. We have argued for the nonblocking timeout and time-published algorithms that large amounts of space are unlikely to be required in practice, and our experimental results support this argument.

In the 15 years since Mellor-Crummey and Scott’s original comparison of spin lock algorithms [MeS91], ratios of single-processor latencies have remained remarkably stable. On a 16MHz Sequent Symmetry multiprocessor, a TATAS lock without contention consumed $7\mu s$ in 1991. The MCS lock consumed $9\mu s$, a difference of 29$\%$. On a 900 MHz SunFire v880, a TATAS lock without contention takes about $60ns$ today. The MCS lock takes about $115ns$, a difference of almost $2\times$. The CLH lock, which was not included in the 1991 study, takes about $75ns$, 25% slower than a TATAS lock, but 35% faster than an MCS lock on a single processor.

With two or more threads competing for access to a lock, the numbers have changed more significantly over the years. In 1991 the TATAS lock (with backoff) ran slightly slower than the MCS lock at modest levels of contention. Today it runs in less than a third of the time of all the queue-based locks. Why then would one consider a queue-based lock?

The answer is three-fold. First, the apparent speed of TATAS on small numbers of processors is misleading in our microbenchmarks: Because the lock is accessed in a tight loop, a single processor is able to reacquire the lock repeatedly, avoiding all cache misses. In fact, in the entire region in which TATAS locks appear to outperform queue-based alternatives, the lock is reacquired by the previous lock holder more than half the time. Second, TATAS does not scale well. With large numbers of processors attempting to acquire the lock simultaneously, we observe dramatic degradation in the relative performance of the TATAS lock and queue-based alternatives. Third, even with patience as high as $2ms$—200 times the average lock-passing time—the TATAS algorithm with timeout fails to acquire the lock about 20% of the time. This failure rate suggests that a regular TATAS lock (no timeout) will see significant variance in acquisition times—in other words, significant unfairness. The queue-based locks, by contrast, guarantee that threads acquire the lock in the order that their requests register with the lock queue.

For applications that do not have sufficient parallelism to tolerate preemption, we have demonstrated that published timestamps provide an effective heuristic by which a thread can accurately guess the scheduling status of its peers, without support from a nonstandard scheduler API. Empirical tests confirm that our MCS-TP and CLH-TP locks combine scalability, preemption tolerance, and low observed space overhead with throughput as high as that of the best previously known solutions. Given the existence of a low-overhead time-of-day register with low system-wide skew, our results make it feasible, for the first time, to use queue-based locks on multiprogrammed systems with a standard kernel interface.

As future work, we conjecture that published timestamps can be used to improve thread interaction in other areas, such as preemption-tolerant barriers, priority-based lock queuing, dynamic adjustment of the worker pool for bag-of-task applications, and contention management for nonblocking concurrent algorithms (see Section 4.4). Finally, we note that we have examined only two points in the design space of TP locks; other variations may merit consideration.

2.8 Acknowledgments

We are indebted to Vitaly Oratovsky and Michael O'Donnell of Mercury Computer Corp. for drawing our attention to the subject of queue-based try locks. Mark Hill, Alaa Alameldeen, and the Computer Sciences Department at the University of Wisconsin–Madison provided access to a Sun Wildfire machine used for preliminary evaluation of the MCS-try and CLH-try locks [ScS01]. Victor Luchangco, Paul Martin, Mark Moir, and Brian Whitney of Sun Microsystems were instrumental in obtaining results on more recent large-scale Sun machines.

The MCS-try and CLH-try locks were originally designed by Michael L. Scott; my primary contribution to their development was to run and analyze experiments to assess their performance characteristics; I also co-developed with Dr. Scott the microbenchmark test driver and co-designed substantial improvements to the CLH-try lock with him and with Bijun He (now at Google, Inc.). I also created the extended description of the MCS-try in Section 2.3. Finally, I designed the new version of MCS-NB-try presented in Section 2.4, and ported the various locks to the Cray T3E and collected the results presented in Figures 2.13 and 2.14.

The idea of using time to further reduce windows of preemption vulnerability in queue-based locks was mine; however it was Bijun He who brought this idea to fruition.

I co-designed the time-published locks with her, but she is the one who completed the implementations, devised the technique used for conditional updates in the CLH-TP lock presented in Figure 2.12, and collected and graphed the performance results described in Section 2.6.4.

3 Ad Hoc Nonblocking Synchronization

3.1 Introduction

As discussed in Chapter 1, lock-based implementations of concurrent data structures come in two flavors: coarse-grained algorithms that do not scale very well; and fine-grained algorithms for which avoiding deadlock and ensuring correct semantics require extremely careful protocols. Either category is further subject to a variety of problems inherent to locking, such as priority inversion or vulnerability to thread crashes and preemption.

An alternative to lock-based synchronization may be found in *nonblocking* algorithms. Such algorithms share with fine-grained locking implementations a need for careful protocols to avoid problems despite a wide range of possible interleavings of concurrent operations. On the other hand, they are by definition immune to many of the problems inherent to lock-based synchronization.

Although nonblocking synchronization generally incurs some amount of base-case overhead, for certain key data structures — stacks [Tre86] and queues [MiS96] in particular — nonblocking implementations outperform [MiS98] lock-based implementations.

Nonblocking algorithms are notoriously difficult to design and implement. Although this difficulty is partially inherent to asynchronous interleavings due to concurrency, it may also be ascribed to the many different concerns that must be addressed in the design process. With lock-free synchronization, for example, one must not only ensure that the algorithm functions correctly, but also guard against livelock. With wait-free synchronization one must additionally ensure that every thread makes progress in bounded time; in general this requires that one "help" conflicting transactions rather than aborting them.

Obstruction-free concurrent algorithms[HLM03a] lighten the burden by separating progress from correctness, allowing programmers to address progress as an out-of-band, orthogonal concern. The core of an obstruction-free algorithm only needs to guarantee progress when only one thread is running (though other threads may be in arbitrary states).

Many nonblocking algorithms are known, including implementations of such common data structures as stacks [Tre86; SuT02; HSY04], queues [Sto92; PLJ94; MiS96; Val94; SHC00; TsZ01; LaS04; MNS05], deques [ADF00; DFG00; Mic03; HLM03a], priority queues [ShZ99; SuT03], hash tables [Mic02; ShS03], linked lists [Val95; Har01; Mic02; HHL05], and skiplists [Pug90]. One common feature shared by all of these algorithms is that every method is *total*: It has no preconditions that must be satisfied before it can be executed. In particular, any blocking that a thread might perform by definition precludes it from being nonblocking. This restriction against any form of blocking stems both from the nonblocking progress condition definitions and from linearizability theory [HeW90], the primary tool by which designers prove the correctness of nonblocking algorithms.

In Section 3.2, we introduce a design methodology that allows some blocking in restricted cases to support concurrent objects with condition synchronization, enabling the use of linearizability theory for objects with partial methods. This in turn allows meaningful definitions of the various nonblocking progress conditions with such objects. We then apply these definitions in Section 3.3 to create general lock-free dual stacks and dual queues, which in turn are building blocks for our lock-free exchangers (Section 3.4) and synchronous queues (Section 3.5), both of which will appear in the concurrency package of Java 6.

3.2 Linearizability and Condition Synchronization

3.2.1 Motivation

Since its introduction fifteen years ago, linearizability has become the standard means of reasoning about the correctness of concurrent objects. Informally, linearizability "provides the illusion that each operation... takes effect instantaneously at some point between its invocation and its response" [HeW90, abstract]. Linearizability is "nonblocking" in the sense that it never requires a call to a total method (one whose precondition is simply **true**) to wait for the execution of any other method. (Certain other correctness criteria, such as serializability [Pap79], may require blocking, e.g. to enforce coherence across a multi-object system.) The fact that it is nonblocking makes linearizability particularly attractive for reasoning about nonblocking *implementations* of concurrent objects, which provide guarantees of various strength regarding the progress of method calls in practice. In a *wait-free* implementation, every contending thread is guaranteed to complete its method call within a bounded number of its own time steps [Her91]. In a *lock-free* implementation, *some* some contending thread is guaranteed to complete its method call within a bounded number of steps (from any thread's point of view) [Her91]. In an *obstruction-free* implementation, a thread is guaranteed to complete its method call within a bounded number of steps in the absence of contention, i.e. if no other threads execute competing methods concurrently [HLM03a].

These various *progress conditions* all assume that every method is total. As Herlihy puts it [Her91, p. 128]:

> We restrict our attention to objects whose operations are total because it is unclear how to interpret the wait-free condition for partial operations. For example, the most natural way to define the effects of a partial *deq* in a concurrent system is to have it wait until the queue becomes nonempty, a specification that clearly does not admit a wait-free implementation.

To avoid this problem the designers of nonblocking data structures typically "totalize" their methods by returning an error flag whenever the current state of the object does not admit the method's intended behavior.

But partial methods are important! Many applications need a `dequeue`, `pop`, or `deletemin` operation that waits when its structure is empty; these and countless other examples of *condition synchronization* are fundamental to concurrent programming.

Given a nonblocking data structure with "totalized" methods, the obvious spin-based strategy is to embed each call in a loop, and retry until it succeeds. This strategy has two important drawbacks. First, it introduces unnecessary contention for memory and communication bandwidth, which may significantly degrade performance, even with careful backoff. Second, it provides no fairness guarantees.

Consider a total queue whose `dequeue` method waits until it can return successfully, and a sequence of calls by threads A, B, C, and D:

C enqueues a 1
D enqueues a 2
A calls dequeue
A's call returns the 2
B calls dequeue
B's call returns the 1

This is clearly a "bad" execution history, because it returns results in the wrong (non-FIFO) order; it implies an incorrect implementation. The following is clearly a "good" history:

A calls dequeue
B calls dequeue
C enqueues a 1
D enqueues a 2

A's call returns the 1
B's call returns the 2

But what about the following:

A calls dequeue
B calls dequeue
C enqueues a 1
D enqueues a 2
B's call returns the 1
A's call returns the 2

If the first line is known to have occurred before the second (this may be the case, for example, if waiting threads can be identified by querying the scheduler, examining a thread control block, or reading an object-specific flag), then intuition suggests that while this history returns results in the right order, it returns them to the wrong threads. If we implement our queue by wrapping the nonblocking "totalized" `dequeue` in a loop, then this third, questionable history may certainly occur.

3.2.2 Linearizability

Following Herlihy and Wing [HeW90], a *history* of an object is a (potentially infinite) sequence of method invocation events $\langle m(args)\ t\rangle$ and response (return) events $\langle r(val)\ t\rangle$, where m is the name of a method, r is a return condition (usually "ok"), and t identifies a thread. An invocation *matches* the next response in the sequence that has the same thread id. Together, an invocation and its matching response are called an *operation*. The invocation and response of operation o may also be denoted $inv(o)$ and $res(o)$, respectively. If event e_1 precedes event e_2 in history H, we write $e_1 <_H e_2$.

A history is *sequential* if every response immediately follows its matching invocation. A non-sequential history is *concurrent*. A *thread subhistory* is the subsequence of a history consisting of all events for a given thread. Two histories are *equivalent* if all their thread subhistories are identical. We consider only *well-formed* concurrent histories, in which every thread subhistory is sequential, and begins with an invocation.

The semantics of an object determine a set of *legal* sequential histories. In a queue, for example, items must be inserted and removed in FIFO order. That is, the nth successful `dequeue` in a legal history must return the value inserted by the nth `enqueue`. Moreover at any given point the number of prior `enqueues` must equal or exceed the number of successful `dequeues`. To permit `dequeue` calls to occur at any time (i.e., to make `dequeue` a *total* method—one whose precondition is simply **true**), one can allow unsuccessful `dequeues` [$\langle deq(\,)\ t\rangle\ \langle no(\bot)\ t\rangle$] to appear in the history whenever the number of prior `enqueues` equals the number of prior successful dequeues.

A (possibly concurrent) history H induces a partial order $\prec_H$ on operations: $o_i \prec_H o_j$ if $res(o_i) <_H inv(o_j)$. We say that a history H is *linearizable* if (a) it is equivalent to some legal sequential history S, and (b) $\prec_H \subseteq \prec_S$. Finally, an object is *concurrent* if its operations are linearizable to those of a sequential version.

3.2.3 Extending Linearizability to Objects with Partial Methods

When an object has partial methods, we divide each such method into separate, first-class *request* and *followup* operations, each of which has its own invocation and response. A total queue, for example, would provide `dequeue_request` and `dequeue_followup` methods. By analogy with Lamport's bakery algorithm [Lam74], the request operation returns a unique *ticket* (also referred to as a *reservation*), which is then passed as an argument to the follow-up method. The follow-up, for its part, returns either the desired result (if one is *matched* to the ticket) or, if the method's precondition has not yet been satisfied, an error indication.

Given standard definitions of well-formedness, a thread t that wishes to execute a partial method p must first call p`_request` and then call p`_followup` in a loop until it succeeds. This is very different from calling a traditional "totalized" method until it succeeds: Linearization of distinguished request operations is the hook that allows object semantics to address the order in which pending requests will be fulfilled.

As a practical matter, implementations may wish to provide a p_`demand` method that waits until it can return successfully, and/or a plain p method equivalent to p_`demand`(p_`request`). The obvious implementation of p_`demand` contains a busy-wait loop, but other implementations are possible. In particular, an implementation may choose to use scheduler-based synchronization to put t to sleep on a semaphore that will be signaled when p's precondition has been met, allowing the processor to be used for other purposes in the interim. We require that it be possible to provide request and follow-up methods, as defined herein, with no more than trivial modifications to any given implementation. The algorithms we present in Section 3.3 provide only a plain p interface, with internal busy-wait loops.

3.2.4 Contention Freedom

When reasoning about progress, we must deal with the fact that a partial method may wait for an arbitrary amount of time (perform an arbitrary number of unsuccessful followups) before its precondition is satisfied. Clearly it is desirable that requests and follow-ups be nonblocking. But in practice, good system performance will also typically require that unsuccessful follow-ups not interfere with progress in other threads. In this spirit, we define a concurrent data structure as *contention-free* if none of its followup operations performs more than a constant number of remote memory accesses across all unsuccessful invocations with the same request ticket. On a cache-coherent machine that can cache remote memory locations, an access by thread t within operation o is said to be *remote* if it writes to memory that may (in some execution) be read or written by threads other than t more than a constant number of times between $inv(o)$ and $res(o)$, or if it reads memory that may (in some execution) be written by threads other than t more than a constant number of times between $inv(o)$ and $res(o)$. On a non-cache-coherent machine, an access by thread t is also remote if it refers to memory that t itself did not allocate. Compared to the local-spin property [MeS91], contention freedom allows operations to block in ways other than busy-wait spinning; in particular, it allows other actions to be performed while waiting for a request to be satisfied.

3.2.5 Dual Data Structures

Borrowing terminology from the BBN Butterfly Parallel Processor of the early 1980s [BBN86], we define a *dual data structure* to be any concurrent object that supports partial methods via the request—followup methodology described in Section 3.2.3. By extension, then, a nonblocking dual data structure is simply dual data structure with nonblocking request and followup operations.

3.3 The Dual Stack and Dual Queue

In this section, we present two sample dual data structures, the dual stack and the dual queue, that exemplify the benefits inherent to our dual methodology.

3.3.1 Semantics

Dual Stack Semantics

A dual stack DS is an object that supports three operations: `push`, `pop_reserve`, and `pop_followup`. The state of a dual stack is a tuple composed from an ordered sequence of requests (r_i), an ordered sequence of data items (v_j), and a set of matched requests (mr_k/mv_k); $DS = \langle\langle r_0, \ldots\ r_i\rangle, \langle v_0, \ldots v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$. The dual stack is initially $\langle\langle\rangle, \langle\rangle, \emptyset\rangle$; the state transition rules defined below ensure that at least one of r_i and v_i is always empty. The operations `push`, `pop_reserve`, and `pop_followup` induce the following state transitions on the tuple $DS = \langle\langle r_0, \ldots\ r_i\rangle$, $\langle v_0, \ldots v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$, with appropriate return values:

- `push(`v_{new}`)`: If the list r_i is empty, changes DS to be $\langle\langle\rangle, \langle v_{new}, v_0, \ldots\ v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$. Otherwise, it changes DS to be $\langle\langle r_1, \ldots\ r_i\rangle, \langle\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k), (r_0/v_{new})\}\rangle$.

- `pop_reserve()`: If the list v_j is empty, changes DS to be $\langle\langle r_{new}, r_0, \ldots\ r_i\rangle, \langle\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$, and returns r_{new} as the reservation (request ticket) for this operation. Otherwise, it changes DS to be $\langle\langle\rangle, \langle v_1, \ldots\ v_j\}\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k), (mr_{new}/v_0)\}\rangle$ and returns mr_{new} as the reservation for this operation.

- `pop_followup(`res`)`: If res matches mr_i in the set of matched requests, changes DS to $\langle\langle\rangle, \langle v_0, \ldots\ v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_{i-1}/mv_{i-1}), (mr_{i+1}/mv_{i+1}), (mr_k/mv_k)\}\rangle$. Otherwise, it returns $failed$ and DS is unchanged.

A common shorthand (for programmer convenience) is to partially combine `pop_reserve()` and `pop_followup()` into a single `pop_conditional()` operation that, if the list v_j is non-empty, returns the result of `pop_followup(pop_reserve());` and otherwise returns the ticket from `pop_reserve()`.

A dual stack has correct LIFO semantics if the following requirements are met for all dual stack operations: [1]

1. If *op* is a `pop_followup` that returns item i, then i was previously pushed by a `push` operation.

2. If *op*$_1$ is a `push` operation that pushed a item i to the dual stack and *op*$_2$ and *op*$_3$ are `pop_followup` operations that return i, then *op*$_2$ = *op*$_3$.

3. If *op*$_1$ is a `pop_followup` operation that returns item i for the reservation returned by `pop_reserve` operation *op*$_2$, where i was pushed by `push` operation

[1]For simplicity of presentation, we have assumed that all items pushed to the dual stack are unique; extending these semantics to support item duplication is straightforward if one couples items with the sequence number of the `push` operation that placed them in the stack.

op_3, then equal numbers of `push` and `pop_reserve` operations completed in the interval bounded by op_2 and op_3.

4. If op is a `pop_followup` operation that returns $failed$ for the N^{th} reservation returned by `pop_reserve`, then either at most $(N-1)$ `push` operations have completed, or at least $(K \geq N+1)$ `pop_reserve` operations have completed and at most $(K-1)$ `push` operations have completed.

We claim (proof ommitted) that any dual stack that correctly implements the `push`, `pop_reserve`, and `pop_followup` operations will also satisfy these semantics.

In summary, a concurrent dual stack is a dual data structure whose operations are linearizable [HeW90] to those of the sequential dual stack just defined. The dual stack we present in Section 3.3.2 is both lock-free and contention-free.

Dual Queue Semantics

A dual queue DQ is an object that supports three operations: `enqueue`, `dequeue_reserve`, and `dequeue_followup`. The state of a dual queue is a tuple composed from an ordered sequence of requests (r_i), an ordered sequence of data items (v_j), and a set of matched requests (mr_k/mv_k); $DS = \langle\langle r_0, \ldots\ r_i\rangle, \langle v_0, \ldots v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$. The dual queue is initially $\langle\langle\rangle, \langle\rangle, \emptyset\rangle$; the state transition rules defined below ensure that at least one of r_i and v_i is always empty. The operations `enqueue`, `dequeue_reserve`, and `dequeue_followup` induce the following state transitions on the tuple $DQ = \langle\langle r_0, \ldots\ r_i\rangle, \langle v_0, \ldots v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$, with appropriate return values:

- `enqueue(`v_{new}`)`: If the list r_i is empty, changes DQ to be $\langle\langle\rangle, \langle v_0, \ldots v_j, v_{new}\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$. Otherwise, it changes DQ to be $\langle\langle r_1, \ldots\ r_i\rangle, \langle\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k), (r_0/v_{new})\}\rangle$.

- `dequeue_reserve()`: If the list v_j is empty, changes DQ to be $\langle\langle r_0, \ldots r_i, r_{new}\rangle, \langle\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k)\}\rangle$, and returns r_{new} as the reservation (request ticket) for this operation. Otherwise, it changes DQ to be $\langle\langle\rangle, \langle v_1, \ldots v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_k/mv_k), (mr_{new}/v_0)\}\rangle$ and returns mr_{new} as the reservation for this operation.

- `dequeue_followup(`res`)`: If res matches mr_i in the set of matched requests, changes DQ to $\langle\langle\rangle, \langle v_0, \ldots\ v_j\rangle, \{(mr_0/mv_0), \ldots\ (mr_{i-1}/mv_{i-1}), (mr_{i+1}/mv_{i+1}), (mr_k/mv_k)\}\rangle$. Otherwise, it returns $failed$ and DQ is unchanged.

A common shorthand (for programmer convenience) is to partially combine `dequeue_reserve()` and `dequeue_followup()` into a single `dequeue_conditional()` operation that, if the list v_j is non-empty, returns the result of `dequeue_followup(dequeue_reserve())`; and otherwise returns the ticket from `dequeue_reserve()`.

A dual queue has correct FIFO semantics if the following requirements are met for all dual queue operations: [2]

1. If *op* is a `dequeue_followup` that returns item i, then i was previously enqueued by an `enqueue` operation.

2. If op_1 is an `enqueue` operation that pushed a item i to the dual queue and op_2 and op_3 are `dequeue_followup` operations that return i, then $op_2 = op_3$.

3. If *op* is a `dequeue_followup` operation that returns item i for the N^{th} reservation returned by a `dequeue_reserve` operation, then i was previously enqueued by the N^{th} `enqueue` operation.

[2]Again, for simplicity of presentation, we have assumed that all items enqueued in the dual queue are unique; extending these semantics to support item duplication is straightforward if one couples items with the sequence number of the `enqueue` operation that placed them in the queue.

4. If *op* is a `dequeue_followup` operation that returns item i for the N^{th} reservation returned by a `dequeue_reserve` operation, then at least N `enqueue` operations have completed.

5. If *op* is a `dequeue_followup` operation that returns `failed` for the N^{th} reservation returned by a `dequeue_reserve` operation, then at most $N - 1$ `enqueue` operations have completed.

We claim (proof ommitted) that any dual queue that correctly implements the `enqueue`, `dequeue_reserve`, and `dequeue_followup` operations will also satisfy these semantics.

In summary, a concurrent dual queue is a dual data structure whose operations are linearizable [HeW90] to those of the sequential dual queue just defined. The dual queue we present in Section 3.3.3 is both lock-free and contention-free.

3.3.2 The Dual Stack

The dual stack is based on the standard lock-free stack of Treiber [Tre86]. So long as the number of calls to `pop` does not exceed the number of calls to `push`, the dual stack behaves the same as its non-dual cousin. Pseudocode for the dual stack appears in Listing 3.1.

When the stack is empty, or contains only reservations, the `pop` method pushes a reservation, and then spins on the `data_node` field within it. A `push` method always pushes a data node. If the previous top node was a reservation, however, the data node is marked as fulfilling it and the two adjacent nodes "annihilate" each other: Any thread that finds a filler node and an underlying reservation at the top of the stack attempts to (a) write the address of the former into the `data_node` field of the latter, and then (b) pop both nodes from the stack. At any given time, the stack contains either all reservations, all data, or one datum (a filler node at the top) followed by reservations.

Both the head pointer and the `next` pointers in stack nodes are *tagged* to indicate whether the next node in the list is a reservation or a datum and, if the latter, whether there is a reservation beneath it in the stack. [3] We assume that nodes are word-aligned, so that these tags can fit in the low-order bits of a pointer. For presentation purposes the pseudocode assumes that data values are integers, though this could obviously be changed to any type (including a pointer) that will fit, together with a serial number, in the target of a double-width CAS (or in a single word on a machine with LL/SC). To differentiate between the cases where the topmost data node is present to fulfill a request and where the stack contains all data, pushes for the former case set both the data and reservation tags; pushes for the latter set only the data tag.

As mentioned in Section 3.2.3 our code provides a single `pop` method that subsumes the sequence of operations from a `pop` request through its successful follow-up. The linearization point in the `pop_reserve` subcomponent of `pop`, like the linearization point in `push`, is the CAS that modifies the top-of-stack pointer. The linearization of the `pop_followup` subcomponent is either when the spin is broken by having data supplied, or immediately after the linearization point of `pop_reserve`, in the case where data was already present in the stack.

The code for `push` is lock-free, as is the code for the `pop_reserve` and `pop_followup` portions of `pop`. Moreover, the spin in `pop` (which would comprise the body of an unsuccessful follow-up operation, if we provided it as a separate method), is entirely local: It reads only the requester's own reservation node, which the requester allocated itself, and which no other thread will write except to terminate the spin. The dual stack therefore satisfies the conditions listed in Section 3.2.5.

[3]For languages like Java in which tag bits in pointers are unavailable, we can either use a run-time type identification (RTTI) mechanism to distinguish between trivial subclasses of a node type (that represent the bit on or off) or place the bits inside the object itself

Though we do not offer a proof, one can analyze the code to confirm that the dual stack satisfies the LIFO semantics presented in Section 3.3.1: [4] If the number of previous `push` operations exceeds the number of previous `pop` operations, then a new `pop` operation p will succeed immediately, and will return the value provided by the most recent previous `push` operation h such that the numbers of pushes and pops between h and p are equal. In a similar fashion, the dual stack satisfies pending requests in LIFO order: If the number of previous `pop` operations exceeds the number of previous `push` operations, then a `push` operation h will provide the value to be returned by the most recent previous `pop` operation p such that the numbers of pushes and pops that linearized between p and h are equal.

The spin in `pop` is terminated by a CAS in some other thread (possibly the fulfilling thread, possibly a helper) that updates the `data_node` field in the reservation. Once the fulfilling `push` has linearized, no thread will be able to make progress until the a successful `pop_followup` linearizes.

It is tempting to consider a simpler implementation in which the fulfilling thread pops a reservation from the stack and then writes the fulfilling datum directly into the reservation. This implementation, however, is not comformant to the dual stack semantics: It admits executions in which the other push operations complete between the pop and the write. In particular, if the fulfilling thread were to failure or stall subsequent to popping the reservation but prior to writing the datum, the reservation would no longer be in the stack and an arbitrary number of additional `pop` operations (performed by other threads, and returning subsequently `push`ed data) could complete before the requester's successful `pop_followup` operation. This allows violations of semantic rules 3 and 4 from Section 3.3.1.

[4]Strictly speaking, the initial CAS can fail, so no CAS-based stack can offer perfectly LIFO semantics. In this work, we have followed the usual convention of defining semantics relative to the linearization points for push and pop operations. We define FIFO ordering in queues similarly.

```
struct cptr {                // counted pointer
  snode *ptr;
  int sn;
};  // 64-bit datatype

struct tptr {                // tagged pointer
  snode *30 ptr;
  bool is_request;           // tags describe pointed-to node;
  bool data_flag;            // together mark a filler node
};  // 32-bit datatype

struct ctptr extends tptr {  // counted tagged pointer
  int sn;
};  // 64-bit datatype

struct dualstack {
  ctptr head;
};

struct snode {               // stack node
  union {
    int data;
    cptr data_node;      // data must overlie ptr, not sn
  };
  tptr next;
};

void ds_init(dualstack *S) {
  stack->head.ptr = {NULL, FALSE, FALSE};
}
void push(int v, dualstack *S) {
  snode *n = (snode)allocate_snode();
  n->data = v;

  while (1) {
    ctptr head = S->head;
    n->next = head;
    if (head.ptr == NULL || (head.data_flag && !head.is_request)) {
      if (cas(&S->head, head, {{n, FALSE, TRUE}, head.sn+1})) return;
    } else if (head.is_request) {
      tptr next = head.ptr->next;
      cptr old = head.ptr->data_node;
      // link in filler node
      if (!cas(&S->head, head, {{n, TRUE, TRUE}, head.sn+1}))
        continue;   // someone else fulfilled the request
      // fulfill request node
      (void)cas(&head.ptr->data_node, old, {n, old.sn+1});
      // link out filler and request
      (void)cas(&S->head, {{n, TRUE, TRUE}, head.sn+1}, {next, head.sn+2});
      return;
    } else {    // data underneath; need to help
      tptr next = head.ptr->next;
      if (next.ptr == NULL) continue;  // inconsistent snapshot
      cptr old = next.ptr->data_node;
      if (head != S->head) continue;   // inconsistent snapshot
      // fulfill request node
      if (old.ptr == NULL)
        (void)cas(&next.ptr->data_node, old, {head.ptr, old.sn+1});
      // link out filler and request
      (void)cas(&S->head, head, {next->next, head.sn+1});
    }
  }
}
```

Listing 3.1: The dual stack

```
int pop(dualstack *S) {
  snode *n = NULL;

  while (1) {
    ctptr head = S->head;
    if (head.data_flag && !head.is_request) {
      tptr next = head.ptr->next;
      if (cas(&S->head, head, {next, head.sn+1})) {
        int result = head.ptr->data;
        deallocate(head.ptr);
        if (n != NULL) deallocate(n);
        return result;
      }
    } else if (head.ptr == NULL || head.is_request) {
      if (n == NULL) {
        n = allocate_snode();
        n->data_node.ptr = {NULL, FALSE, FALSE};
      }
      n->next = {head.ptr, TRUE, FALSE};
      if (!cas(&S->head, head, {{n, TRUE, FALSE}, head.sn+1}))
        continue;   // couldn't push request
      while (NULL == n->data_node.ptr); // local spin
      // help remove my request node if needed
      head = S->head;
      if (head.ptr == n)
        (void)cas(&S->head, head, {n->next, head.sn+1});
      int result = n->data_node.ptr->data;
      deallocate(n->data_node.ptr);  deallocate(n);
      return result;
    } else {  // data underneath; need to help
      tptr next = head.ptr->next;
      if (next.ptr == NULL) continue;  // inconsistent snapshot
      cptr old = next.ptr->data_node;
      if (head != S->head) continue;   // inconsistent snapshot
      // fulfill request node
      if (old.ptr == NULL)
        (void)cas(&next.ptr->data_node, old, {head.ptr, old.sn+1});
      // link out filler and request
      (void)cas(&S->head, head, {next->next, head.sn+1});
    }
  }
}
```

Listing 3.1: (continued)

3.3.3 The Dual Queue

The dual queue is based on the M&S lock-free queue [MiS96]. So long as the number of calls to `dequeue` does not exceed the number of calls to `push`, it behaves the same as its non-dual cousin. Source code for the dual queue appears in Listing 3.2.

The dual queue is initialized with a single "dummy" node; the first real datum (or reservation) is always in the second node, if any. At any given time the second and subsequent nodes will either all be reservations or all be data.

When the queue is empty, or contains only reservations, the `dequeue` method enqueues a reservation, and then spins on the `request` pointer field of the former tail node. The `enqueue` method, for its part, fulfills the request at the head of the queue, if any, rather than enqueue a datum. To do so, the fulfilling thread uses a CAS to update the reservation's `request` field with a pointer to a node (outside the queue) containing the provided data. This simultaneously fulfills the request and breaks the requester's spin.[5] Any thread that finds a fulfilled request at the head of the queue removes and frees it.

As in the dual stack, queue nodes are tagged as requests by setting a low-order bit in pointers that point to them. We again assume, without loss of generality, that data values are integers, and we provide a single `dequeue` method that subsumes the sequence of operations from a `dequeue` request through its successful follow-up.

The code for `enqueue` is lock-free, as is the code for the `dequeue_reserve` and `dequeue_followup` portions of `dequeue`. The spin in `dequeue` (which would comprise the body of an unsuccessful follow-up) accesses a node that no other thread will write except to terminate the spin. The dual queue therefore satisfies the conditions listed in Section 3.2.5 on a cache-coherent machine. (On a non-cache-coherent machine we could simply arrange for the head-most node in the queue to always be the dummy node and spin on the node that we allocated and inserted. This has the disadvantage of moving a cache line miss onto the critical path when an enqueuer provides data to a dequeuer. In the implementation presented here, this cache line miss does not occur until after the waiter is signaled.)

[5]Note, however, that acting on the head of the queue requires careful consistency validation of the `head`, `tail`, and `next` pointers. Extending the technique of the original M&S queue, we use a two-stage check to ensure sufficient consistency to prevent untoward race conditions.

Though we do not offer a proof, one can analyze the code to confirm that the dual queue satisfies the FIFO semantics presented in Section 3.3.1: If the number of previous `enqueue` operations exceeds the number of previous `dequeue` operations, then a new, nth `dequeue` operation will return the value provided by the nth `enqueue`. In a similar fashion, the dual queue satisfies pending requests in FIFO order: If the number of previous `dequeue` operations exceeds the number of previous `enqueue` operations, then a new, nth `enqueue` operation will provide a value to the nth `dequeue`.

The spin in `dequeue` is terminated by a CAS in another thread's `enqueue` method. Note again that a simpler algorithm, in which the `enqueue` method could remove a request from the queue and then fulfill it, would be semantically nonconformant: An arbitrary number of dequeue operations could complete between the CAS used for removal and the linearization point for a successful `dequeue_followup` if the thread performing the `enqueue` were to stall.

3.3.4 Memory Management

For both the dual stack and the dual queue, correctness is dependent on an assumption that a block of memory, once allocated for use as a stack or queue node, cannot be used to contain other data until no thread maintains a reference to it. For garbage-collected languages, one gets this "for free". However, for C (as for other non-garbage-collected languages), the standard `malloc` and `free` library routines in particular do not suffice for this purpose. One approach that one could use to ensure this requirement be met would be to use a pointer tracking scheme [Mic04; HLM02] that manages threads' outstanding pointer references. For the experiments described here, however, we instead use a custom memory allocator [Sco02] (shown in Listing 2.9) that permanently marks memory as being for use in the dual data structure; it has the advantage of being extremely fast.

```
struct cptr {
  qnode *ptr;
  int sn;
};  // 64-bit datatype

struct ctptr {              // counted tagged pointer
  qnode *31 ptr;
  bool is_request;        // tag describes pointed-to node
  int sn;
};  // 64-bit datatype

struct qnode {
  cval data;
  cptr request;
  ctptr next;
};

struct dualqueue {
  cptr head;
  ctptr tail;
};

void dq_init(dualqueue *Q)
{
  qnode *qn = allocate_qnode();
  qn->next.ptr = NULL;
  Q->head.ptr = Q->tail.ptr = qn;
  Q->tail.is_request = FALSE;
}

void enqueue(int v, dualqueue *Q) {
  qnode *n = allocate_qnode();
  n->data = v;
  n->next.ptr = n->request.ptr = NULL;
  while (1) {
    ctptr tail = Q->tail;
    cptr head = Q->head;
    if (tail.ptr == head.ptr) || !tail.is_request) {
      cptr next = tail.ptr->next;
      if (tail == Q->tail) {  // tail and next are consistent
        if (next.ptr != NULL) {   // tail falling behind
          (void)cas(&Q->tail, tail, {{next.ptr, next.is_request}, tail.sn+1});
        } else {  // try to link in the new node
          if (cas(&tail.ptr->next, next, {{n, FALSE}, next.sn+1})) {
            (void)cas(&Q->tail, tail, {{n, FALSE}, tail.sn+1});
            return;
          }
        }
      }
    } else {  // queue consists of requests
      ctptr next = head.ptr->next;
      if (tail == Q->tail) {    // tail has not changed
        cptr req = head.ptr->request;
        if (head == Q->head) {  // head, next, and req are consistent
          bool success = (req.ptr == NULL
            && cas(&head.ptr->request, req, {n, req.sn+1}));
          // try to remove fulfilled request even if it’s not mine
          (void)cas(&Q->head, head, {next.ptr, head.sn+1});
          if (success) return;
        }
      }
    }
  }
}
```

Listing 3.2: The dual queue

```
int dequeue(dualqueue *Q) {
  qnode *n = allocate_qnode();
  n->is_request = TRUE;
  n->ptr = n->request = NULL;

  while (1) {
    cptr head  = Q->head;
    ctptr tail = Q->tail;
    if ((tail.ptr == head.ptr) || tail.is_request) {
      // queue empty, tail falling behind, or queue contains data (queue could also
      // contain exactly one outstanding request with tail pointer as yet unswung)
      cptr next = tail.ptr->next;
      if (tail == Q->tail) {  // tail and next are consistent
        if (next.ptr != NULL) {   // tail falling behind
          (void)cas(&Q->tail, tail, {{next.ptr, next.is_request}, tail.sn+1});
        } else {  // try to link in a request for data
          if (cas(&tail.ptr->next, next, {{n, TRUE}, next.sn+1})) {
            // linked in request; now try to swing tail pointer
            (void)cas(&Q->tail, tail, {{n, TRUE}, tail.sn+1}) {
            // help someone else if I need to
            if (head == Q->head && head.ptr->request.ptr != NULL) {
              (void)cas(&Q->head, head, {head.ptr->next.ptr, head.sn+1});
            }
            while (tail.ptr->request.ptr == NULL);  // spin
            // help snip my node
            head =  Q->head;
            if (head.ptr == tail.ptr) {
              (void)cas(&Q->head, head, {n, head.sn+1});
            }
            // data is now available; read it out and go home
            int result = tail.ptr->request.ptr->data;
            deallocate(tail.ptr->request.ptr);  deallocate(tail.ptr);
            return result;
          }
        }
      }
    } else {  // queue consists of real data
      cptr next = head.ptr->next;
      if (tail == Q->tail) {
        // head and next are consistent; read result before swinging head
        int result = next.ptr->data;
        if (cas(&Q->head, head, {next.ptr, head.sn+1})) {
          deallocate(head.ptr);  deallocate(n);
          return result;
        }
      }
    }
  }
}
```

Listing 3.2: (continued)

3.3.5 Experimental Results

In this section we compare the performance of the dual stack and dual queue to that of Treiber's lock-free stack [Tre86], the M&S lock-free queue [MiS96], and four lock-based alternatives. With Treiber's stack and the M&S queue we embed the calls to `pop` and `dequeue`, respectively, in a tight loop[6] that repeats until the operations succeed. Two lock-based alternatives, the "locked stack" and the "locked queue" employ similar loops. The remaining two alternatives are lock-based dual data structures. Like the nonblocking dual stack and dual queue, the "dual locked stack" and "dual locked queue" can contain either data or requests. All updates, however, are protected by a test-and-set lock.

Our experimental platform is a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III processors. Our benchmark creates $n + 1$ threads for an n thread test. Thread 0 executes as follows:

```
while time has not expired
   for i = 1 to 3
      insert -1 into data structure
   repeat
      pause for about 50$\mu$s
   until data structure is empty
   pause for about 50$\mu$s
```

Other threads all run the following:

```
while time has not expired
   remove val from data structure
   if val == -1
      for i = 1 to 32
         insert i into data structure
   pause for about 0.5$\mu$s
```

These conventions arrange for a series of "rounds" in which the data structure alternates between being full of requests and being full of data. Three threads, chosen more

[6]We do not test separately with exponential backoff between attempts to `pop/dequeue`. Although this would also serve to mitigate the contention that hurts performance, we know that exponential backoff has only limited scalability from our analysis of TATAS locks in Chapter 2.6. We believe that a comparable effect would be visible here, but only on a machine larger than the one we test on here.

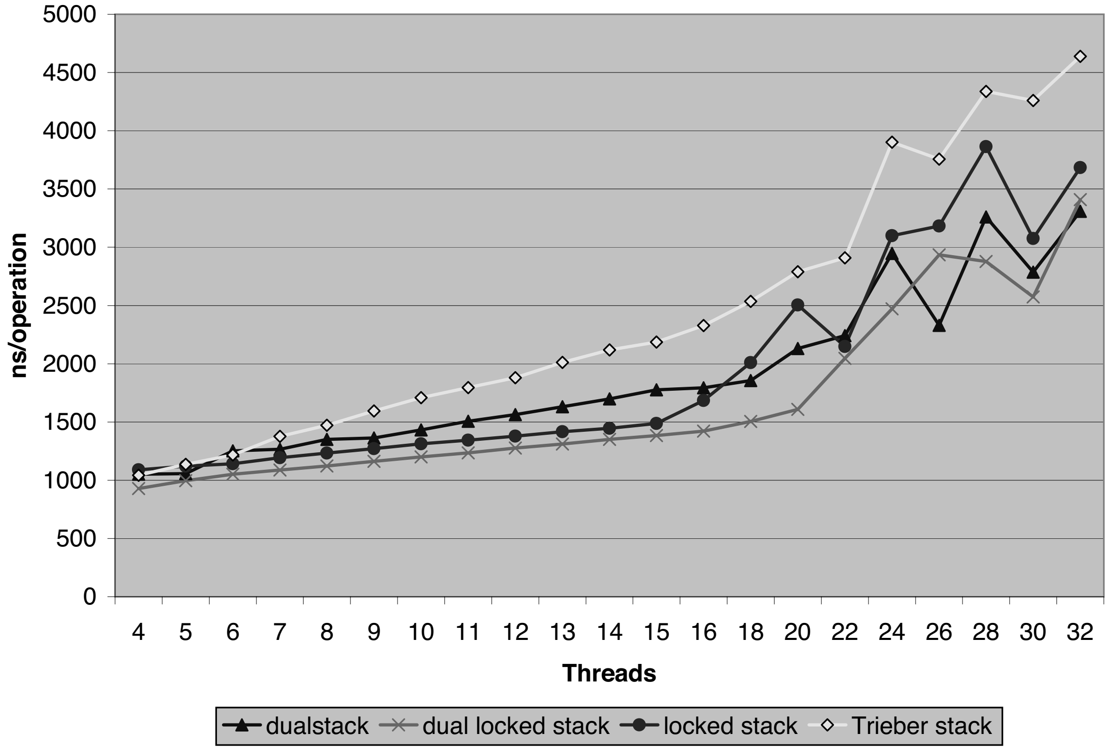


Figure 3.1: Dual stack performance evaluation: Benchmark time per operation for stack algorithms.

or less at random, prime the structure for the next round, and then join their peers in emptying it. We ran each test for two seconds, and report the minimum per-operation run time across five trials. Spot checks of longer runs revealed no anomalies. Choosing the minimum effectively discards the effects of periodic execution by kernel daemons.

Code for the various algorithms was written in C (with embedded assembly for CAS), and was compiled with `gcc` version 3.3.2 and the `-O3` level of optimization. As discussed previously, we use the fast local memory allocator from Section 2.4 [Sco02].

Stack results appear in Figure 3.1. For both lock-based and lock-free algorithms, dualism yields a significant performance improvement: At 14 worker threads the dual locked stack is about 9% faster than (takes 93% as much time as) the locked stack that retries failed `pop` calls repeatedly; the nonblocking dual stack is about 20% faster than its non-dual counterpart. In each case the lock-based stack is faster than the correspond-

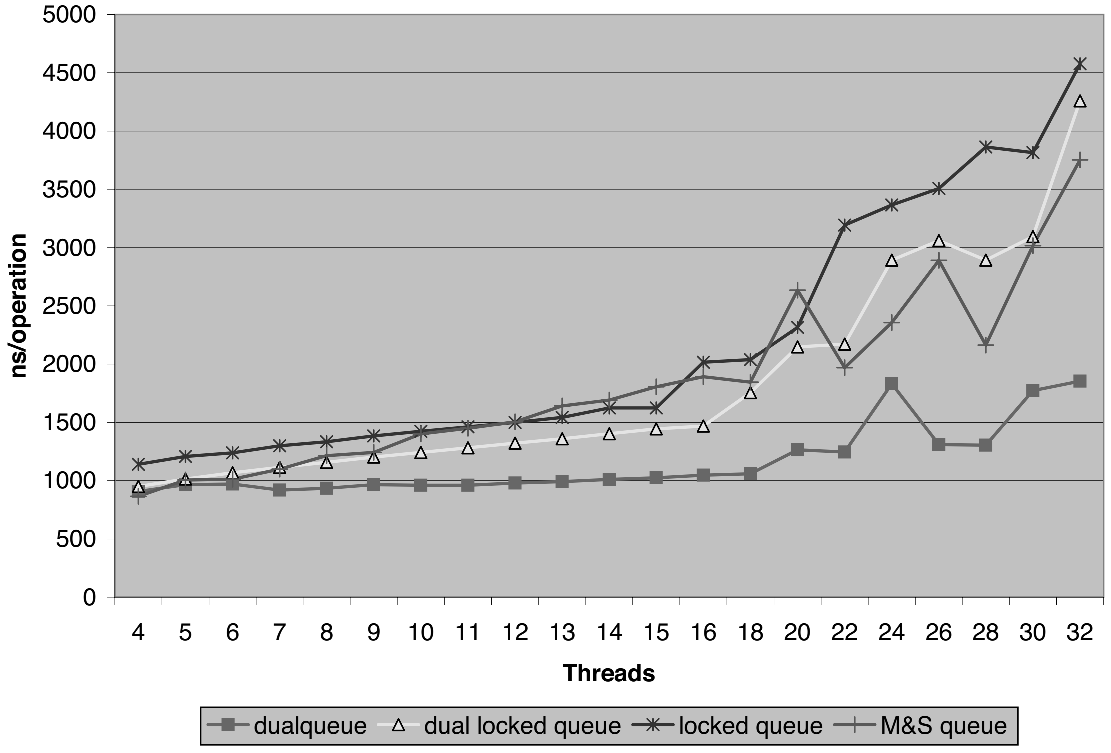


Figure 3.2: Dual queue performance evaluation: Benchmark time per operation for queue algorithms.

ing lock-free stack due, we believe, to reduced contention for the top-of-stack pointer. Nonetheless, both dual versions outperform the non-dual stacks; again, by spinning locally within their request nodes, waiters avoids hammering on the top-of-stack pointer. The resulting benefits are analogous to those obtained from local spin in queue-based locks [MeS91].

Queue results appear in Figure 3.2. Here dualism again yields significant improvements: At 14 worker threads the dual locked queue is about 14% faster than the locked queue that retries failed `dequeue` calls repeatedly; the nonblocking dual queue is more than 40% faster than its non-dual counterpart. Unlike the stacks, the nonblocking dual queue outperforms the dual locked queue by a significant margin; we attribute this difference to the potential concurrency between enqueues and dequeues. The M&S queue is slightly faster than the locked queue at low thread counts, slightly slower for

12–15 threads, and significantly faster once the number of threads exceeds the number of processors, and the lock-based algorithm begins to suffer from preemption in critical sections. As in the dual stack, we attribute this performance gain to local spinning in request nodes that reduces contention on the queue's head and tail pointers. Performance of the nonblocking dual queue is almost flat out to 16 threads (the size of the machine), and reasonable well beyond that, despite an extremely high level of contention in our benchmark; this algorithm appears to be of great potential use on any cache-coherent machine.

3.4 A Scalable Elimination-based Exchanger

The problem of *exchange channels* (sometimes known as *rendezvous channels*) arises in a variety of concurrent programs. In it, a thread t_a with datum d_a that enters the channel pairs up with another thread t_b (with datum d_b) and exchanges data such that t_a returns with d_b and t_b returns with d_a. More generally, $2N$ threads form N pairs $\langle t_{a_1}, t_{b_1}\rangle$, $\langle t_{a_2}, t_{b_2}\rangle$, $\langle t_{a_3}, t_{b_3}\rangle$, ..., $\langle t_{a_N}, t_{b_N}\rangle$ and exchange data pairwise.

In the basic exchange problem, if no partner is available immediately, thread t_a waits until one becomes available. In the *abortable* exchange problem, however, t_a specifies a patience p_a that represents a maximum length of time it is willing to wait for a partner to appear; if no partner appears within p_a μseconds, t_a returns empty-handed. Caution must be applied in implementations to ensure that a thread t_b that sees t_a just as it "gives up" and returns failure must not return d_a: Exchange must be bilateral.

Exchange channels are frequently used in parallel simulations. For example, the Promela modeling language for the SPIN model checker [Hol97] uses them to simulate interprocess communication channels. Another typical use is in operating systems and server software. In a system with one producer and one consumer, the producer might work to fill a buffer with data, then exchange it with the buffer-draining consumer.

This simultaneously bounds memory allocation for buffers and throttles the producer to generate data no faster than the consumer can process it.

In this section, we present a novel lock-free exchanger implementation that improves performance and scalability dramatically compared to previously known implementations. Our exchanger has been adopted for inclusion in the concurrency package of Java 6.

3.4.1 Elimination

Elimination is a technique introduced by Shavit and Touitou [ShT95] that improves the concurrency of data structures. It exploits the observation, for example, that one Push and one Pop, when applied with no intermediate operations to a stack data structure, yield a state identical to that from before the operations. Intuitively, then, if one could pair up Push and Pop operations, there would be no need to reference the stack data structure; they could "cancel each other out". Elimination thus reduces contention on the main data structure and allows parallel completion of operations that would otherwise require accessing a common central memory location.

More formally, one may define linearization points [HeW90] for mutually-canceling elimination operations in a manner such that no other linearization points intervene between them; since the operations effect (collectively) no change to the base data structure state, the history of operations – and its correctness – is equivalent to one in which the two operations never happened.

Although the original eliminating stack of Shavit and Touitou [ShT95] is not linearizable, follow-up work by Hendler et al. [HSY04] details one that is. Elimination has also been used for shared counters [ABH00] and even for FIFO queues [MNS05].

3.4.2 Algorithm Description

Our exchanger uses a novel combination of nonblocking dual data structures and elimination arrays to achieve high levels of concurrency. The implementation is originally based on a combination of our dual stack [ScS04b] and the eliminating stack of Hendler et al. [HSY04], though peculiarities of the exchange channel problem limit the visibility of this ancestry. Our exchanger is implemented in Java.

To simplify understanding, we present our exchanger algorithm in two parts. Section 3.4.2 first illustrates a simple exchanger that satisfies the requirements for being a lock-free dual data structure as defined in Section 3.2.5. We then describe in Section 3.4.2 the manner in which we incorporate elimination to produce a scalable lock-free exchanger.

A Simple Nonblocking Exchanger

The main data structure we use for the simple exchanger is a modification to the dual stack [ScS04b]. Additionally, we use an inner node class that consists of a reference to an `Object` offered for exchange and an `AtomicReference` representing the hole for an object. We associate one node with each thread attempting an exchange. Exchange is accomplished by successfully executing a `compare_and_swap`, updating the hole from its initial null value to the partner's node. In the event that a thread has limited patience for how long to wait before abandoning an exchange, signaling that it is no longer interested consists of executing a `compare_and_swap` on its *own* hole, updating the value from null to a `FAIL` sentinel. If this `compare_and_swap` succeeds, no other thread can successfully match the node; conversely, the `compare_and_swap` can only fail if some other thread has already matched it.

From this description, one can see how to construct a simple nonblocking exchanger. Referencing the implementation in Listing 3.3: Upon arrival, if the top-of-stack is null (line 07), we `compare_and_swap` our thread's node into it (08) and wait until either its patience expires (10–12) or another thread matches its node to us (09, 17). Alternatively, if the top-of-stack is non-null (19), we attempt to `compare_and_swap` our node into the existing node's hole (20); if successful, we then `compare_and_swap` the top-of-stack back to null (21). Otherwise, we help remove the matched node from the top of the stack; hence, the second `compare_and_swap` is unconditional.

In this simple exchanger, the initial linearization point for an in-progress swap is when the `compare_and_swap` on line 08 succeeds; this inserts a reservation into the channel for the next data item to arrive. The linearization point for a fulfilling operation is when the `compare_and_swap` on line 20 succeeds; this breaks the waiting thread's spin (lines 09–16). (Alternatively, a successful `compare_and_swap` on line 11 is the linearization point for an aborted exchange.) As it is clear that the waiter's spin accesses no remote memory locations and that both inserting and fulfilling reservations are lock-free (a `compare_and_swap` in this case can only fail if another has succeeded), the simple exchanger constitutes a lock-free implementation of the exchanger dual data structure as defined in Section 3.2.5.

Adding Elimination

Although the simple exchanger from the previous section is nonblocking, it will not scale very well: The top-of-stack pointer is a hotspot for contention. This scalability problem can be resolved by adding an elimination step to the simple exchanger from Listing 3.3. Source code for the full exchanger appears in Listing 3.4.

In order to support elimination, we replace the single top-of-stack pointer with an `arena` (array) of $(P+1)/2$ Java SE 5.0 `AtomicReferences`, where P is the number of processors in the runtime environment. Logically, the reference in position 0 is the top-of-stack; the other references are simply locations at which elimination can occur.

```
Object exchange(Object x, boolean timed,
  long patience) throws TimeoutException {
  boolean success = false;
  long start = System.nanotime();
  Node mine = new Node(x);
  for (;;) {
    Node top = stack.getTop();
    if (top == null) {
      if (stack.casTop(null, mine)) {
        while (null == mine.hole) {
          if (timedOut(start, timed, patience) {
            if (mine.casHole(null, FAIL))
              throw new TimeoutException();
            break;
          }
          /* else spin */
        }
        return mine.hole.item;
      }
    } else {
      success = top.casHole(null, mine);
      stack.casTop(top, null);
      if (success)
        return top.item;
    }
  }
}
```

Listing 3.3: A simple lock-free exchanger

Following the lead of Hendler et al.[HSY04], we incorporate elimination with backoff when encountering contention at top-of-stack. As in their work, by only attempting elimination under conditions of high contention, we incur no additional overhead in the common no-contention case.

Logically, in each iteration of a main loop, we attempt an exchange in the 0th arena position exactly as in the simple exchanger. If we successfully insert or fulfill a reservation, we proceed exactly as before. The difference, however, comes when a `compare_and_swap` fails. Now, instead of simply retrying immediately at the top-of-stack, we back off to attempt an exchange at a random secondary arena location. In contrast to exchanges at `arena[0]`, we limit the length of time we wait with a reservation in the remainder of the arena to a value significantly smaller than our overall patience. After canceling the secondary reservation, we return to `arena[0]` for another iteration of the loop.

In iteration i of the main loop, the arena location at which we attempt a secondary exchange is selected randomly from the range $1..b$, where b is the lesser of i and the arena size. Hence, the first secondary exchange is always at `arena[1]`, but with each iteration of the main loop, we increase the range of potential backoff locations until we are randomly selecting a backoff location from the entire arena. Similarly, the length of time we wait on a reservation at a backoff location is randomly selected from the range $0..2^{(b+k)} - 1$, where k is a base for the exponential backoff.

From a correctness perspective, the same linearization points as in the simple exchanger are again the linearization points for the eliminating exchanger; however, they can occur at any of the arena slots, not just at a single top-of-stack. Although the eliminating stack can be shown to support LIFO ordering semantics, we have no particular ordering semantics to respect in the case of an exchange channel: Any thread in the channel is free to match to any other thread, regardless of when it entered the channel.

The use of timed backoff accentuates the probabilistic nature of limited-patience `exchange`. Two threads that attempt an exchange with patience zero will only discover each other if they both happen to probe the top of the stack at almost exactly the same time. However, with increasing patience levels, the probability decreases that they will fail to match after temporally proximate arrivals. Other parameters that influence this fall include the number of processors and threads, hardware instruction timings, and the accuracy and responsiveness of timed waits. Our experimental evidence suggests that in modern environments, the chance of backoff arena use causing two threads to miss each other is far less than the probability of thread scheduling or garbage collection delaying a blocked thread's wakeup for long enough to miss a potential match.

Pragmatics

Our exchanger implementation reflects a few additional pragmatic considerations to maintain good performance:

First, we use an array of `AtomicReferences` rather than a single `Atomic-ReferenceArray`. Using a distinct reference object per slot helps avoid some false sharing and cache contention, and places responsibility for their placement on the Java runtime system rather than on this class.

Second, the time constants used for exponential backoff can have a significant effect on overall throughput. We empirically chose a base value to be just faster than the minimum observed round-trip overhead, across a set of platforms, for timed `parkNanos()` calls (which temporarily remove a thread from the active scheduling list) on already-signaled threads. By so doing, we have selected the smallest value that does not greatly underestimate the actual wait time. Over time, future versions of this class might be expected to use smaller base constants.

3.4.3 Experimental Results

In this section, we present an empirical analysis of our exchanger implementation.

Benchmarks

We present experimental results for two benchmarks. The first is a microbenchmark in which threads swap data in an exchange channel as fast as they can. The second exemplifies the way that an exchange channel might be used in a real-world application. It consists of a parallel implementation of a solver for the traveling salesman problem, implemented via genetic algorithms. It accepts as parameters a number of cities C, a population size P, and a number of generations G, in each of which B breeders mate and create B children to replace B individuals that die (the remaining $P - B$ individuals carry forward to the next generation). Breeders enter a central exchange channel one or more times to find a partner with which to exchange genes (a subset of the circuit); the number of partners ranges from four down to one in a simulated annealing fashion. Between randomization of the order of breeders and the (semi-)

```java
public class Exchanger<V> {
  private static final int SIZE =
    (Runtime.getRuntime().availableProcessors() + 1) / 2;
  private static final long BACKOFF_BASE = 128L;
  static final Object FAIL = new Object();
  private final AtomicReference[] arena;
  public Exchanger() {
    arena = new AtomicReference[SIZE + 1];
    for (int i = 0; i < arena.length; ++i)
      arena[i] = new AtomicReference();
  }

  public V exchange(V x) throws InterruptedException {
    try {
      return (V)doExchange(x, false, 0);
    } catch (TimeoutException cannotHappen) {
      throw new Error(cannotHappen);
    }
  }
  public V exchange(V x, long timeout, TimeUnit unit)
    throws InterruptedException, TimeoutException {
    return (V)doExchange(
      x, true, unit.toNanos(timeout));
  }

  private Object doExchange(
    Object item, boolean timed, long nanos)
    throws InterruptedException, TimeoutException {
    Node me = new Node(item);
    long lastTime = (timed)? System.nanoTime() : 0;
    int idx = 0;
    int backoff = 0;

    for (;;) {
      AtomicReference<Node> slot =
        (AtomicReference<Node>)arena[idx];

      // If this slot is occupied, an item is waiting...
      Node you = slot.get();
      if (you != null) {
        Object v = you.fillHole(item);
        slot.compareAndSet(you, null);
        if (v != FAIL)     // ... unless it’s cancelled
          return v;
      }

      // Try to occupy this slot
      if (slot.compareAndSet(null, me)) {
        Object v = ((idx == 0)?
          me.waitForHole(timed, nanos) :
          me.waitForHole(true, randomDelay(backoff)));
        slot.compareAndSet(me, null);
        if (v != FAIL)
          return v;
        if (Thread.interrupted())
          throw new InterruptedException();
        if (timed) {
          long now = System.nanoTime();
          nanos -= now - lastTime;
          lastTime = now;
          if (nanos <= 0)
            throw new TimeoutException();
        }
```

Listing 3.4: The full lock-free exchanger

```
        me = new Node(item);
        if (backoff < SIZE - 1)
          ++backoff;
        idx = 0;   // Restart at top
      }

      else // Retry with a random non-top slot <= backoff
        idx = 1 + random.nextInt(backoff + 1);
    }
  }

  private long randomDelay(int backoff) {
    return ((BACKOFF_BASE << backoff) - 1) &
      random.nextInt();
  }
  static final class Node
    extends AtomicReference<Object> {
    final Object item;
    final Thread waiter;
    Node(Object item) {
      this.item = item;
      waiter = Thread.currentThread();
    }

    Object fillHole(Object val) {
      if (compareAndSet(null, val)) {
        LockSupport.unpark(waiter);
        return item;
      }
      return FAIL;
    }

    Object waitForHole(
      boolean timed, long nanos) {
      long lastTime = (timed)?
        System.nanoTime() : 0;
      Object h;
      while ((h = get()) == null) {
        // If interrupted or timed out, try to
        // cancel by CASing FAIL as hole value.
        if (Thread.currentThread().isInterrupted() ||
          (timed && nanos <= 0)) {
          compareAndSet(null, FAIL);
        } else if (!timed) {
          LockSupport.park();
        } else {
          LockSupport.parkNanos(nanos);
          long now = System.nanoTime();
          nanos -= now - lastTime;
          lastTime = now;
        }
      }
      return h;
    }
  }
}
```

Listing 3.4: (continued)

nondeterministic manner in which pairings happen in the exchange channel, we achieve a diverse set of matings with high probability.

Methodology

All results were obtained on our 16-processor SunFire 6800. We tested each benchmark with both our new exchanger and the Java SE 5.0 *java.util.concurrent.Exchanger*[7] in Sun's Java SE 5.0 HotSpot VM, ranging from 2 to 32 threads. For both tests, we compute the average of three test runs.

For the traveling salesman application, we used 100 cities, a population of 1000 chromosomes, and 200 breeders. We measure the total wall-clock time required to complete 20000 generations and calculate the total number of generations per second achieved at each thread level.

Figure 3.3 displays throughput and the rate at which exchanges are successful for our microbenchmark analysis of exchangers. Figure 3.4 presents total running time, as a function of the number of threads, and the generation completion throughput for our parallel genetic algorithm-based traveling salesman solver.

Discussion

As can be seen from the top half of Figure 3.3, our exchanger outperforms the Java SE 5.0 *Exchanger* by a factor of two at two threads up to a factor of 50 by 10 threads. The performance of the Java SE 5.0 *Exchanger* degrades as the number of threads participating in swaps increases. This lack of scalability may be attributed in part to the coarse-grained locking strategy. Another explanation for this difference may be seen in the bottom half of Figure 3.3. While our nonblocking exchanger is able

[7]Actually, the Java SE 5.0 *Exchanger* contains a flaw in which a wake-up signal is sometimes delivered to the wrong thread, forcing a would-be exchanger to time out before noticing it has been matched. For our tests, we compare instead to a modified version that corrects this issue.

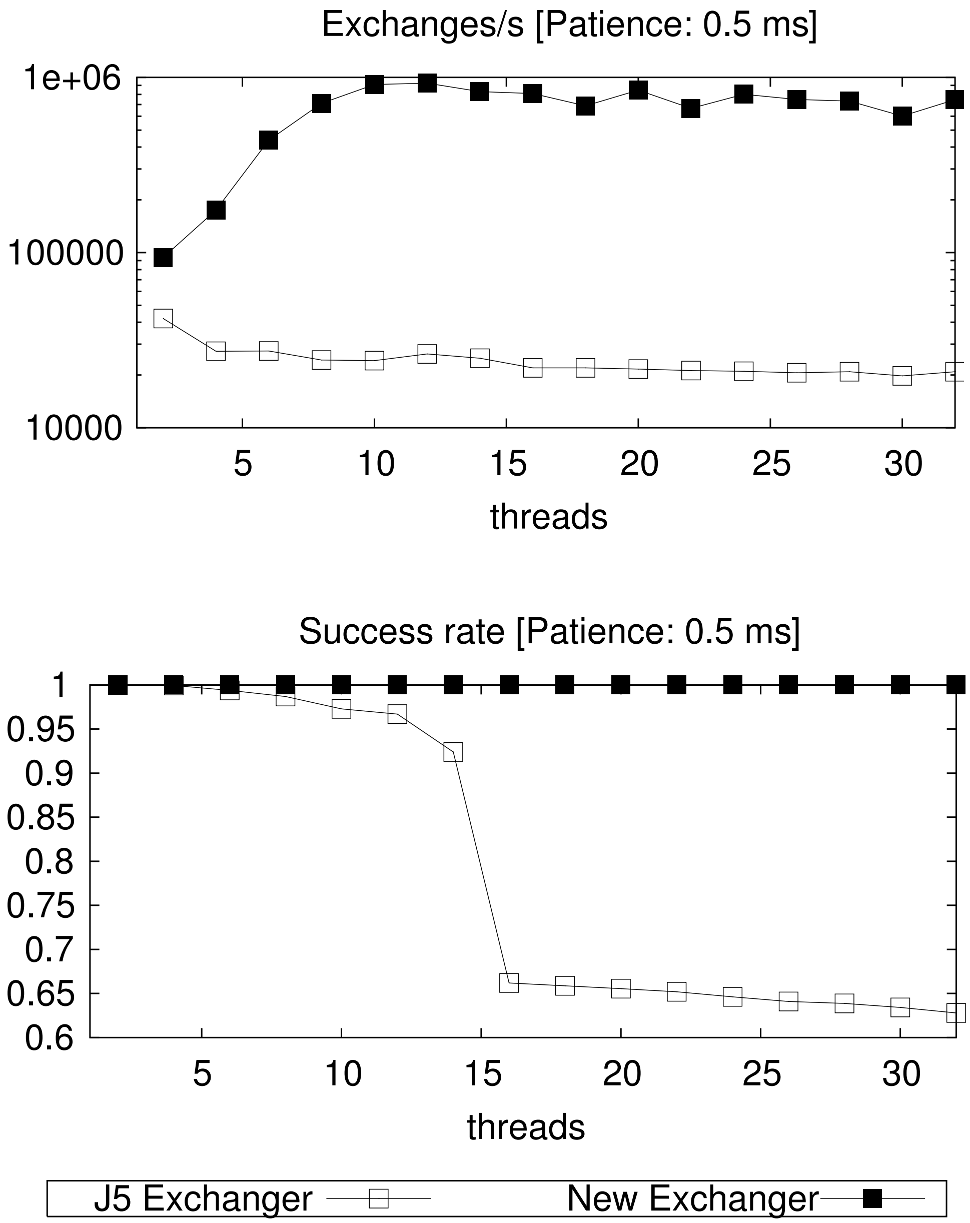


Figure 3.3: Exchanger performance evaluation: Microbenchmarked throughput (top) and success rate (bottom). Note that the throughput graph is in log scale.

to maintain nearly 100% success in exchange operations, the Java SE 5.0 *Exchanger* gets dramatically worse as the number of active threads increases, particularly in the presence of preemption (beyond 16 threads).

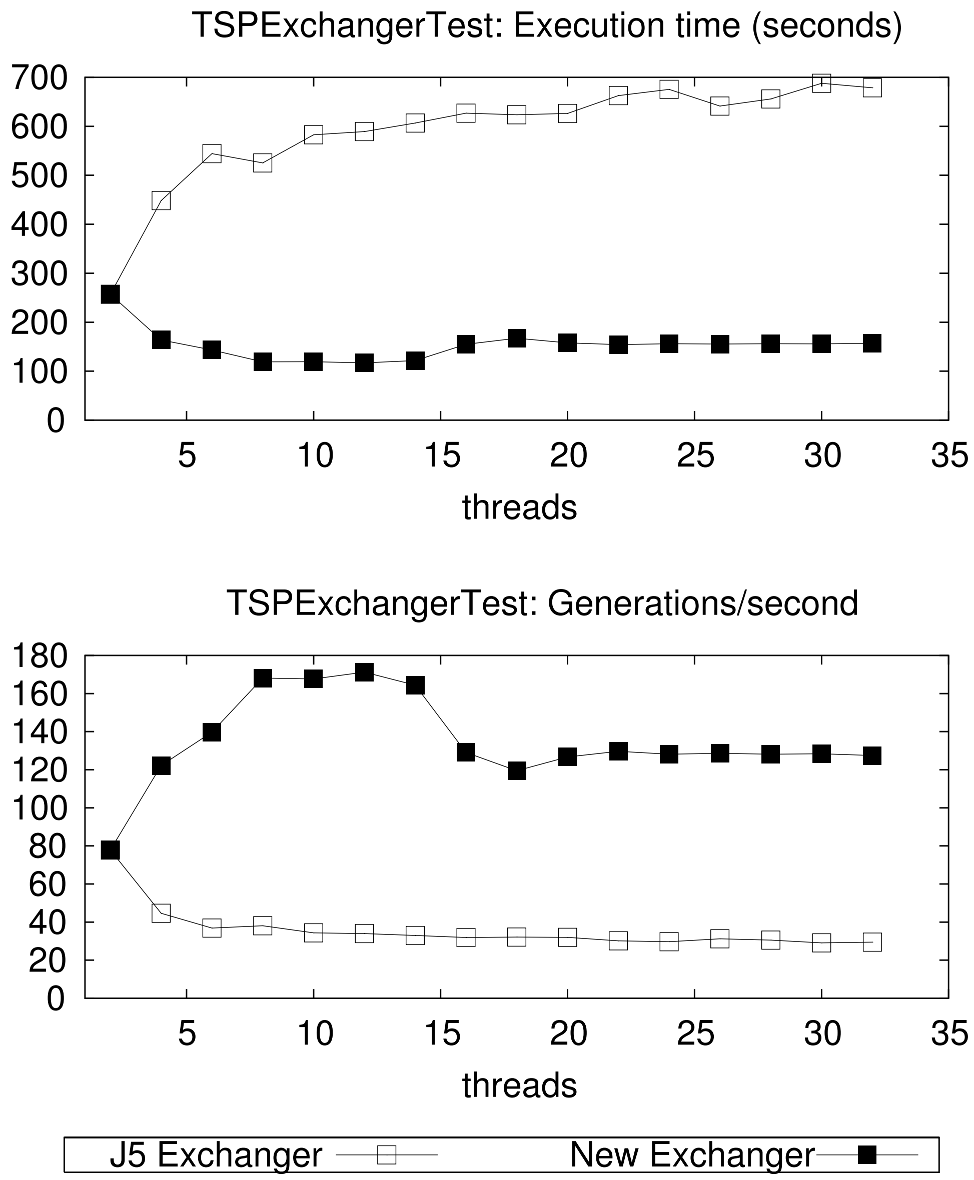


Figure 3.4: Exchanger performance evaluation: Traveling salesman benchmark. Total Execution Time (top) and Generations per second throughput (bottom).

For the traveling salesman application, we see no difference in the total running time (Figure 3.4 top) at two threads, but by 10 threads the difference in running time is nearly a factor of five. When we look at the throughput rate of computation (generations per second) in the bottom half of Figure 3.4, we see that again, our exchanger scales more-

or-less linearly up to 8 threads, and continues to gain parallel speedup through 12, but the Java SE 5.0 *Exchanger* degrades in performance as the number of threads increases. The drop-off in throughput for our exchanger we see beginning at 16 threads reflects the impact of preemption in slowing down exchanges and inter-generation barriers

Field Notes: Multi-party Exchange

Consider the exchanger used with more than two threads. In our parallel traveling salesman implementation (pseudocode for a piece of which appears in Listing 3.5), each thread is assigned a certain number of individuals to breed, and the breedings are conducted in parallel. As mentioned earlier, each breeding consists of swapping genes with a partner found from a central exchange channel.

With two threads, no complications arise, but beginning at four, a problem appears wherein all but one thread can meet up enough times to finish breeding, leaving one thread high and dry. Consider the following example, in which four threads $(T_1..T_4)$ each have three elements (a, b, c) to swap. Suppose that swaps occur according to the following schedule:

$\langle T_1(a), T_2(a) \rangle$
$\langle T_1(b), T_3(a) \rangle$
$\langle T_2(b), T_3(b) \rangle$
$\langle T_1(c), T_2(c) \rangle$
$\langle T_3(c), T_4(a) \rangle$

Now, T_4 still needs to swap b and c, but has no one left to swap with. Although one could use a barrier between trips to the exchanger to keep threads synchronized, this would be an exceedingly high-overhead solution; it would hurt scalability and performance. Further, it would result in more deterministic pairings between threads in our traveling salesman application, which is undesirable in genetic algorithms.

```
  for (int 1 = 0; i < numToBreed; i++) {
    Chromosome parent = individuals[breeders[first+i]];
    try {
      Chromosome child = new Chromosome(parent);
*1*   Chromosome peer = x.exchange(child, 100,
        TimeUnit.MICROSECONDS);
      children[i] = child.reproduceWith(peer, random);
    } catch (TimeoutException e) {
*2*   if (1 == barrier.getPartiesRemaining()) {
        // No peers left, so we mate with ourselves
*3*     mateWithSelf(i, numToBreed, children);
        break;
      }
*4*   // Spurious failure; retry the breeding
      --i;
    }
  }
  barrier.await();
```

Listing 3.5: Multi-party exchange in the traveling salesman problem

Instead, we detect this case by limiting how long would-be breeders wait in the exchange channel. If they time out from the breeding (*1*), we enter a catch block where we check a barrier to see if we're the only ones left (*2*). If not, we retry the breeding (*4*); otherwise, we know that we're the only thread left and we simply recombine within the breeders we have left (*3*).

It is unfortunate that an external checking idiom is required to make use of the exchange channel with multiple threads. Our traveling salesman benchmark already needs a barrier to ensure that one generation completes before the next begins, so this adds little overhead in our specific case. However, the same cannot be readily guaranteed across all potential multi-party swap applications.

On the other hand, suppose we had a channel that allows an exchange party to be one of two colors (red or blue) and that constrains exchanges to being between parties of dissimilar color. Such an exchanger could be built, for example, as a shared-memory implementation of generalized input-output guards [Ber80] for Hoare's Communicating Sequential Processes (CSP) [Hoa78]. Then, by assigning `RED` to threads with odd ID and `BLUE` to threads with even ID, we could simplify the code as shown in Listing 3.6.

```
for (int 1 = 0; i < numToBreed; i++) {
  Chromosome parent = individuals[breeders[first+i]];
  Chromosome child = new Chromosome(parent);
  Exchanger.Color clr = ((1 == tid & 1) ? RED : BLUE;
  Chromosome peer = x.exchange(child, clr);
  children[i] = child.reproduceWith(peer, random);
}
```

Listing 3.6: Multi-party exchange with a red-blue exchanger

Note that we no longer need timeout: Assuming the number of threads and breeders are both even, an equal number will swap with red as with blue; stranded exchanges are no longer possible. (The slightly reduced non-determinism in breeding seems a small price to pay for simpler code and for eliminating the determinism that occurs when a thread must breed with itself.) A red-blue exchanger also generalizes producer-consumer buffer exchange to multiple producers and multiple consumers. By simply marking producers blue and consumers red, swaps will always consist of a producer receiving an empty buffer and a consumer receiving a full one.

Implementing a red-blue exchanger would be a relatively straightforward extension to our code. In particular, rather than assuming that any pair of threads match in `arena[0]`, we could switch to a full implementation of a dual stack, using `Push()` for red threads and `Pop()` for blue. We would similarly need to update elimination in the nonzero arena slots. No other changes to our algorithm would be needed, however, to support bi-chromatic exchange.

3.5 Scalable Synchronous Queues

A synchronous queue (perhaps better known as a “synchronous channel”) is one in which each producer presenting an item (via a `put` operation) must wait for a consumer to `take` this item, and vice versa. For decades, synchronous queues have played a prominent role in both the theory and practice of concurrent programming. They constitute the central synchronization primitive of Hoare’s CSP [Hoa78] and of languages derived from it, and are closely related to the *rendezvous* of Ada. They are also widely used in message-passing software and in hand-off designs [And91].

Unfortunately, the design-level tractability of synchronous queues has often come at the price of poor performance. “Textbook” algorithms for implementing synchronous queues contain a series of potential contention or blocking points in every `put` or `take`. (We consider in this paper only those synchronous queues operating within a single multithreaded program; not across multiple processes or distributed nodes.) For example, Listing 3.7 shows one of the most commonly used implementations, due to Hanson [Han97], which uses three separate semaphores.

Such heavy synchronization burdens are especially significant on contemporary multiprocessors and their operating systems, in which the blocking and unblocking of threads tend to be very expensive operations. Moreover, even a series of uncontended semaphore operations usually requires enough costly machine-level atomic and barrier (fence) instructions to incur substantial overhead.

It is also difficult to extend this and other “classic” synchronous queue algorithms to support other common operations. These include `poll`, which takes an item only if a

```
public class HansonSQ<E> {
  E item = null;
  Semaphore sync = new Semaphore(0);
  Semaphore send = new Semaphore(1);
  Semaphore recv = new Semaphore(0);

  public E take() {
    recv.acquire();
    E x = item;
    sync.release();
    send.release();
    return x;
  }

  public void put(E x) {
    send.acquire();
    item = x;
    recv.release();
    sync.acquire();
  }
}
```

Listing 3.7: Hanson’s synchronous queue

producer is already present, and `offer` which fails unless a consumer is waiting. Similarly, many applications require the ability to time out if producers or consumers do not appear within a certain *patience* interval or if the waiting thread is asynchronously interrupted. One of the *java.util.concurrent.ThreadPoolExecutor* implementations uses all of these capabilities: Producers deliver tasks to waiting worker threads if immediately available, but otherwise create new worker threads. Conversely, worker threads terminate themselves if no work appears within a given keep-alive period (or if the pool is shut down).

Additionally, applications using synchronous queues vary in their need for *fairness*: Given multiple waiting producers, it may or may not be important to an application whether the one waiting the longest (or shortest) will be the next to pair up with the next arriving consumer (and vice versa). Since these choices amount to application-level policy decisions, algorithms and implementations should minimize imposed constraints. For example, while fairness is often considered a virtue, a thread pool normally runs better if the most-recently-used waiting worker thread is usually the next to receive new work, due to footprint retained in the cache and VM system.

In the remainder of this section, we present synchronous queue algorithms that combine a rich programming interface with very low intrinsic overhead. More specifically, our algorithms avoid all blocking other than that which is intrinsic to the notion of synchronous handoff: A producer thread must wait until a consumer appears (and vice-versa), but there is no other way for one thread's delay to impede another's progress. We describe two algorithmic variants: a *fair* algorithm that ensures strict FIFO ordering and an *unfair* algorithm that is actually based on a LIFO stack. Our synchronous queues have been adopted for inclusion in the concurrency package of Java 6.

```
00 public class NaiveSQ<E> {
01   boolean putting = false;
02   E item = null;
03
04   public synchronized E take() {
05     while (item == null)
06       wait();
07     E e = item;
08     item = null;
09     notifyAll();
10     return e;
11   }
12
13   public synchronized void put(E e) {
14     if (e == null) return;
15     while (putting)
16       wait();
17     putting = true;
18     item = e;
19     notifyAll();
20     while (item != null)
21       wait();
22     putting = false;
23     notifyAll();
24   }
25 }
```

Listing 3.8: Naive synchronous queue

3.5.1 Algorithm Descriptions

Classic Synchronous Queues

Perhaps the simplest implementation of synchronous queues is the naive monitor-based algorithm that appears in Listing 3.8. In this implementation, a single monitor serializes access to a single item and to a `putting` flag that indicates whether a producer has currently supplied data. Producers wait for the flag to be clear (lines 15–16), set the flag (17), insert an item (18), and then wait until a consumer takes the data (20–21). Consumers await the presence of an item (05–06), take it (07), and mark it as taken (08) before returning. At each point where their actions might potentially unblock another thread, producer and consumer threads awaken all possible candidates (09, 20, 24). Unfortunately, this approach results in a number of wake-ups quadratic in the number of waiting producer and consumer threads; coupled with the high cost of blocking or unblocking a thread, this results in very poor performance.

Hanson's synchronous queue (Listing 3.7) improves upon the naive approach by using semaphores to target wake-ups to only the single producer or consumer thread that an operation has unblocked. However, as noted in Section 3.5, it still incurs the overhead of three separate synchronization events per transfer for each of the producer and consumer; further, it normally blocks at least once per `put` or `take` operation. It is possible to streamline some of these synchronization points in common execution scenarios by using a fast-path acquire sequence [Lam87]. Such a version appears in early releases of the *dl.util.concurrent* package, which later evolved into *java.util.concurrent*. However, such minor incremental changes improve performance by only a few percent in most applications.

The Java SE 5.0 Synchronous Queue

The Java SE 5.0 synchronous queue (Listing 3.9) uses a pair of queues (in fair mode; stacks for unfair mode) to separately hold waiting producers and consumers. This approach improves considerably on semaphore-based approaches. First, in the case where a consumer finds a waiting producer or a producer finds a waiting consumer, the new arrival needs to perform only one synchronization operation, acquiring a lock that protects both queues (line 18 or 33). Even if no counterpart is waiting, the only additional synchronization required is to await one (25 or 40). A complete handoff thus requires only three synchronization operations, compared to six incurred by Hanson's algorithm. In particular, using a queue instead of a semaphore allows producers to publish data items as they arrive (line 21) instead of having to first wake up from blocking on a semaphore; consumers need not wait.

```java
public class Java5SQ<E> {
  ReentrantLock qlock = new ReentrantLock();
  Queue waitingProducers = new Queue();
  Queue waitingConsumers = new Queue();

  static class Node
    extends AbstractQueuedSynchronizer {
    E item;
    Node next;

    Node(Object x) { item = x; }
    void waitForTake() { /* (uses AQS) */ }
    E waitForPut()  { /* (uses AQS) */ }
  }

  public void put(E e) {
    Node node;
    boolean mustWait;
    qlock.lock();
    node = waitingConsumers.pop();
    if ((mustWait = (node == null)))
      node = waitingProducers.push(e);
    qlock.unlock();

    if (mustWait)
      node.waitForTake();
    else
      node.item = e;
  }

  public E take() {
    Node node;
    boolean mustWait;
    qlock.lock();
    node = waitingProducers.pop();
    if ((mustWait = (node == null)))
      node = waitingConsumers.push(null);
    qlock.unlock();

    if (mustWait)
      return node.waitForPut();
    else
      return node.item;
  }
}
```

Listing 3.9: The Java SE 5.0 *SynchronousQueue* class, fair (queue-based) version. The unfair version uses stacks instead of queues, but is otherwise identical. (For clarity, we have omitted timeout, details of the way in which AbstractQueuedSynchronizers are used, and code to generalize waitingProducers and waitingConsumers to either stacks or queues.)

Combining Dual Data Structures with Synchronous Queues

A key limitation of the Java SE 5.0 *SynchronousQueue* class is its reliance on a single lock to protect both queues. Coarse-grained synchronization of this form is well known for introducing serialization bottlenecks. By creating nonblocking implementations, we eliminate this impediment to scalability.

Our new algorithms add support for timeout and for bidirectional synchronous waiting to our previous nonblocking dual queue and dual stack algorithms [ScS04b]. We describe the new algorithms in three steps. First, Section 3.5.1 reviews our earlier dual stack and dual queue and presents the modifications needed to make them synchronous. Second, Section 3.5.1 sketches the manner in which we add timeout support. Finally, Section 3.5.1 discusses additional pragmatic issues. Throughout the discussion, we present fragments of code to illustrate particular features; full source appears at the end of this section in Listing 3.14 and online at `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/SynchronousQueue.java`.

Basic Synchronous Dual Queues and Stacks

Our dual stack and dual queue algorithms from Section 3.3 already block when a consumer arrives before a producer; extending them to support synchronous handoff is thus a matter of arranging for producers to block until a consumer arrives. We can do this in the synchronous dual queue by simply having a producer block on its data pointer until a consumer updates it to null (implicitly claiming the data). For the synchronous dual stack, we extend the annihilating approach to include *fulfilling requests* that pair with data at the top of the stack in the same manner that fulfilling data nodes pair with requests.

```
00 class Node { E data; Node next; ... }
01
02 void enqueue(E e) {
03     Node offer = new Node(e, Data);
04
05     while (true) {
06         Node t = tail;
07         Node h = head;
08         if (h == t || !t.isRequest()) {
09             Node n = t.next;
10             if (t == tail) {
11                 if (null != n) {
12                     casTail(t, n);
13                 } else if (t.casNext(n, offer)) {
14                     casTail(t, offer);
15                     while (offer.data == e)
16                         /* spin */;
17                     h = head;
18                     if (offer == h.next)
19                     casHead(h, offer);
20                     return;
21                 }
22             }
23         } else {
24             Node n = h.next;
25             if (t != tail || h != head || n == null)
26                 continue; // inconsistent snapshot
27             boolean success = n.casData(null, e);
28             casHead(h, n);
29             if (success)
30                 return;
31         }
32     }
33 }
```

Listing 3.10: Synchronous dual queue: Spin-based enqueue. `Dequeue` is symmetric except for the direction of data transfer. (For clarity, code to support timeout is omitted.)

The Synchronous Dual Queue

Listing 3.10 shows the `enqueue` method of the synchronous dual queue. (`Dequeue` is symmetric except for the direction of data transfer.) To enqueue, we first read the head and tail pointers of the queue (lines 06–07). From here, there are two main cases. The first occurs when the queue is empty (`h == t`) or contains data (line 08). We read the next pointer for the tail-most node in the queue (09). If all values read are mutually consistent (10) and the queue's tail pointer is current (11), we attempt to insert our offering at the tail of the queue (13–14). If successful, we wait until a consumer signals that it has claimed our data (15–16), which it does by updating our node's data pointer to null. Then we help remove our node from the head of the queue and return (18–20). The initial linearization point for this code path occurs at line 13 when we successfully insert our offering into the queue; the final linearization point occurs when we notice at line 15 that our data has been taken.

The other case occurs when the queue consists of reservations (requests for data). In this case, we first read the head node's successor (24) and verify consistency (25). Then, we attempt to supply our data to the head-most reservation (27). If this succeeds, we dequeue the reservation (28) and return (30). If it fails, we need to go to the next reservation, so we dequeue the head-most one anyway (28) and retry the entire operation (32, 05). The linearization point for this code path occurs when we successfully supply data to a waiting consumer at line 27; this also corresponds to the final linearization point for the consumer.

The Synchronous Dual Stack

Code for the synchronous dual stack's `push` operation appears in Listing 3.11. (`Pop` is symmetric except for the direction of data transfer.) We begin by reading the node at the top of the stack (line 06). The three main conditional branches (beginning at lines 07, 17, and 26) correspond to the type of node we find.

The first case occurs when the stack is empty or contains only data (line 07). We attempt to insert a new datum (09), and wait for a consumer to claim our data (11–12) before returning. The initial linearization point for this code path occurs when we push our datum at line 09; the final linearization point occurs when our data has been taken at line 11.

The second case occurs when the stack contains (only) requests for data (17). We attempt to place a fulfilling datum on the top of the stack (19); if we succeed, any other thread that wishes to perform an operation must now help us fulfill the request before proceeding to its own work. We then read our way down the stack to find the successor

```
00 class Node { E data; Node next, match; ... }
01
02 void push(E e) {
03     Node f, d = new Node(e, Data);
04
05     while (true) {
06         Node h = head;
07         if (null == h || h.isData()) {
08             d.next = h;
09             if (!casHead(h, d))
10                 continue;
11             while (d.match == null)
12                 /* spin */;
13             h = head;
14             if (null != h && d == h.next)
15                 casHead(h, d.next);
16             return;
17         } else if (h.isRequest()) {
18             f = new Node(e, Data | Fulfilling, h);
19             if (!casHead(h, f))
20                 continue;
21             h = f.next;
22             Node n = h.next;
23             h.casMatch(null, f);
24             casHead(f, n);
25             return;
26         } else { // h is fulfilling
27             Node n = h.next;
28             Node nn = n.next;
29             n.casMatch(null, h);
30             casHead(h, nn);
31         }
32     }
33 }
```

Listing 3.11: Synchronous dual stack: Spin-based annihilating push. `Pop` is symmetric except for the direction of data transfer. (For clarity, code for timeout is omitted.)

node to the reservation we're fulfilling (21–22) and mark the reservation fulfilled (23). Note that our `compare_and_swap` could fail if another thread helps us and performs it first. Finally, we pop both the reservation and our fulfilling node from the stack (24) and return. The linearization point for this code path is at line 19, when we push our fulfilling datum above a reservation; it corresponds to the final linearization point for a `pop` operation.

The remaining case occurs when we find another thread's fulfilling datum or request node (26) at the top of the stack. We must complete the pairing and annihilation of the top two stack nodes before we can continue our own work. We first read our way down the stack to find the data or request node for which the fulfillment node is present (27–

28) and then we mark the underlying node as fulfilled (29) and pop the paired nodes from the stack (30).

Supporting Timeout

Although the algorithms presented in Section 3.5.1 are complete implementations of synchronous queues, real systems require the ability to specify limited patience so that a producer (or consumer) can time out if no consumer (producer) arrives soon enough to take (provide) our datum. As we noted earlier, Hanson's synchronous queue offers no simple way to do this. A key benefit of our new algorithms is that they support timeout in a relatively straightforward manner.

In the synchronous dual queue, recall that a producer blocks on its data pointer if it finds data in the queue. A consumer that finds the producer's data node head-most in the queue attempts an atomic update to clear the data pointer. If the `compare_and_swap` fails, the consumer assumes that another consumer has taken this datum, so it helps clear the producer's data node from the queue. To support timeout, therefore, it suffices for the producer to clear its own data pointer and leave; when a consumer eventually finds the abandoned node, it will remove it through the existing helping mechanism. To resolve the race wherein the producer attempts to time out at the same time as a consumer attempts to claim its data, the producer needs to clear its data node pointer with an atomic `compare_and_swap`. Similarly, a consumer waiting on the data pointer in a reservation node needs to `compare_and_swap` it to a special `ABANDONED` sentinel value and the next producer that finds its request will helpfully remove it from the queue.

Supporting timeout in the synchronous dual stack is similar to supporting it in the synchronous dual queue, but annihilation complicates matters somewhat. Specifically, if a request or data node times out, then a fulfilling node can eventually rest just above it in the stack. When this happens, the abandoned node must be deleted from mid-stack so that the fulfilling node can pair to a request below it (and other threads need to be aware of this need when helping). But what if it was the last data or request node in the stack? We now have a stack consisting of just a fulfillment node. As this case is easy to detect, we handle it by atomically setting the stack pointer to null (which can also be helped by other threads) and having the fulfilling thread start over. Finally, we adopt the requirement that the fulfilling thread cannot time out so long as it has a fulfilling node atop the stack.

Pragmatics

Our synchronous queue implementations reflect a few additional pragmatic considerations to maintain good performance. First, because Java does not allow one to set flag bits in pointers, we add a word to nodes in our synchronous queues within which we mark mode bits. We chose this technique over two primary alternatives. The class *java.util.concurrent.AtomicMarkableReference* allows direct association of tag bits with a pointer, but exhibits very poor performance. Using run-time type identification (RTTI) to distinguish between multiple subclasses of the Node classes would similarly allow us to embed tag bits in the object type information. While this approach performs well in isolation, it increases long-term pressure on the JVM's memory allocation and garbage collection routines by requiring construction of a new node after each contention failure.

Adding timeout support to the original dual stack and dual queue [ScS04b] requires careful management of memory ownership to ensure that cancelled nodes are reclaimed properly. Java's garbage collection removes this burden from the implementations we present in this paper. On the other hand, we must take care to "forget" references to data, nodes, and threads that might be retained for a long time by blocked threads (preventing the garbage collector from reclaiming them).

For sake of clarity, the synchronous queues we described earlier in this section blocked with busy-wait spinning to await a counterpart consumer. In practice, however, busy-wait is useless overhead on a uniprocessor and can be of limited value on

```
void clean(Node s) {
    Node past = s.next;
    if (past != null && past.isCancelled())
        past = past.next;

    Node p;
    while ((p = head) != null && p != past &&
            p.isCancelled())
        casHead(p, p.next);

    while (p != null && p != past) {
        Node n = p.next;
        if (n != null && n.isCancelled())
            p.casNext(n, n.next);
        else
            p = n;
    }
}
```

Listing 3.12: Synchronous dual stack: Cleaning cancelled nodes (unfair mode)

even a small-scale multiprocessor. Alternatives include descheduling a thread until it is signaled, or yielding the processor within a spin loop [KLM91]. In practice, we mainly choose the spin-then-yield approach, using the `park` and `unpark` methods contained in *java.util.concurrent.locks.LockSupport* [Lea05] to remove threads from and restore threads to the ready list. On multiprocessors (only), nodes next in line for fulfillment spin briefly (about one-quarter the time of a typical context-switch) before parking. On very busy synchronous queues, spinning can dramatically improve throughput because it handles the case of a near-simultaneous "fly-by" between a producer and consumer without stalling either. On less busy queues, the amount of spinning is small enough not to be noticeable.

Finally, the simplest approach to supporting timeout involves marking nodes cancelled and abandoning them for another thread to eventually unlink and reclaim. If, however, items are offered at a very high rate, but with a very low timeout patience, this "abandonment" cleaning strategy can result in a long-term build-up of cancelled nodes, exhausting memory supplies and degrading performance. It is important to effect a more sophisticated cleaning strategy.

In our implementation, we perform cleaning differently in stacks (unfair mode) and queues. Listing 3.12 displays the cleaning strategy for stacks; the parameter `s` is a cancelled node that needs to be unlinked. This implementation potentially requires an $O(N)$ traversal to unlink the node at the bottom of the stack; however, it can run concurrently with other threads' stack access.

We work our way from the top of the stack to the first node we see past `s`, cleaning cancelled nodes as we go. Cleanup occurs in two main phases. First, we remove cancelled nodes from the top of the stack (06–08), then we remove internal nodes (10–15). We note that our technique for removing internal nodes from the list is dependent on garbage collection: A node C unlinked from the list by being bypassed can be relinked to the list if its predecessor B becomes cancelled and another thread concurrently links B's predecessor A to C. Resolving this race condition requires complicated protocols [ScS01]. In a garbage-collected language, by contrast, unlinked nodes remain unreclaimed while references are extant.

In contrast with our cleaning strategy for stacks, for queues we usually remove a node in $O(1)$ time when it is cancelled. However, at any given time, the last node inserted in the list cannot be deleted (because there is no obvious way to update the tail pointer backwards). To accommodate the case where a node is "pinned" at the tail of the queue, we save a reference to its predecessor (in a `cleanMe` field of the queue) after first unlinking any node that was previously saved. Since only one node can be pinned at any time, at least one of the node to be unlinked and the cached node can always be reclaimed.

Listing 3.13 presents our cleaning strategy for queues. In this code, `s` is a cancelled node that needs to be unlinked, and `pred` is the node known to precede it in the queue. We begin by reading the first two nodes in the queue (lines 02–03) and unlinking any cancelled nodes from the head of the queue (04–07). Then, we check to see if the queue is empty (08–10), or has a lagging tail pointer (11–17), before checking whether `s` is the tail-most node. Assuming it is not pinned as the tail node, we unlink `s` unless

```
00 void clean(Node pred, Node s) {
01     while (pred.next == s) {
02         Node h = head;
03         Node hn = h.next;
04         if (hn != null && hn.isCancelled()) {
05             advanceHead(h, hn);
06             continue;
07         }
08         Node t = tail;
09         if (t == h)
10             return;
11         Node tn = t.next;
12         if (t != tail)
13             continue;
14         if (tn != null) {
15             advanceTail(t, tn);
16             continue;
17         }
18         if (s != t) {
19             Node sn = s.next;
20             if (sn == s || pred.casNext(s, sn))
21                 return;
22         }
23         Node dp = cleanMe;
24         if (dp != null) {
25             Node d = dp.next;
26             Node dn;
27             if (d == null || d == dp ||
28                 !d.isCancelled() ||
29                 (d != t && (dn = d.next) != null &&
30                  dn != d && dp.casNext(d, dn)))
31                 casCleanMe(dp, null);
32             if (dp == pred)
33                 return;
34         } else if (casCleanMe(null, pred))
35             return;
36     }
37 }
```

Listing 3.13: Synchronous dual queue: Cleaning cancelled nodes (fair mode)

another thread has already done so (18–22). The check in line 20 (`sn == s`) queries whether a cancelled node has been removed from the list: We link a cancelled node's next pointer back to itself to flag this state.

If `s` *is* pinned, we read the currently saved node `cleanMe` (23). If no node was saved, we save a reference to `s` predecessor and are done (34–35). Otherwise, we must first remove `cleanMe`'s cancelled successor (25). If that successor is gone (27, `d == null`) or no longer in the list (27, `d == dp`; recall that nodes unlinked from the list have their next pointers aimed back at themselves), or uncancelled (28), we simply clear out the `cleanMe` field. Else, if the successor is not currently tail-most (29), and is still in the list (30, `dn != d`), we remove it (30, `casNext` call).

3.5.2 Experimental Results

Benchmarks

We present results for several microbenchmarks and one "real world" scenario. The microbenchmarks employ threads that produce and consume as fast as they can; this represents the limiting case of producer–consumer applications as the cost to process elements approaches zero. We consider producer-consumer ratios of $1 : N$, $N : 1$, and $N : N$. Separately, we stress the timeout code by dynamically adjusting patience between the longest that fails and the shortest that succeeds.

Our "real world" scenario instantiates synchronous queues as the core of the Java SE 5.0 class *java.util.concurrent.ThreadPoolExecutor*, which in turn forms the backbone of many Java-based server applications. Our benchmark produces *tasks* to be run by a pool of worker threads managed by the *ThreadPoolExecutor*.

Methodology

We obtained results on an AMD V40z with 4 2.4GHz Opteron processors and on our SunFire 6800, using Sun's Java SE 5.0 HotSpot VM and from 2 to 64 threads. We tested each benchmark with both the fair and unfair (stack-based) versions of the Java SE 5.0 *java.util.concurrent.SynchronousQueue* and our new nonblocking algorithms. For tests that do not require timeout, we additionally test with Hanson's synchronous queue. Our new algorithms are labeled "NewSynchQueue6" in the graphs.

Figure 3.5 displays the rate at which data is transferred from multiple producers to multiple consumers; Figure 3.6 displays the rate at which data is transfered from a single producer to multiple consumers; Figure 3.7 displays the rate at which a single consumer receives data from multiple producers. Figure 3.8 displays the handoff attempt

```
import java.util.concurrent.locks.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.*;
import java.util.*;

public class NewSynchQueue<E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    private static final long serialVersionUID = -3223113410248163686L;

    /**
     * Shared internal API for dual stacks and queues.
     */
    static abstract class Transferer {
        abstract Object transfer(Object e, boolean timed, long nanos);
    }

    static final int NCPUS = Runtime.getRuntime().availableProcessors();
    static final int maxTimedSpins = (NCPUS < 2)? 0 : 32;
    static final int maxUntimedSpins = maxTimedSpins * 16;
    static final long spinForTimeoutThreshold = 1000L;

    /** Dual stack */
    static final class TransferStack extends Transferer {

        /* Modes for SNodes, ORed together in node fields */
        /** Node represents an unfulfilled consumer */
        static final int REQUEST    = 0;
        /** Node represents an unfulfilled producer */
        static final int DATA       = 1;
        /** Node is fulfilling another unfulfilled DATA or REQUEST */
        static final int FULFILLING = 2;

        /** Return true if m has fulfilling bit set */
        static boolean isFulfilling(int m) { return (m & FULFILLING) != 0; }

        /** Node class for TransferStacks. */
        static final class SNode {
            volatile SNode next;        // next node in stack
            volatile SNode match;       // the node matched to this
            volatile Thread waiter;     // to control park/unpark
            Object item;                // data; or null for REQUESTs
            int mode;
            // Note: item and mode fields don’t need to be volatile
            // since they are always written before, and read after,
            // other volatile/atomic operations.

            SNode(Object item) {
                this.item = item;
            }

            static final AtomicReferenceFieldUpdater<SNode, SNode>
                nextUpdater = AtomicReferenceFieldUpdater.newUpdater
                (SNode.class, SNode.class, "next");

            boolean casNext(SNode cmp, SNode val) {
                return (cmp == next &&
                        nextUpdater.compareAndSet(this, cmp, val));
            }
```

Listing 3.14: The full synchronous queue

```
        static final AtomicReferenceFieldUpdater<SNode, SNode>
            matchUpdater = AtomicReferenceFieldUpdater.newUpdater
            (SNode.class, SNode.class, "match");

        /**
         * Tries to match node s to this node, if so, waking up thread.
         * Fulfillers call tryMatch to identify their waiters.
         * Waiters block until they have been matched.
         *
         * @param s the node to match
         * @return true if successfully matched to s
         */
        boolean tryMatch(SNode s) {
            if (match == null &&
                matchUpdater.compareAndSet(this, null, s)) {
                Thread w = waiter;
                if (w != null) {    // waiters need at most one unpark
                    waiter = null;
                    LockSupport.unpark(w);
                }
                return true;
            }
            return match == s;
        }

        /**
         * Tries to cancel a wait by matching node to itself.
         */
        void tryCancel() {
            matchUpdater.compareAndSet(this, null, this);
        }

        boolean isCancelled() {
            return match == this;
        }
    }

    /** The head (top) of the stack */
    volatile SNode head;

    static final AtomicReferenceFieldUpdater<TransferStack, SNode>
        headUpdater = AtomicReferenceFieldUpdater.newUpdater
        (TransferStack.class,  SNode.class, "head");

    boolean casHead(SNode h, SNode nh) {
        return h == head && headUpdater.compareAndSet(this, h, nh);
    }

    /**
     * Creates or resets fields of a node. Called only from transfer
     * where the node to push on stack is lazily created and
     * reused when possible to help reduce intervals between reads
     * and CASes of head and to avoid surges of garbage when CASes
     * to push nodes fail due to contention.
     */
    static SNode snode(SNode s, Object e, SNode next, int mode) {
        if (s == null) s = new SNode(e);
        s.mode = mode;
        s.next = next;
        return s;
    }
```

Listing 3.14: (continued)

```
/**
 * Puts or takes an item.
 */
Object transfer(Object e, boolean timed, long nanos) {

    SNode s = null; // constructed/reused as needed
    int mode = (e == null)? REQUEST : DATA;

    for (;;) {
        SNode h = head;
        if (h == null || h.mode == mode) {  // empty or same-mode
            if (timed && nanos <= 0) {      // can’t wait
                if (h != null && h.isCancelled())
                    casHead(h, h.next);     // pop cancelled node
                else
                    return null;
            } else if (casHead(h, s = snode(s, e, h, mode))) {
                SNode m = awaitFulfill(s, timed, nanos);
                if (m == s) {               // wait was cancelled
                    clean(s);
                    return null;
                }
                if ((h = head) != null && h.next == s)
                    casHead(h, s.next);     // help s’s fulfiller
                return mode == REQUEST? m.item : s.item;
            }
        } else if (!isFulfilling(h.mode)) { // try to fulfill
            if (h.isCancelled())            // already cancelled
                casHead(h, h.next);         // pop and retry
            else if (casHead(h, s=snode(s, e, h, FULFILLING|mode))) {
                for (;;) { // loop until matched or waiters disappear
                    SNode m = s.next;       // m is s’s match
                    if (m == null) {        // all waiters are gone
                        casHead(s, null);   // pop fulfill node
                        s = null;           // use new node next time
                        break;              // restart main loop
                    }
                    SNode mn = m.next;
                    if (m.tryMatch(s)) {
                        casHead(s, mn);     // pop both s and m
                        return (mode == REQUEST)? m.item : s.item;
                    } else                  // lost match
                        s.casNext(m, mn);   // help unlink
                }
            }
        } else {                            // help a fulfiller
            SNode m = h.next;               // m is h’s match
            if (m == null)                  // waiter is gone
                casHead(h, null);           // pop fulfilling node
            else {
                SNode mn = m.next;
                if (m.tryMatch(h))          // help match
                    casHead(h, mn);         // pop both h and m
                else                        // lost match
                    h.casNext(m, mn);       // help unlink
            }
        }
    }
}
```

Listing 3.14: (continued)

```
        /**
         * Spins/blocks until node s is matched by a fulfill operation.
         *
         * @param s the waiting node
         * @param timed true if timed wait
         * @param nanos timeout value
         * @return matched node, or s if cancelled
         */
        SNode awaitFulfill(SNode s, boolean timed, long nanos) {
            /*
             * When a node/thread is about to block, it sets its waiter
             * field and then rechecks state at least one more time
             * before actually parking, thus covering race vs
             * fulfiller noticing that waiter is non-null so should be
             * woken.
             *
             * When invoked by nodes that appear at the point of call
             * to be at the head of the stack, calls to park are
             * preceded by spins to avoid blocking when producers and
             * consumers are arriving very close in time.  This can
             * happen enough to bother only on multiprocessors.
             *
             * The order of checks for returning out of main loop
             * reflects fact that interrupts have precedence over
             * normal returns, which have precedence over
             * timeouts. (So, on timeout, one last check for match is
             * done before giving up.) Except that calls from untimed
             * SynchronousQueue.{poll/offer} don't check interrupts
             * and don't wait at all, so are trapped in transfer
             * method rather than calling awaitFulfill.
             */
            long lastTime = (timed)? System.nanoTime() : 0;
            Thread w = Thread.currentThread();
            SNode h = head;
            int spins = (shouldSpin(s)?
                         (timed? maxTimedSpins : maxUntimedSpins) : 0);
            for (;;) {
                if (w.isInterrupted())
                    s.tryCancel();
                SNode m = s.match;
                if (m != null)
                    return m;
                if (timed) {
                    long now = System.nanoTime();
                    nanos -= now - lastTime;
                    lastTime = now;
                    if (nanos <= 0) {
                        s.tryCancel();
                        continue;
                    }
                }
                if (spins > 0)
                    spins = shouldSpin(s)? (spins-1) : 0;
                else if (s.waiter == null)
                    s.waiter = w; // establish waiter so can park next iter
                else if (!timed)
                    LockSupport.park();
                else if (nanos > spinForTimeoutThreshold)
                    LockSupport.parkNanos(nanos);
            }
        }
```

Listing 3.14: (continued)

```
        /**
         * Returns true if node s is at head or there is an active
         * fulfiller.
         */
        boolean shouldSpin(SNode s) {
            SNode h = head;
            return (h == s || h == null || isFulfilling(h.mode));
        }

        /**
         * Unlinks s from the stack.
         */
        void clean(SNode s) {
            s.item = null;   // forget item
            s.waiter = null; // forget thread

            SNode past = s.next;
            if (past != null && past.isCancelled())
                past = past.next;

            // Absorb cancelled nodes at head
            SNode p;
            while ((p = head) != null && p != past && p.isCancelled())
                casHead(p, p.next);

            // Unsplice embedded nodes
            while (p != null && p != past) {
                SNode n = p.next;
                if (n != null && n.isCancelled())
                    p.casNext(n, n.next);
                else
                    p = n;
            }
        }
    }

    /** Dual Queue */
    static final class TransferQueue extends Transferer {
        /** Node class for TransferQueue. */
        static final class QNode {
            volatile QNode next;          // next node in queue
            volatile Object item;         // CAS'ed to or from null
            volatile Thread waiter;       // to control park/unpark
            final boolean isData;

            QNode(Object item, boolean isData) {
                this.item = item;
                this.isData = isData;
            }

            static final AtomicReferenceFieldUpdater<QNode, QNode>
                nextUpdater = AtomicReferenceFieldUpdater.newUpdater
                (QNode.class, QNode.class, "next");

            boolean casNext(QNode cmp, QNode val) {
                return (next == cmp &&
                        nextUpdater.compareAndSet(this, cmp, val));
            }

            static final AtomicReferenceFieldUpdater<QNode, Object>
                itemUpdater = AtomicReferenceFieldUpdater.newUpdater
                (QNode.class, Object.class, "item");
```

Listing 3.14: (continued)

```
        boolean casItem(Object cmp, Object val) {
            return (item == cmp &&
                    itemUpdater.compareAndSet(this, cmp, val));
        }

        /**
         * Tries to cancel by CAS'ing ref to this as item.
         */
        void tryCancel(Object cmp) {
            itemUpdater.compareAndSet(this, cmp, this);
        }

        boolean isCancelled() {
            return item == this;
        }

        /**
         * Returns true if this node is known to be off the queue
         * because its next pointer has been forgotten due to
         * an advanceHead operation.
         */
        boolean isOffList() {
            return next == this;
        }
    }

    /** Head of queue */
    transient volatile QNode head;
    /** Tail of queue */
    transient volatile QNode tail;
    /**
     * Reference to a cancelled node that might not yet have been
     * unlinked from queue because it was the last inserted node
     * when it cancelled.
     */
    transient volatile QNode cleanMe;

    TransferQueue() {
        QNode h = new QNode(null, false); // initialize to dummy node.
        head = h;
        tail = h;
    }

    static final AtomicReferenceFieldUpdater<TransferQueue, QNode>
        headUpdater = AtomicReferenceFieldUpdater.newUpdater
        (TransferQueue.class,  QNode.class, "head");

    /**
     * Tries to cas nh as new head; if successful, unlink
     * old head's next node to avoid garbage retention.
     */
    void advanceHead(QNode h, QNode nh) {
        if (h == head && headUpdater.compareAndSet(this, h, nh))
            h.next = h; // forget old next
    }

    static final AtomicReferenceFieldUpdater<TransferQueue, QNode>
        tailUpdater = AtomicReferenceFieldUpdater.newUpdater
        (TransferQueue.class, QNode.class, "tail");
```

Listing 3.14: (continued)

```
/**
 * Tries to cas nt as new tail.
 */
void advanceTail(QNode t, QNode nt) {
    if (tail == t)
        tailUpdater.compareAndSet(this, t, nt);
}

static final AtomicReferenceFieldUpdater<TransferQueue, QNode>
    cleanMeUpdater = AtomicReferenceFieldUpdater.newUpdater
    (TransferQueue.class, QNode.class, "cleanMe");

/**
 * Tries to CAS cleanMe slot.
 */
boolean casCleanMe(QNode cmp, QNode val) {
    return (cleanMe == cmp &&
            cleanMeUpdater.compareAndSet(this, cmp, val));
}

/**
 * Puts or takes an item.
 */
Object transfer(Object e, boolean timed, long nanos) {
    /* Basic algorithm is to loop trying to take either of
     * two actions:
     *
     * 1. If queue apparently empty or holding same-mode nodes,
     *    try to add node to queue of waiters, wait to be
     *    fulfilled (or cancelled) and return matching item.
     *
     * 2. If queue apparently contains waiting items, and this
     *    call is of complementary mode, try to fulfill by CAS'ing
     *    item field of waiting node and dequeuing it, and then
     *    returning matching item.
     *
     * In each case, along the way, check for and try to help
     * advance head and tail on behalf of other stalled/slow
     * threads.
     *
     * The loop starts off with a null check guarding against
     * seeing uninitialized head or tail values. This never
     * happens in current SynchronousQueue, but could if
     * callers held non-volatile/final ref to the
     * transferer. The check is here anyway because it places
     * null checks at top of loop, which is usually faster
     * than having them implicitly interspersed.
     */

    QNode s = null; // constructed/reused as needed
    boolean isData = (e != null);

    for (;;) {
        QNode t = tail;
        QNode h = head;
        if (t == null || h == null)         // saw uninitialized value
            continue;                       // spin
```

Listing 3.14: (continued)

```
            if (h == t || t.isData == isData) { // empty or same-mode
                QNode tn = t.next;
                if (t != tail)                  // inconsistent read
                    continue;
                if (tn != null) {               // lagging tail
                    advanceTail(t, tn);
                    continue;
                }
                if (timed && nanos <= 0)        // can't wait
                    return null;
                if (s == null)
                    s = new QNode(e, isData);
                if (!t.casNext(null, s))        // failed to link in
                    continue;

                advanceTail(t, s);              // swing tail and wait
                Object x = awaitFulfill(s, e, timed, nanos);
                if (x == s) {                   // wait was cancelled
                    clean(t, s);
                    return null;
                }

                if (!s.isOffList()) {           // not already unlinked
                    advanceHead(t, s);          // unlink if head
                    if (x != null)              // and forget fields
                        s.item = s;
                    s.waiter = null;
                }
                return (x != null)? x : e;

            } else {                            // complementary-mode
                QNode m = h.next;               // node to fulfill
                if (t != tail || m == null || h != head)
                    continue;                   // inconsistent read

                Object x = m.item;
                if (isData == (x != null) ||    // m already fulfilled
                    x == m ||                   // m cancelled
                    !m.casItem(x, e)) {         // lost CAS
                    advanceHead(h, m);          // dequeue and retry
                    continue;
                }

                advanceHead(h, m);              // successfully fulfilled
                LockSupport.unpark(m.waiter);
                return (x != null)? x : e;
            }
        }
    }
```

Listing 3.14: (continued)

```
        /**
         * Spins/blocks until node s is fulfilled.
         *
         * @param s the waiting node
         * @param e the comparison value for checking match
         * @param timed true if timed wait
         * @param nanos timeout value
         * @return matched item, or s if cancelled
         */
        Object awaitFulfill(QNode s, Object e, boolean timed, long nanos) {
            /* Same idea as TransferStack.awaitFulfill */
            long lastTime = (timed)? System.nanoTime() : 0;
            Thread w = Thread.currentThread();
            int spins = ((head.next == s) ?
                         (timed? maxTimedSpins : maxUntimedSpins) : 0);
            for (;;) {
                if (w.isInterrupted())
                    s.tryCancel(e);
                Object x = s.item;
                if (x != e)
                    return x;
                if (timed) {
                    long now = System.nanoTime();
                    nanos -= now - lastTime;
                    lastTime = now;
                    if (nanos <= 0) {
                        s.tryCancel(e);
                        continue;
                    }
                }
                if (spins > 0)
                    --spins;
                else if (s.waiter == null)
                    s.waiter = w;
                else if (!timed)
                    LockSupport.park();
                else if (nanos > spinForTimeoutThreshold)
                    LockSupport.parkNanos(nanos);
            }
        }
```

Listing 3.14: (continued)

rate, given very limited patience for producers and consumers. Figure 3.9 presents execution time per task for our *ThreadPoolExecutor* benchmark.

Discussion

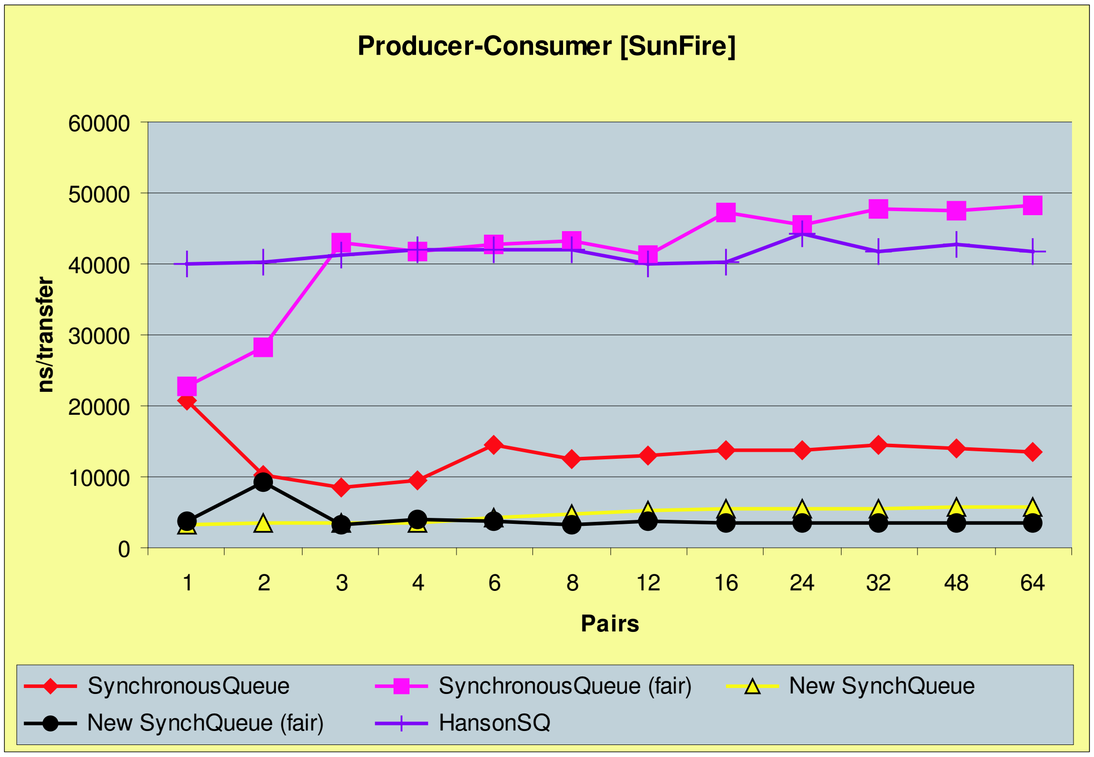
As can be seen from Figure 3.5, Hanson's synchronous queue and the Java SE 5.0 fair-mode synchronous queue both perform relatively poorly, taking 4 (Opteron) to 8 (SunFire) times as long to effect a transfer relative to the faster algorithms. The unfair (stack-based) Java SE 5.0 synchronous queue in turn incurs twice the overhead of either

```
        /**
         * Gets rid of cancelled node s with original predecessor pred.
         */
        void clean(QNode pred, QNode s) {
            s.waiter = null; // forget thread
            /*
             * At any given time, exactly one node on list cannot be
             * deleted -- the last inserted node. To accommodate this,
             * if we cannot delete s, we save its predecessor as
             * "cleanMe", deleting the previously saved version
             * first. At least one of node s or the node previously
             * saved can always be deleted, so this always terminates.
             */
            while (pred.next == s) { // Return early if already unlinked
                QNode h = head;
                QNode hn = h.next;   // Absorb cancelled first node as head
                if (hn != null && hn.isCancelled()) {
                    advanceHead(h, hn);
                    continue;
                }
                QNode t = tail;      // Ensure consistent read for tail
                if (t == h)
                    return;
                QNode tn = t.next;
                if (t != tail)
                    continue;
                if (tn != null) {
                    advanceTail(t, tn);
                    continue;
                }
                if (s != t) {        // If not tail, try to unsplice
                    QNode sn = s.next;
                    if (sn == s || pred.casNext(s, sn))
                        return;
                }
                QNode dp = cleanMe;
                if (dp != null) {    // Try unlinking previous cancelled node
                    QNode d = dp.next;
                    QNode dn;
                    if (d == null ||               // d is gone or
                        d == dp ||                 // d is off list or
                        !d.isCancelled() ||        // d not cancelled or
                        (d != t &&                 // d not tail and
                         (dn = d.next) != null &&  //   has successor
                         dn != d &&                //   that is on list
                         dp.casNext(d, dn)))       // d unspliced
                        casCleanMe(dp, null);
                    if (dp == pred)
                        return;      // s is already saved node
                } else if (casCleanMe(null, pred))
                    return;          // Postpone cleaning s
            }
        }
    }
```

Listing 3.14: (continued)

```
    private transient volatile Transferer transferer;
    public NewSynchQueue() {
        this(false);
    }
    public NewSynchQueue(boolean fair) {
        transferer = (fair)? new TransferQueue() : new TransferStack();
    }

    public void put(E o) throws InterruptedException {
        if (o == null) throw new NullPointerException();
        if (transferer.transfer(o, false, 0) == null)
            throw new InterruptedException();
    }

    public boolean offer(E o, long timeout, TimeUnit unit)
        throws InterruptedException {
        if (o == null) throw new NullPointerException();
        if (transferer.transfer(o, true, unit.toNanos(timeout)) != null)
            return true;
        if (!Thread.interrupted())
            return false;
        throw new InterruptedException();
    }

    public boolean offer(E e) {
        if (e == null) throw new NullPointerException();
        return transferer.transfer(e, true, 0) != null;
    }

    public E take() throws InterruptedException {
        Object e = transferer.transfer(null, false, 0);
        if (e != null)
            return (E)e;
        throw new InterruptedException();
    }

    public E poll(long timeout, TimeUnit unit) throws InterruptedException {
        Object e = transferer.transfer(null, true, unit.toNanos(timeout));
        if (e != null || !Thread.interrupted())
            return (E)e;
        throw new InterruptedException();
    }

    public E poll() {
        return (E)transferer.transfer(null, true, 0);
    }

    // Remaining methods are stubbed BlockingQueue interface calls
    // and bits needed to cope with the Java SE 5.0 serialization
    // strategy. As these are not germane to the algorithm itself,
    // we omit them from this listing.
}
```

Listing 3.14: (continued)



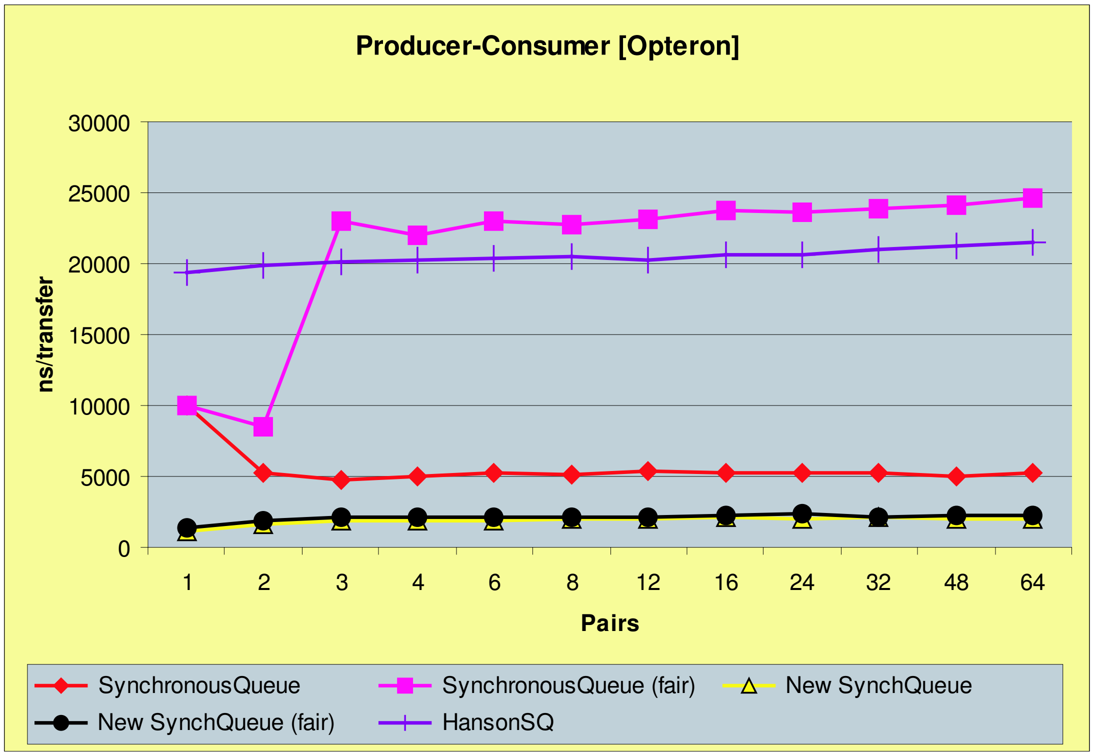
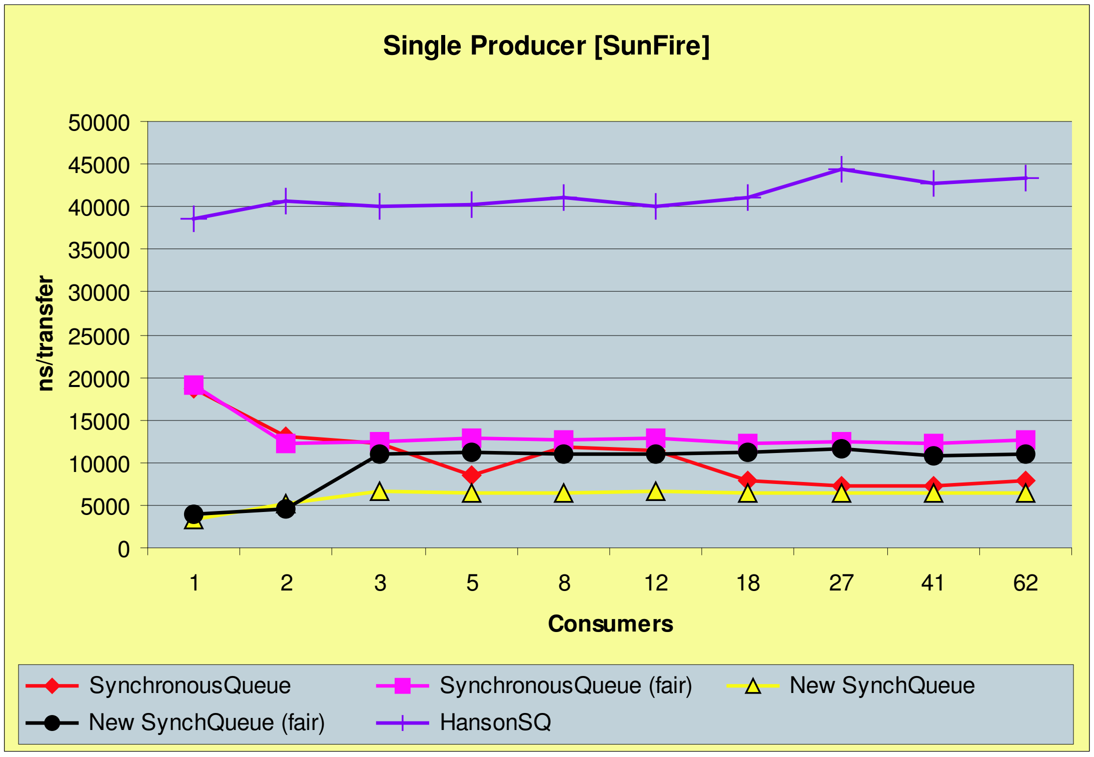


Figure 3.5: Synchronous handoff: $N : N$ producers : consumers



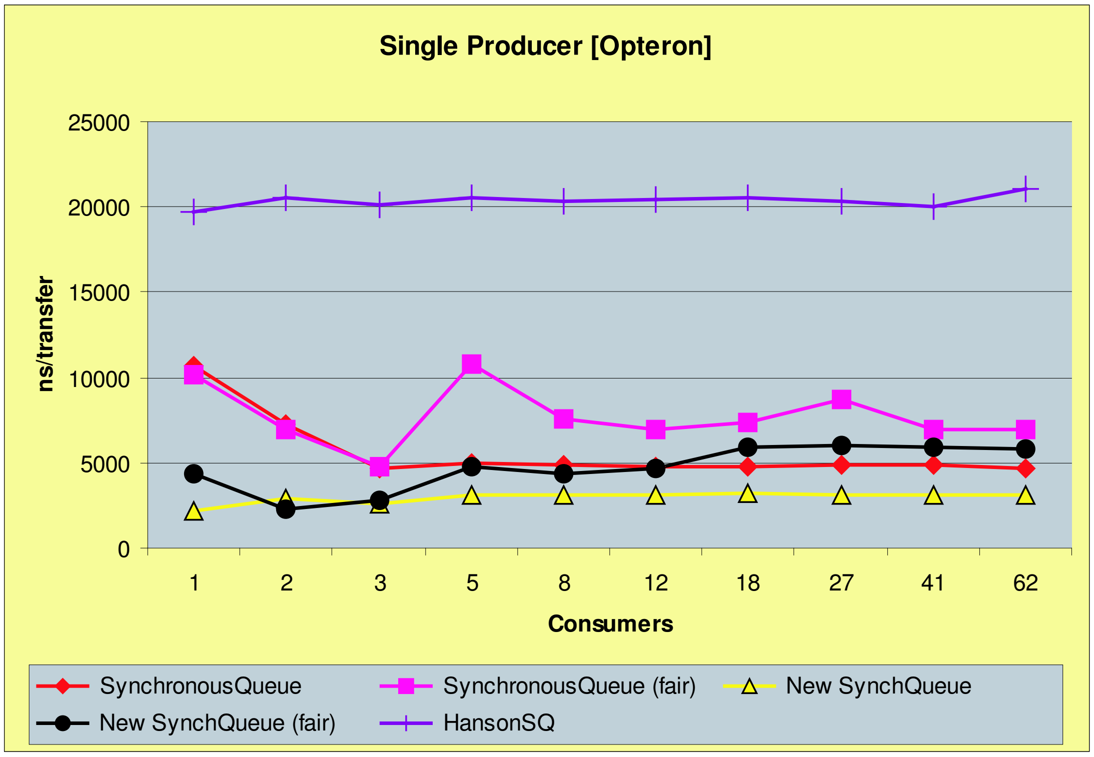
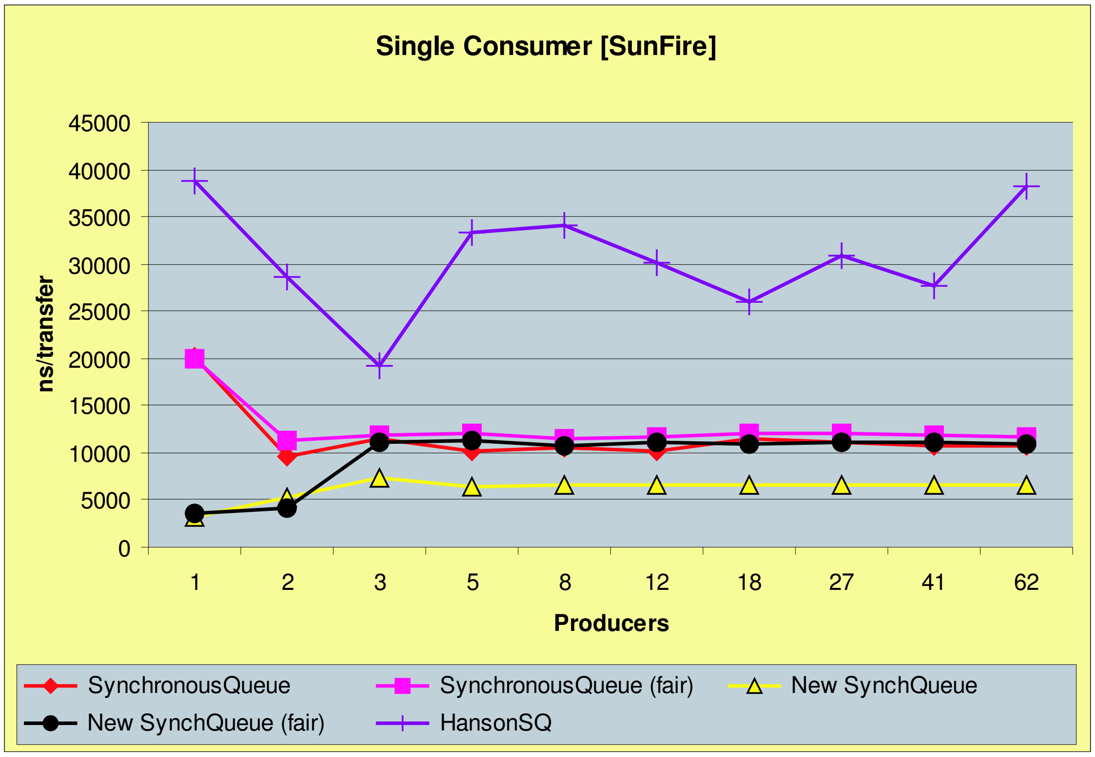


Figure 3.6: Synchronous handoff: $1 : N$ producers : consumers



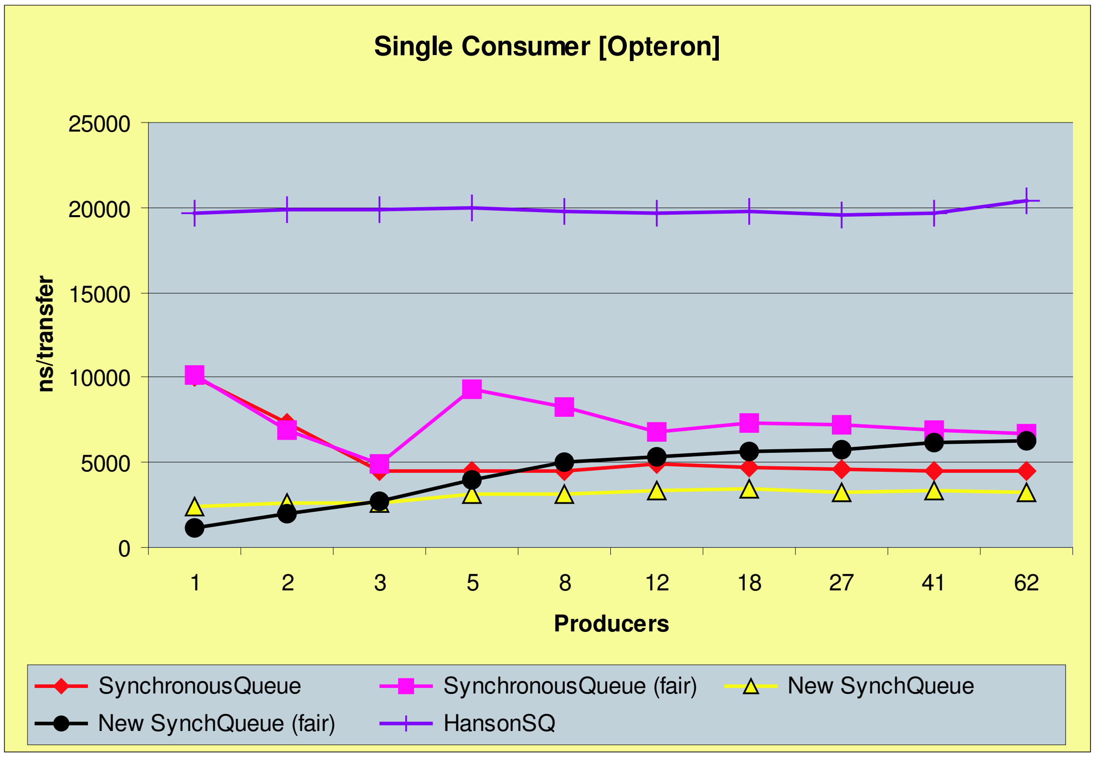
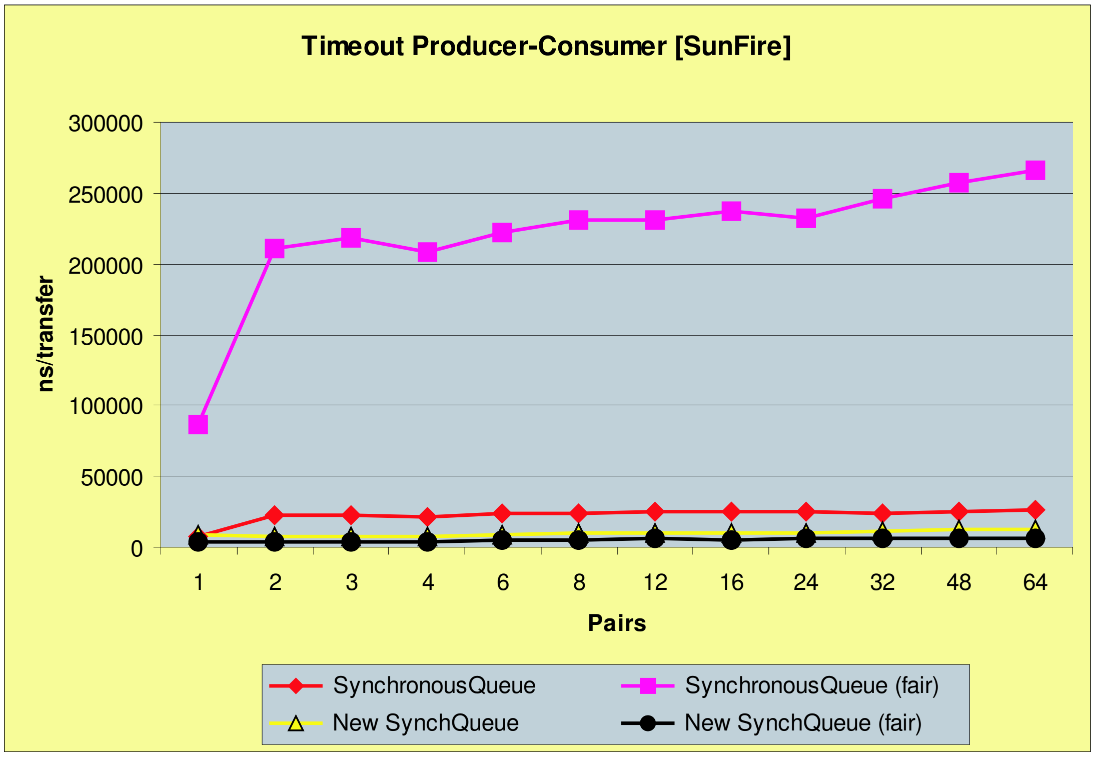


Figure 3.7: Synchronous handoff: N : 1 producers : consumers



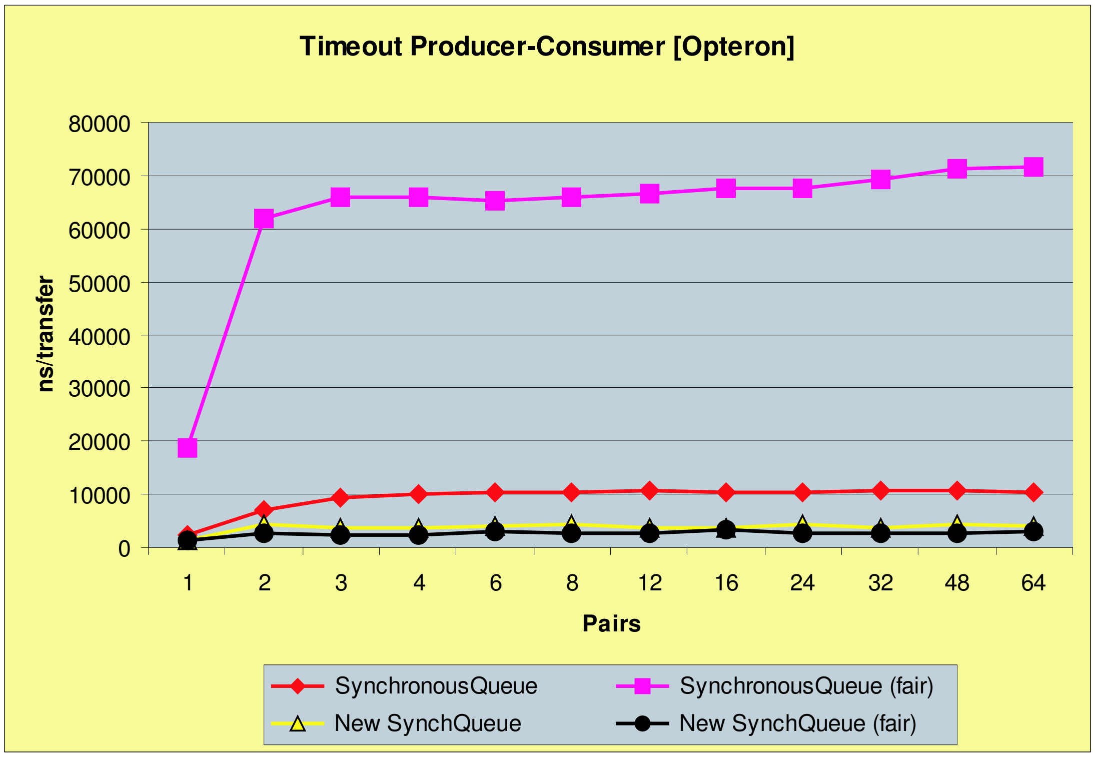
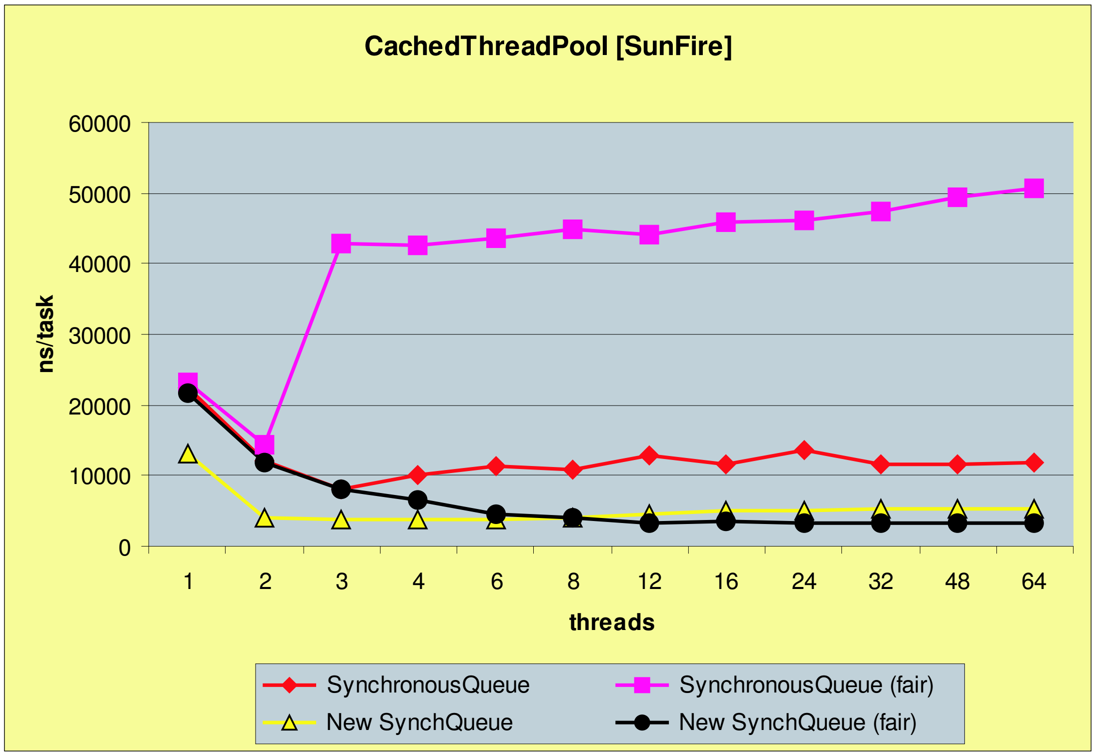


Figure 3.8: Synchronous handoff: Low patience transfers



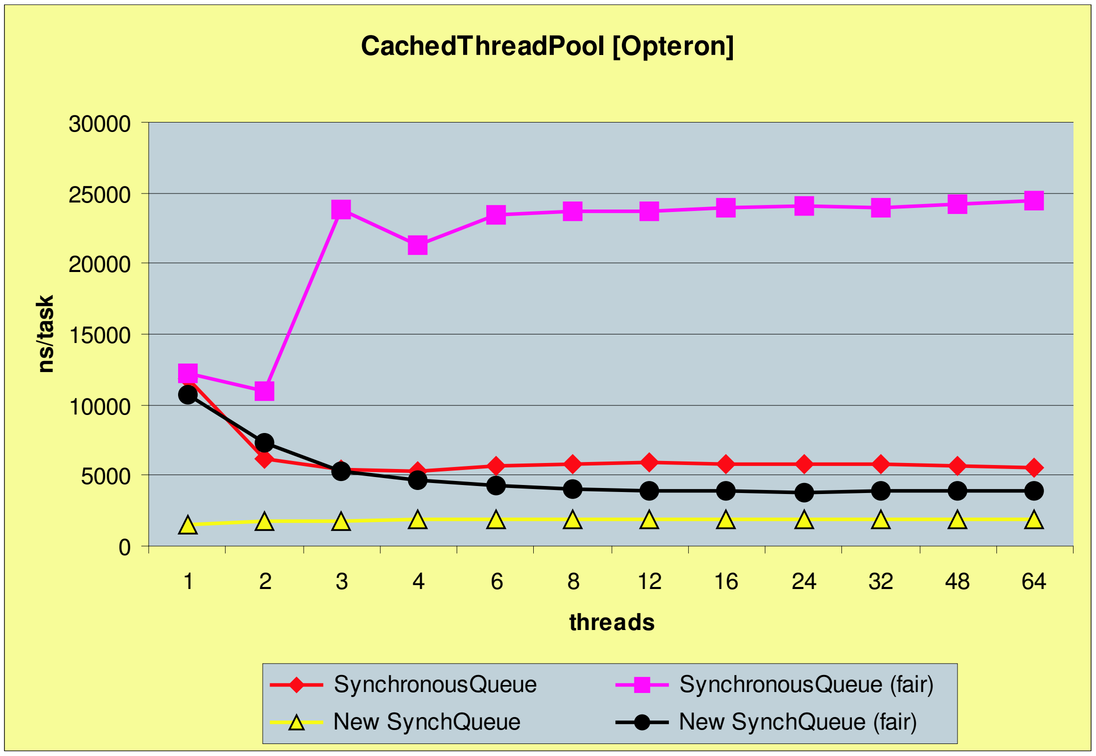


Figure 3.9: Synchronous queue: ThreadPoolExecutor benchmark

the fair or unfair version of our new algorithm, both versions of which are comparable in performance. The main reason that the Java SE 5.0 fair-mode queue is so much slower than unfair is that the fair-mode version uses a fair-mode entry lock to ensure FIFO wait ordering. This causes pile-ups that block the threads that will fulfill waiting threads. This difference supports our claim that blocking and contention surrounding the synchronization state of synchronous queues are major impediments to scalability.

When a single producer struggles to satisfy multiple consumers (Figure 3.6), or a single consumer struggles to receive data from multiple producers (Figure 3.7), the disadvantages of Hanson's synchronous queue are accentuated. Because the singleton necessarily blocks for every operation, the time it takes to produce or consume data increases noticeably. Our new synchronous queue consistently outperforms the Java SE 5.0 implementation (fair vs. fair and unfair vs. unfair) at all levels of concurrency.

The dynamic timeout test (Figure 3.8) reveals another deficiency in the Java SE 5.0 *SynchronousQueue* implementation: Fair mode has a pathologically bad response to timeout, due mainly to the cost of time-out in its fair-mode reentrant locks. Meanwhile, either version of our new algorithm outperforms the unfair-mode Java SE 5.0 *SynchronousQueue* by a factor of five.

Finally, in Figure 3.9, we see that the performance differentials we observe in *java.util.concurrent.SynchronousQueue* translate directly into overhead in the *java.util. concurrent.ThreadPoolExecutor*: Our new fair version outperforms the Java SE 5.0 implementation by factors of 14 (SunFire) and 6 (Opteron); our unfair version outperforms Java SE 5.0 by a factor of three on both platforms. Interestingly, the relative performance of fair and unfair versions of our new algorithm differs between the SunFire and Opteron platforms. Generally, unfair mode tends to improve locality by keeping some threads "hot" and others buried at the bottom of the stack. Conversely, however, it tends to increase the number of times threads are scheduled and descheduled. On the SunFire platform, context switches have a higher relative overhead compared to other factors; this is why our fair synchronous queue eventually catches and surpasses the unfair version's performance. By way of comparison, the cost of context switches is relatively smaller on the Opteron platform, so the trade-off tips in favor of increased locality and the unfair version performs best.

Across all benchmarks, our fair synchronous queue universally outperforms all other fair synchronous queues and our unfair synchronous queue outperforms all other unfair synchronous queues, regardless of preemption or the level of concurrency.

3.6 Conclusions

Linearizability is central to the study of concurrent data structures. It has historically been limited by its restriction to methods that are total. We have shown how to encompass partial methods by separating them into first-class request and followup operations. By so doing, we obtain meaningful definitions of wait-free, lock-free, and obstruction-free implementations of concurrent objects with condition synchronization.

We have presented concrete lock-free implementations of a *dual stack* and a *dual queue*. Performance results on a commercial multiprocessor suggest that dualism can yield significant performance gains over naive retry on failure. The dual queue, in particular, appears to be an eminently useful algorithm, outperforming the M&S queue (with retry on failure) in our experiments by almost a factor of two for large thread counts.

We have further presented a novel lock-free exchange channel that achieves very high scalability through the use of elimination. In a head-to-head microbenchmark comparison, our algorithm outperforms the Java SE 5.0 *Exchanger* by a factor of two at two threads and a factor of 50 at 10 threads. We have further shown that this performance differential spells the difference between performance degradation and linear parallel speedup in a genetic algorithm-based traveling salesman application. Our exchange channel has been adopted for inclusion in Java 6.

Finally, we have presented two new lock-free synchronous queue implementations that outperform all previously known algorithms by a wide margin. In striking contrast to previous implementations, there is little performance cost for fairness. Again, in head-to-head comparisons, our algorithms consistently outperform the Java SE 5.0 *SynchronousQueue* (formerly the fastest known implementation of a synchronous queue) by a factor of three in unfair mode and up to a factor of 14 in fair mode. We have further shown that this performance differential translates directly to factors of two and ten when substituting our new synchronous queue in for the core of the Java SE 5.0 *ThreadPoolExecutor*, which is itself at the heart of many Java-based server implementations. Our new synchronous queues have also been adopted for inclusion in Java 6.

That our new lock-free synchronous queue implementations are more scalable than the Java SE 5.0 *SynchronousQueue* class is unsurprising: Nonblocking algorithms often scale far better than corresponding lock-based algorithms. More surprisingly, our new implementations are faster even at low levels of contention; nonblocking algorithms often exhibit greater base-case overhead than do lock-based equivalents. We unreservedly recommend our new algorithms anywhere synchronous handoff is used.

3.7 Future work

Our relaxation of linearizability and nonblocking progress conditions has led directly to algorithms that improve, sometimes dramatically, upon the performance of previously known implementations. Nonetheless, we believe that work in this area is just beginning. In particular, nonblocking dual data structures could undoubtedly be developed for double-ended queues, priority queues, sets, dictionaries, and other abstractions. Each of these may in turn have variants that embody different policies as to which of several pending requests to fulfill when a matching operation makes a precondition true. One could imagine, for example, a stack that grants pending requests in FIFO order, or (conceivably) a queue that grants them in LIFO order. More plausibly, one could imagine an arbitrary system of thread priorities, in which a matching operation fulfills the highest priority pending request.

Another candidate for future work is extending the use of elimination – as in the exchanger – with other algorithms. In particular, incorporating elimination with our synchronous queues could further improve their performance and scalability.

4 Software Transactional Memory

4.1 Introduction

In the introduction to Chapter 3, we mentioned that nonblocking algorithms are notoriously difficult to design and implement. Realistically, only experts in the field are able to produce them. It is telling, perhaps, that in the last two years at least three papers have been published on data structures as fundamental as queues [ScS04b; LaS04; DDG04]. What then is the rank-and-file programmer to do?

An alternative to creating ad hoc implementations of each data structure is to use a general purpose *universal construction* that allows them to be created mechanically. The term *software transactional memory* (STM) was coined by Shavit and Touitou [ShT97] as a software-only implementation of a hardware-based scheme proposed by Herlihy and Moss [HeM93]. Although early STM systems were primarily academic curiosities, more modern systems [Fra04; HaF03; HLM03b] have reduced runtime overheads sufficiently to outperform coarse-grained locks when several threads are active.

STM-based algorithms can generally be expected to be slower than either ad hoc nonblocking algorithms or fine-grained lock-based code. At the same time, they are as easy to use as coarse-grain locks: One simply brackets the code that needs to be atomic.

In fact, STM systems allow correct sequential code to be converted, mechanically, into highly concurrent correct nonblocking code. Further, algorithms based on nonblocking STMs avoid many problems commonly associated with locks: deadlock, priority inversion, convoying, and preemption and fault vulnerability.

The remainder of this chapter is organized as follows. Section 4.2 traces a history of universal constructions and transactional memories. Then, in Section 4.3, we present the dynamic software transactional memory (DSTM) system of Herlihy et al. [HLM03b]. A novel feature of DSTM is its use of modular *contention managers* that negotiate the interplay between competing transactions' needs to access individual objects in memory. We say that contention management policies are modular because the programmer can swap in at runtime whichever one is most desirable and effective for the current software and hardware environment. In Section 4.4, we analyze over a dozen and a half distinct policies for contention management in order to provide a preliminary characterization of how one might want to choose a policy. Finally, we conclude in Section 4.5.

4.2 Literature Review

In this section, we trace an overview of the history of *universal constructions* and transactional memory systems.

4.2.1 Early Work: The Origins of Universal Constructions

The first universal constructions are due to Herlihy [Her90; Her93]. His simple lock-free protocol consists of three steps. First, the thread uses the `load_linked` instruction to read a global pointer to the object to be updated. Next, the thread copies the entire object, validates the private copy, and applies sequential updates to it. Finally, it uses the `store_conditional` instruction to swing the global pointer to its local copy. If this succeeds, the operation is complete; otherwise, some other thread has succeeded and the thread needs to retry.

In this same paper, Herlihy also presents a wait-free extension to his lock-free algorithm. In it, an array has an entry for each thread; each entry includes an abstract representation of the type of operations to be performed on the data structure. When a thread wishes to perform an operation, it first registers its request by filling its array entry. It then walks through the entire array, attempting active operations for *all* threads. This approach, which Herlihy terms *operation combining*, is an early example of what has come to be known as a *helping mechanism.*

Herlihy's constructions work well for small objects and small numbers of threads, though the lock-free construction augmented with exponential backoff between attempts outperforms the wait-free version. However, the amount of copying required renders these constructions useful only for small objects. Another problem with Herlihy's protocols is that unsuccessful update attempts waste large amounts of resources by potentially allowing up to $N - 1$ (where N is the total number of threads) attempts to fail for each one that succeeds.

One attempt to address these issues is due to Alemany and Felten [AlF92]. They attempt to conserve resources by bounding the number of threads that can concurrently attempt an update on the object to a small constant k. This is done by adding a shared counter to the object. Threads atomically increment the counter before reading the global pointer to the object; they then spin so long as the counter is greater than k. As threads complete each attempt, they decrement the counter to allow other threads to attempt their own updates. It may seem that if all k "enabled" threads are preempted or otherwise delayed, then this protocol would not guarantee lock-freedom. Indeed, Alemany and Felten require support from the operating system to decrement the counter each time an enabled thread is delayed and re-increment it each time the thread's delay ends. In exchange for this system requirement, they achieve a moderate performance improvement.

Anderson and Moir [AnM97] replace use of the shared counter with a more sophisticated k-exclusion algorithm to achieve similar gains in performance by bounding the level of contention that wait-free algorithms must handle.

Another limitation of Herlihy's construction is that all operations conflict with each other, regardless of whether they overlap in terms of which portions of the object they update. Barnes's lock-free construction [Bar93] addresses this problem with a technique he names *caching*. In his approach, the object is carved up into small cells. This technique has four steps. First, a thread completes its operation on a cached copy of each cell that it needs to modify. Caches are copied from the "live" object, then re-read to ensure consistency. Second, it verifies that the cells have not been changed from the cached version, aborting and starting over if they have. Third, it registers a claim on each updated cell, and finally each claimed cell is replaced with an updated version. Claims are registered in a strictly ascending order (based on a unique key assigned to each cell) in order to prevent livelock. If a claim is found at any time in a needed cell, threads can help each other by completing the updates needed by the thread that installed the claim. This helping mechanism is often described as "recursive" because in the thread of helping one thread, another conflict that requires further helping could be discovered. Other disadvantages of this approach include the requirement that each cell be exactly the same size and that each cell contain additional space for the claim data structure.

4.2.2 Hardware-based Transaction Support

All of the techniques described thus far exhibit noticeable performance degradation compared to simpler lock-based schemes. One attempt to rectify this is Herlihy and Moss' transactional memory [HeM93]. Theirs is a hardware scheme in which data structure operations are encapsulated into *transactions*, groups of memory updates that collectively represent the operation. A special transactional cache holds tentative updates to data as well as memory locations that have been transactionally read (but not updated). The cache coherence mechanism is then extended such that so long as the transactionally cached data are not invalidated, the transaction can continue. Once the updates are completed, a *commit* instruction releases them to other processors. One particularly interesting feature of this scheme is that data can be transactionally loaded with hints that it is intended for either read-only or full read-write access; read-only access allows operations to proceed concurrently if neither updates a particular value. While they can proceed concurrently even without hints, hints can help reduce the cost of loading and then upgrading data.

Although Herlihy and Moss are frequently cited as the first to propose transactional memory, an earlier description of an architecture that explicitly supports transactions as a mechanism for parallelizing code is due to Knight [Kni86]. His work, however, is targeted at parallelizing Lisp on Lisp Machine hardware, and is not generally applicable. In particular, it requires strict serialization on the order in which transactions commit.

An even earlier scheme is due to Stonebraker [Sto84]. This system implements transactions by providing hardware support for locking at page-level granularity. As such it is not really suitable for fine-level manipulation of data structures in the manner in which transactional memory is now perceived as being most useful. A subsequent scheme by Chang and Mergen [ChM88] shrinks the granularity from full pages to segments within a page; however, it still imposes the restriction that only one transaction at a time can access any portion of the page for write access. This scheme was implemented in a commercial processor: the IBM PC/RT.

4.2.3 Software-based Transactional Approaches

In response to the need for special hardware support required by Herlihy and Moss' scheme [HeM93], researchers began looking for software-based approaches to transactional memory. The first of these is due to Shavit and Touitou [ShT97], who coined the term *software transactional memory* (STM). Their scheme is *static* in that transactions consist of updates to predetermined locations, in some predefined order. Further, only transactional memory locations, which have additional ownership data added in, can be updated by transactions. As in Barnes's method [Bar93], transactions consist of claiming a sequence of memory locations, validating each location, then updating memory locations once validated; once all memory locations are claimed, the transaction is guaranteed to complete.

To avoid the recursive helping of Barnes's approach, Shavit and Touitou use what they call *non-redundant helping*. In this scheme, if a transaction encounters conflict while attempting to claim its list of nodes, it releases all claimed nodes before attempting to help the other thread. Importantly, it will only try to help one other thread behind which its own transaction is blocked; if it encounters contention on this secondary transaction, it simply gives up helping. Either way, the transaction then retries its own operation after completing any helping steps. Because locations are acquired in canonical order, at most $N - 1$ threads can be helping another thread at any time.

Where Shavit and Touitou's approach is lock-free, the group update mechanism of Afek et al. [ADT95] is wait-free. In their methodology, a thread first performs a breadth-first search of a binary tree in order to claim a vacant node. It then "climbs" the tree, atomically merging the queue-ordered lists for pairs of child nodes into a single queue of operations at each step. Then, when it reaches the root of the tree, it performs operations from the head of the queue until its operation has been performed and the result status recorded. As in the approach of Shavit and Touitou, this approach is limited to *static transactions* in which the sequence of memory operations is known in advance. However, their approach adds the distinct advantage that the time complexity of executing a transaction is dependent on contention rather than on the total number of threads.

Anderson and Moir address two limitations of Herlihy's original scheme. First, only a single object could be accessed within the context of a transaction. Anderson and Moir's multi-object scheme [AnM95b] uses a multiword `store_conditional` operation to simultaneously install multiple locally-updated versions of objects in a single atomic operation. Second, their large object scheme [AnM95a] attacks the problem of heavy copying overhead in Herlihy's proposal. Rather than requiring users to explicitly fragment objects into cells, this scheme attempts to provide the illusion that all memory for an object is contained in a single sequential array. Special `Read()` and `Write()` methods translate from "array" indices to the particular block in which the word of interest resides; each group of `BLOCK_SIZE` words is mapped to a different block. Wait-free and lock-free constructions are then built out of multiword `store_conditional` operations on extended-length (`BLOCK_SIZE`) words. The large object scheme avoids copying data and has $\Omega(W)$ time complexity for W-word `load_linked` and `store_conditional` operations.

The first *dynamic* transactional memory implementation is due to Moir [Moi97]. By comparison with static transactions, dynamic transactions allow access to arbitrary blocks chosen while the transaction proceeds. However, Moir's scheme does not allow inclusion of additional blocks once the STM is initialized. In this protocol, each thread has a predefined number of blocks (the exact number is picked to be high enough to support any transaction needed for the object), and builds up copies of object pieces in these blocks. Conflict between transactions is noted when copying out the block as this thread marks a transaction as the current "owner" of a chunk of the object. Finally, when all updates are prepared in the thread's blocks, a multiword `compare_and_swap` operation effects the transaction. This multiword `compare_and_swap` implementation, while being lock-free, establishes sufficient communication with a transaction handler to optionally allow wait-free transactions.

4.2.4 Multiword Atomic Update Implementations

An interesting hardware-based scheme is due to Stone et al. [SSH93]. Like Herlihy and Moss' transactional memory [HeM93], their "Oklahoma Update" is an extension to cache coherence protocols. However, rather than supporting full transactional semantics, they seek merely to provide multi-word atomic updates. We discuss their scheme here because modifications to a number of memory locations, combined with a caching algorithm, such as Barnes's [Bar93] or Moir's [Moi97], can be used to effect a universal construction. The implementation of their scheme consists of adding support for multiple *reservation stations* to the data cache, each of which can hold a tentative update to a cache line. As in `load_linked/store_conditional`, conflicts on a cache line invalidate the stored data. Once a program has added updates for all data that it wishes to update, a single `write-if-reserved` instruction releases all updates, effectively implementing a multi-word `load_linked/store_conditional`: Either all updates are immediately effective, or none of them are.

Another parallel between the Oklahoma Update and Herlihy and Moss's transactional memory is that the hardware updates to support it have not been met with great enthusiasm by hardware vendors. Again, researchers have turned to software-based implementations. The first of these is due to Israeli and Rappoport [IsR94]. They propose a technique that provides a property they name *disjoint access parallelism*. In it, multiple concurrent operations that do not overlap on any memory addresses are able to proceed in parallel. Their NCAS [1] protocols are built out of three successive layers.

The bottom layer is an implementation of LL/SC, based on `compare_and_swap` in which each targetable memory location is extended with an additional bit for each thread. LL in this scheme sets the thread-specific bit for the current thread and retrieves the current contents of the memory. SC checks that the bit is still set, and if so, attempts to update the entire block of memory to the new value and clear all thread bits. In this

[1]Even though some protocols provide multiword LL/SC implementations, I will refer to them as NCAS. The *N* indicates that some number of words larger than 1 may be updated.

way, a successful update by one thread inherently prevents any other pending thread updates from succeeding. The middle layer is a weakened NCAS that relaxes the semantics somewhat by allowing a “component CAS” to succeed if the target memory address already matches its new value and by allowing updates to a prefix of variables to be made even if not all component `compare_and_swap` memory locations match their expected values. The top layer is then built from this weakened NCAS.

Although technically correct, Israeli and Rappoport’s scheme suffers from two serious drawbacks. First, only special memory locations may be the target of an NCAS owing to the need for a per-thread bit in support of the LL/SC implementation. (It is unclear whether this requirement still holds on hardware with native LL/SC support.) More importantly, the bootstrapped character of their implementation results in a protocol that is impractically slow.

Afek et al. [AMT97] present a multi-object, wait-free NCAS implementation. In their scheme, to claim k memory locations, a thread first claims the lower $k - 1$ recursively with the same scheme, and so on down to the 2-location claim, which is handled separately. It then marks the top node as a “parent claim”; each memory location can be claimed as either a child or a parent claim. (If the appropriate claim type is already in use, a thread must help the thread that has installed the claim.) Their protocol then assesses the *conflict graph* between updates, a directed graph in which edges between nodes indicate conflicts between updates (from child claim to parent claim). They then use a node coloring scheme to establish a partial order between conflicting updates (non-conflicting updates are not ordered). Once this “locking order” is finally established, updates can be processed.

Anderson et al. [ARJ97] give a wait-free NCAS implementation based on a novel helping mechanism that they name *incremental helping*. In incremental helping, a thread t_1 first describes its intended update in a shared *announcement* variable. If t_1 finds plans for another thread t_2 in the announcement variable, t_1 helps t_2 before performing its own update. This ensures that a thread will help at most one other thread, but is not disjoint-access parallel due to overlapped use of the announcement variable.

Moir [Moi00] relaxes the heavyweight `load_linked/store_conditional` used as the basis for an earlier universal construction [AnM95a] in order to improve its efficiency. Specifically, he observes that many of the values used by the `LL` portion of the `load_linked/store_conditional` are never used, so he devises a way to generate them only as needed.

Attiya and Dagan [AtD01] present an algorithm that implements binary `load_linked/store_conditional` (`load_linked/store_conditional` on two locations) from unary `load_linked/store_conditional` instructions. They do this by considering the conflict graph induced over multiple binary `load_linked/store_conditional` operations, where nodes represent memory locations and edges represent updates over pairs of memory locations. For conflicts that are a simple path, they adapt the Cole and Vishkin's [CoV86] *deterministic coin tossing* technique to break up the graph so that updates can proceed. For more complex conflict graphs, they introduce a synchronization procedure that reduces the graph to a series of paths. Although this algorithm does not directly support NCAS operations, it can be used as a foundation for techniques that do. For example, an early version of Attiya and Dagan's protocol forms the base case for the NCAS implementation of Afek et al. [AMT97] discussed previously.

Harris, Fraser and Pratt [HFP02] propose an NCAS implementation that is based on a two-layer scheme. First, they build a `double-compare-single-swap` (DCSS) primitive from standard `compare_and_swap` operations. Then, they create a *descriptor* that holds each of the N component `compare_and_swaps`. Threads proceed to walk through each component `compare_and_swap`, conditionally replacing the contents of each target memory address with a pointer to the descriptor if a status word in the descriptor indicates that the update is still in progress. If another descriptor is encountered when doing this update, the thread pauses to perform recursive helping (as introduced by Barnes [Bar93]). If a data value replaced by a pointer to the descriptor doesn't match the expected value, the entire update is aborted and the old values are `compare_and_swap`'ed back; otherwise, once all data values are replaced by pointers to the descriptor, the update status is marked as successful and the new values are `compare_and_swap`'ed in. This scheme has two major advantages. First, it uses no additional storage for memory that can be subject to NCAS, and can thus operate on arbitrary memory locations. Second, it performs comparably to fine-grained locking in benchmark testing conducted by Harris *et al.*

4.2.5 Using Locks as the Basis of Transactions

Yet another approach to creating universal constructions consists of letting programmers write their code to use locks, and then uses the locking as a hook upon which to build nonblocking implementations. The first scheme along these lines that we consider is due to Turek, Shasha, and Prakash [TSP92]. In their scheme, explicit replacements are given for reads, writes, locking actions, and unlocking actions. Specifically, when a thread "acquires" a lock, its operation is helped by other threads that wish to access the same lock. Cyclic data dependencies can only arise if the original lock-based protocol admitted deadlock. A direct consequence of this is that the sequence of operations that constitute a transaction must be explicitly spelled out such that any thread can, at any time, determine what a lock-holder's next step will be. This transaction information plus an encoded instruction pointer are stored in a data structure; threads acquire the lock by CAS'ing a shared data value from `nil` to a pointer to this structure.

A more recent approach is due to Rajwar and Goodman [RaG02]. They base their technique on the observation that programmers often use coarse-grained locking to be sure "all bases are covered" and that programs can often run correctly even if the lock is never acquired. Hence, the conservative locking strategies that programmers often use to ensure correctness can frequently be elided dynamically [RaG01], provided that one can detect and roll back concurrent updates that would have been prevented had the locking been performed. They build on this work on speculative lock elision by automatically wrapping transactions around the critical sections of sequences of instructions detected at runtime as locks. (Their hardware proposal is geared primarily towards identification of TAS and TATAS locks; however, were such a scheme actually implemented in hardware, it seems plausible that explicit `lock` and `unlock` instructions would be added to ease the cost of lock recognition.) While inside a transaction, standard data cache techniques ensure that a thread has sole access to the cache lines that hold the data it plans to update and standard cache invalidation provides a hook that allows conflict between transactions to be detected. To resolve conflict, they use a logical timestamping technique in which transactions win conflicts over younger counterparts; this guarantees that a transaction will eventually be the oldest outstanding one, and hence, systems-wide progress. [2]

Another hardware-based approach that aims to improve performance by converting lock-based to nonblocking code is due to Martínez and Torellas [MaT02]. Their proposal uses thread-level speculation (TLS) to speculatively ignore barriers, locks, and flags. As with Rajwar and Goodman's work, special hardware is used to check for conflict between speculatively and non-speculatively executing threads. Compared to Rajwar and Goodman's work, this proposal requires considerably more hardware investment, and is only applicable to multithreaded machines; Rajwar and Goodman's work is more generally applicable. Martínez and Torellas define a *safe thread* that is always able to execute non-speculatively; this asymmetry guarantees progress systemwide, though it admits starvation of individual operations unless each thread of execution "gets a turn" to be the safe thread.

Oplinger and Lam propose more explicit TLS-based transactions [OpL02]. In their model, programmers explicitly code fine-grained transactional software operations and

[2]This use of timestamps is virtually identical to the way they are used in Alemany and Felten's work [AlF92].

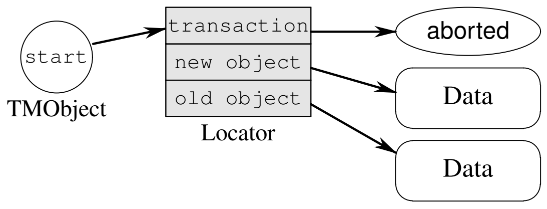


Figure 4.1: Transactional object structure

mark the points at which they have succeeded or failed. In a technique they name *procedural speculation*, transactions are executed in TLS hardware; speculation rollback mechanisms automatically effect transaction rollback on abortion.

4.3 Dynamic Software Transactional Memory

The DSTM system of Herlihy et al. [HLM03b] is an object-based software transactional memory (STM) system. Its transactions operate on blocks of memory. Typically, each block corresponds to one Java object. Each transaction performs a standard sequence of steps: initialize; open and update one or more objects (possibly choosing later objects based on data in earlier objects); attempt to commit; if committing fails, retry. Objects can be opened for full read-write access, read-only access, or for temporary access (where the object can later be discarded if changes to by other transactions won't affect the viability of current one).

Under the hood, each object is represented by a *TMObject* data structure that consists of a pointer to a *Locator* object. The Locator in turn has pointers to the transaction that created it, together with old and new data object pointers (see Figure 4.1).

Transaction *Descriptor*s consist of a *read set* to track objects that have been accessed in read-only mode and a word that reflects the transaction's current status: `aborted`, `active`, or `committed`.

When a transaction attempts to *access* an object (for read-only permission), we first read the Locator pointer in the TMObject. We then read the status word for the Loca-

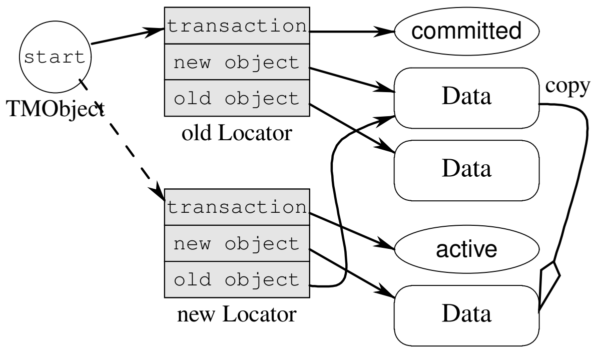


Figure 4.2: Opening a TMObject after a recent commit

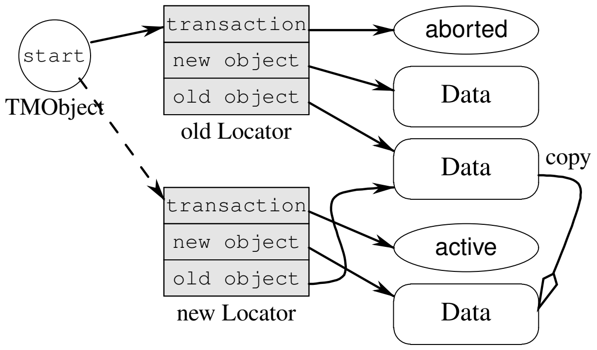


Figure 4.3: Opening a TMObject after a recent abort

tor's transaction Descriptor to determine whether the old or the new data object pointer is current: If the status word is `committed` the new object is current (Figure 4.2); otherwise the old one is (Figure 4.3). Next, we build a new Locator that points to our transaction and has the active version of the data as its old object. We copy the data for the new object and then atomically update the TMObject to point to our new Locator. Finally, we store the new Locator and its corresponding TMObject in our Descriptor's read set.

To *acquire* read-write permissions for an accessed object, we build a new Locator that points to our Descriptor and has the current version of the data as its old object. We instantiate a new object with a copy of the target object's data and then atomically update the TMObject to point to our new Locator; this implicitly aborts the competing transaction by guaranteeing that any attempt it makes to commit will fail.

We validate the viability of a transaction by verifying that each Locator in the read set is still current for the appropriate TMObject. Validating already-accessed objects with each new object accessed or acquired ensures that a transaction never sees mutually inconsistent data; hence, programming transactions for DSTM is equivalent to sequential programming from the user's perspective.

To commit a transaction, we atomically update its Descriptor's status word from `active` to `committed`. If successful, this update signals that all of the transaction's updated TMObjects are now current.

With this implementation, only one transaction at a time can acquire an object for write access because a TMObject can point to only one transaction's Locator. If another transaction wishes to access an already-acquired object, it must first atomically update the "enemy" transaction's status field from `active` to `aborted`. We invoke a *contention manager* to decide whether to abort the enemy transaction or delay our own.

4.3.1 Visible and Invisible Reads

In the original version of the DSTM, read-only access to objects is achieved by storing TMObject→Locator mappings in a private read set. At validation time, a conflict is detected if the current and stored Locators do not match. We dub this implementation *invisible* because it associates no artifact from the reading transaction with the object. A competing transaction attempting to acquire the object for write access cannot tell that readers exist, so there is no "hook" through which contention management can address a potential conflict.

Alternatively, read-only accesses can be made visible by adding the Descriptor to a linked list of readers associated with the TMObject. This implementation adds overhead to both read and write operations: A writer that wishes to acquire the object must explicitly abort each reader in the list. In exchange, we gain the ability to explicitly manage conflicts between readers and writers, to abort doomed readers early, and to skip incremental validation of accessed objects when accessing or acquiring a new object.

4.3.2 Benchmarks

In this chapter, we present experimental results with seven different benchmarks. Three implementations of an integer set (IntSet, IntSetUpgrade, RBTreeTMNodeStyle) are drawn from the original DSTM paper [HLM03b]. These three repeatedly but randomly insert or delete integers in the range $0 \ldots 255$. The first two implementations are based on a sorted linked list. In IntSet, every object is acquired for write access; every pair of transactions necessarily conflict. IntSetUpgrade initially accesses objects read-only and acquires them in read/write mode as needed. This allows some concurrency between transactions, but a transaction that performs a write earlier in the list will abort a longer-running transaction.

RBTreeTMNodeStyle, like IntSetUpgrade, upgrades access as needed; however, it implements the integer set with a red-black tree. It has two principle differences from the IntSet benchmarks, however. First, modified objects can be in completely different sub-branches of the tree; many transactions can execute in parallel without affecting each other. Second, the process of rebalancing the tree after an insert or delete operation works upwards to the root of the tree. Hence, a transaction coming up can meet one heading down; the resulting cyclic dependencies will tend to punish contention management schemes that are overly reluctant to abort a competitor.

The fourth benchmark (Stack) is a concurrent stack that supports push and pop transactions. Similarly, the fifth (Counter) is a simple concurrent counter. Transactions in these two benchmarks are notable for being very short; that every pair is in conflict is a secondary consideration.

Transactions in the sixth benchmark (ArrayCounter) consist of either ordered increments or decrements in an array of 256 counters. Increment transactions update each counter in turn, starting at 0 and working toward 255 in ascending order before committing; decrements reverse the order. Not only do every pair of transactions conflict with each other, but an increment and a decrement are very likely to repeatedly encounter each other somewhere in the middle of the array. We designed ArrayCounter as a "torture test" to stress contention managers' ability to avoid livelock.

By far the most complex benchmark we test with, the seventh (LFUCache) simulates cache replacement in an HTTP web proxy using the least-frequently used (LFU) algorithm [RoD90]. Briefly, the LFU algorithm assumes that frequency (rather than recency) of web page access is the best predictor for whether a web page is likely to be accessed again in the future (and thus, worth caching).

The simulation uses a two-part data structure to emulate the cache. The first part is a lookup table of 2048 integers, each of which represents the hash code for an individual HTML page. These are stored as a single array of `TMObjects`. Each contains the key value for the object (an integer in our simulation) and a pointer to the page's location in the main portion of the cache. The pointers are null if the page is not currently cached.

The second, main part of the cache consists of a fixed size priority queue heap of 255 entries (a binary tree, 8 layers deep), with lower frequency values near the root. Each priority queue heap node contains a frequency (total number of times the cached page has been accessed) and a page hash code (effectively, a backpointer to the lookup table). The priority queue heap is inverse: Lower frequency values are generally closer to the root than higher values.

Worker threads repeatedly access a page. To approximate the workload for a real web cache, we pick pages randomly from a Zipf distribution with exponent 2. So, for page i, the cumulative probability $p_c(i) \propto \sum_{0<j\leq i} j^{-2}$. We precompute this distribution (normalized to a sum of one million) so that a page can be chosen with a flat random number.

The algorithm for "accessing a page" first finds the page in the lookup table and reads its heap pointer. If that pointer is non-null, we increment the frequency count for the cache entry in the heap and then reheapify the cache using backpointers to update lookup table entries for data that moves. If the heap pointer is null, we replace the root node of the heap (guaranteed by heap properties to be least-frequently accessed) with a node for the newly accessed page. In order to induce hysteresis and give pages a chance to accumulate cache hits, we perform a modified reheapification in which the new node switches place with any children that have the *same* frequency count (of one).

The Zipf distribution we use for page access will tend to cluster most hits into only a few pages, which will quickly reach the leaves of the priority queue heap. Thereafter, some parallelism results from threads hitting different pages, but most transactions will tend to be in conflict. Reheapification of the priority queue heap can be expected to be fairly rare, so like the Stack and Counter benchmark, most transactions will be very short. Even for transactions that do cause reheapification, object access is always from the tree of the priority queue heap down towards the leaves; this unidirectional access pattern means that transactions will not repeatedly "bump into" each other.

4.3.3 Experimental Methodology

All results presented in this chapter were obtained on a SunFire 6800, a cache-coherent multiprocessor with 16 1.2GHz UltraSPARC III processors. Except where otherwise noted, we tested in Sun's Java 1.5 beta 1 HotSpot JVM, augmented with a JSR 166 update jar file from Doug Lea's web site [Lea].

Our benchmark testing consists of running a variable number of threads, each executing random transactions in one of the benchmarks from Section 4.3.2. In each case, if a transaction fails to commit, the thread immediately retries it; otherwise, it selects and executes a new random transaction without delay. In particular, we use no non-critical work for these experiments as was done in Chapter 2.

Operations in these benchmarks have been artificially restricted to exacerbate contention between transactions and highlight differences between contention managers (described in detail in Section 4.4). For example, in the integer set benchmarks, only integers in the range 0. . . 255 are inserted or deleted; a greater range would decrease the probability of transactional conflict – particularly in the case of RBTreeTMNodeStyle.

4.4 Contention Management

One of my primary contributions to the original DSTM system was to augment it with a modular interface for "plug-in" contention managers that separate issues of progress from the correctness of a given data structure. The central goal of a good contention manager is to mediate transactions' conflicting needs to access data objects.

A key benefit of obstruction-freedom is that it allows programmers to separate concerns of progress from those of correctness. In particular, obstruction-free algorithms have no hard and fast progress requirements when more than one thread is executing concurrently; this allows the use of heuristics that exploit information about time, runtime load, the hardware and software environments, or even the specific types of transactions being executed – practical sources of information that have been largely ignored in the literature on lock-free synchronization. A contention manager, then, wraps a collection of heuristics into a single policy that aims to maximize throughput at some reasonable level of fairness, balancing the complexity of the decision-making process against the runtime overhead it incurs.

Any obstruction-free algorithm can be augmented with a variety of contention management policies. For example, the Adaptive STM [MSS05] implements the same contention management interface as DSTM, so it can use the same managers. Other algorithms, such as the obstruction-free deque of Herlihy et al. [HLM03a] are more restricted: Because individual operations do not create any visible interim state, threads cannot identify the peers with which they are competing. This precludes use of some of the more context-sensitive policies detailed below, but other options remain viable: No special information about competitors is required, for example, to employ exponential backoff on each failed attempt to complete an operation. DSTM includes a particularly rich set of information-providing "hooks", yielding a vast design space for contention management policies.

The correctness requirement for contention managers is simple and quite weak. Informally, any active transaction that asks sufficiently many times must eventually get permission to abort a conflicting transaction. More precisely, every call to a contention manager method eventually returns (unless the invoking thread stops taking steps for some reason), and every transaction that repeatedly requests to abort another transaction is eventually granted permission to do so. This requirement is needed to preserve obstruction-freedom: A transaction T that is forever denied permission to abort a conflicting transaction will never commit even if it runs by itself. [3] If the conflicting transaction is also continually requesting permission to abort T, and incorrectly being denied this permission, the situation is akin to deadlock. Conversely, if T is eventually allowed to abort any conflicting transaction, then T will eventually commit if it runs by itself for long enough.

At one extreme, a policy that never aborts an "enemy" transaction [4] can lead to deadlock in the event of priority inversion or mutual blocking, to starvation if a transaction deterministically encounters enemies, and to a major loss of performance in the face of page faults and preemptive scheduling. At the other extreme, a policy that always aborts an enemy may also lead to starvation, or to livelock if transactions repeatedly restart and then at the same step encounter and abort each other. A good contention manager must lie somewhere in between, aborting enemy transactions often enough to tolerate page faults and preemption, yet seldom enough to make starvation unlikely in practice. The contention manager's duty is to ensure progress; we say that it does so

[3]Here and elsewhere "runs by itself" means that no concurrent transaction takes a step, not that no concurrent transaction exists.

[4]Never aborting an enemy violates the requirements we just presented for a policy to ensure obstruction-freedom; however, it is useful to consider for illustrative purposes.

out-of-band because its code is orthogonal to that of the transactions it manages, and neither contributes to their conceptual complexity nor affects their correctness.

4.4.1 The Contention Management Interface

DSTM's contention management interface [ScS04a] comprises notification methods, "hooks" for various events that transpire during the processing of transactions; and a request method that asks the manager to decide whether enemy transactions should be aborted. Notifications include:

- Beginning a transaction
- Successfully committing a transaction
- Failing to commit a transaction
- Self-abortion of a transaction
- Beginning an attempt to open an object (for read-only, temporary, or read-write access)
- Successfully opening an object (3 variants)
- Failing to open an object (3 variants) due to failed transaction validation
- Successfully changing to read-only, temporary, or read-write access an object already open in another mode (6 total variants)

Each thread has its own instance of a contention manager; contention management is distributed rather than centralized. By tracking the notification messages that occur in the processing of a transaction, a contention manager assembles information that allows it to decide whether aborting a competing transaction will improve overall throughput. Although in principle there is no reason different threads cannot have contention managers of different types, in practice, we believe that cooperative implementation of policy distributed across all managers is key to their effectiveness; having managers trying to effect multiple policies concurrently subverts this benefit.

4.4.2 Contention Management Policies

In this section, we enumerate a series of example policies that one might use for contention management. Because the design space for contention managers is extremely wide and because there is little prior work to guide us in this area, we have focused in these early studies on the "broad brush" technique of covering many widely different policy options. We anticipate that subsequent studies will narrow in for more detailed analysis of individual policies that we have identified here.

The policies described in this section reflect both our initial study of contention management [ScS04a] and a follow-on study that incorporated lessons learned from it [ScS05].

Aggressive

The Aggressive manager ignores all notification methods, and always chooses to abort an enemy transaction at conflict time. Although this makes it highly prone to livelock, it forms a useful baseline against which to compare other policies.

Polite

The Polite contention manager [5] uses exponential backoff to resolve conflicts encountered when opening objects. Upon detecting contention, it spins for a period of time randomly selected from the range $1 \ldots 2^n k$ ns, where n is the number of retries

[5]Before we modularized its contention management interface, DSTM always used this scheme for conflict resolution.

that have been necessary so far for access to an object, and k is an architectural tuning constant (4 works well on our machine). After a maximum of 22 retries, the polite manager unconditionally aborts an enemy transaction. One might expect the Polite manager to be particularly vulnerable to performance loss due to preemption and page faults.

Randomized

A very simple contention manager, the Randomized policy ignores all notification methods. When it encounters contention, it flips a coin to decide between aborting the other transaction and waiting for a random interval of up to a certain length. The coin's bias and the maximum waiting interval are tunable parameters; we used 50% and 64 ns, respectively. Note that DSTM transactions coupled with this policy are "merely" probabilistically obstruction-free: Executions exist in which the coin comes up "wait" infinitely often. But with probability 1, they do not happen in practice.

Karma

The Karma manager attempts to judge the amount of work that a transaction has done so far when deciding whether to abort it. Although it is hard to estimate the amount of work that a transaction performs on the data contained in an object, the number of objects the transaction has opened may be viewed as a rough indication of investment. For system throughput, aborting a transaction that has just started is preferable to aborting one that is in the final stages of an update spanning tens (or hundreds) of objects.

The Karma manager tracks the cumulative number of objects opened by a transaction as its priority. More specifically, it resets the priority of the current thread to zero when a transaction commits and increments that priority when the thread successfully opens an object. When a thread encounters a conflict, the manager compares priorities and aborts the enemy if the current thread's priority is higher. Otherwise, the manager waits for a fixed amount of time to see if the enemy has finished. Once the number of retries plus the thread's current priority exceeds the enemy's priority, the manager kills the enemy transaction.

What about the thread whose transaction was aborted and has to start over? In a way, we owe it a karmic debt: It was killed before it had a chance to finish its work. We thus allow it to keep the priority ("karma") that it had accumulated before being killed, so it will have a better chance of being able to finish its work in its "next life". Note that every thread necessarily gains at least one point in each unsuccessful attempt. This allows short transactions to gain enough priority to compete with others of much greater lengths.

Eruption

The Eruption manager is similar to the Karma manager in that both use the number of opened objects as a rough measure of investment. It resolves conflicts, however, by increasing pressure on the transactions that a blocked transaction is waiting on, eventually causing them to "erupt" through to completion. Each time an object is successfully opened, the transaction gains one point of "momentum" (priority). When a transaction finds itself blocked by one of higher priority, it adds its momentum to the conflicting transaction and then waits for it to complete. Like the Karma manager, Eruption waits for time proportional to the difference in priorities before killing an enemy transaction.

The reasoning behind this management policy is that if a particular transaction is blocking resources critical to many other transactions, it will gain all of their priority in addition to its own and thus be much more likely to finish quickly and stop blocking the others. Hence, resources critical to many transactions will be held (ideally) for short periods of time. Note that while a transaction is blocked, other transactions can accumulate behind to increase its priority enough to outweigh the transaction behind which it is blocked.

In addition to the Karma manager, Eruption draws on Tune *et al.*'s `QOldDep` and `QCons` techniques for marking instructions in the issue queue of a superscalar out-of-order microprocessor to predict instructions most likely to lie on the critical path of execution [TLT01].

KillBlocked

Adapted from McWherter *et al.*'s POW lock prioritization policy [MSA04], the KillBlocked manager is less complex than Karma or Eruption, and features rapid elimination of cyclic blocking. The manager marks a transaction as blocked when first notified of an (unsuccessful) non-initial attempt to open an object. The manager aborts an enemy transaction whenever (a) the enemy is also blocked, or (b) a maximum waiting time has expired.

Kindergarten

Based loosely on the conflict resolution rule in Chandy and Misra' Drinking Philosophers problem [ChM84], the Kindergarten manager encourages transactions to take turns accessing an object. For each transaction T, the manager maintains a list (initially empty) of enemy transactions in favor of which T has previously aborted. At conflict time, the manager checks the enemy transaction and aborts it if present in the list; otherwise it adds the enemy to the list and backs off for a short length of time. It also stores the enemy's hash code [6] as the transaction on which T is currently waiting. If after a fixed number of backoff intervals it is still waiting on the same enemy, the Kindergarten manager aborts transaction T. When the calling thread retries T, the Kindergarten manager will find the enemy in its list and abort it.

[6]obtained from the Java method `Object.hashCode()`

Timestamp

The Timestamp manager is an attempt to be as fair as possible to transactions. The manager records the current time at the beginning of each transaction. When it encounters contention between transaction T and some enemy E, it compares timestamps. If T's timestamp is earlier, the manager aborts E. Otherwise, it begins waiting for a series of fixed intervals. After half the maximum number of these intervals, it flags E's transaction as potentially defunct. After the maximum number of intervals, if E's defunct flag has been set all along, the T aborts E. If E's flag has ever been reset, however, the manager doubles T's wait period and restarts. Meanwhile, if E performs any transaction-related operations, its manager will see and clear the defunct flag.

Timestamp's goal is to avoid aborting an earlier-started transaction regardless of how slowly it runs or how much work it performs. The defunct flag provides a feedback mechanism for the other transaction to enable us to distinguish a dead or preempted transaction from one that is still active. Of course, using timestamps to resolve contention is hardly new; similar algorithms have been in use in the database community for at least a quarter-century [BeG80].

QueueOnBlock

The QueueOnBlock manager reacts to contention by linking itself into a queue hosted by the enemy transaction. It then spins on a "finished" flag that is eventually set by the enemy transaction's manager at completion time. Alternatively, if it has waited for too long, it aborts the enemy transaction and continues; this is necessary to preserve obstruction freedom. For its part, the enemy transaction walks through the queue setting flags for competitors when it is either finished or aborted. Note that not all of these competitors need have been waiting for the same object. If more than one was, any that lose the race to next open it will enqueue themselves with the winner.

Clearly, QueueOnBlock does not effectively deal with object dependency cycles: At least one transaction must time out before any can progress. On the other hand, if the object access pattern is free of such dependencies, this manager will usually avoid aborting another transaction.

PublishedTimestamp

In our initial assessment, we found that a major disadvantage of the Timestamp protocol in the length of time it needs to abort an inactive (usually preempted) transaction. To remedy this, we leverage a heuristic [HS05] that provides a high quality estimate of whether a thread is currently active. Adapting the heuristic to this setting, transactions update a "recency" timestamp with every notification event or query message. A thread is presumed active unless its recency timestamp lags the global system time by some threshold.

PublishedTimestamp aborts an enemy transaction E whose recency timestamp is old enough to exceed E's inactivity threshold. This inactivity threshold is reset to an initial value ($1\mu s$) each time a thread's transaction commits successfully. When an aborted transaction restarts, we double its threshold (up to a maximum of $2^{15}\mu s$).

Just as in the Timestamp manager, a transaction will abort any transaction it meets whose base timestamp is newer than its own. The base timestamp is reset to the system time iff the previous transaction committed successfully.

Polka

In our initial studies, we found that Karma and Polite were frequently among the best performing contention managers, though neither gave reasonable performance on all benchmarks. To create a combination manager that merges their best features, we combine Polite's randomized exponential backoff with Karma's priority accumulation mechanism. The result, Polka (named for the managers it joins), backs off for a number of intervals equal to the difference in priorities between the transaction and its enemy. Unlike Karma, however, the length of these backoff intervals increases exponentially.

As we will note in the experimental portion of this Section, our results suggest that writes are considerably more important than reads for many of our benchmarks. Accordingly, the Polka manager unconditionally aborts a group of (visible) readers that hold an object needed for read-write access.

4.4.3 Experimental Results

In our initial assessment of contention manager performance, we ran each benchmark–manager pairing with both visible and invisible read implementations. For each combination, we varied the level of concurrency from 1 to 128 threads, running individual tests for 10 seconds. We present results averaged across three test runs.

Figures 4.4–4.8 show averaged results for the counter and LFUCache benchmarks, the read-black tree-based integer set benchmark, and the two linked list-based integer set benchmarks. Each graph is shown both in total and zoomed in on the first 16 threads (where multiprogramming does not occur).

Second-round Experiment Results

In our second round of experiments, we ran each benchmark–manager pairing with both visible and invisible read implementations. For each combination, we varied the level of concurrency from 1 to 48 threads, running individual tests for 10 seconds. We present results averaged across three test runs.

Figures 4.9 and 4.10 display throughput for the various benchmarks. For the benchmarks that acquire all objects for read-write access (Figure 4.9), differences in overhead for supporting the two types of reads are minimal; we show only invisible reads.

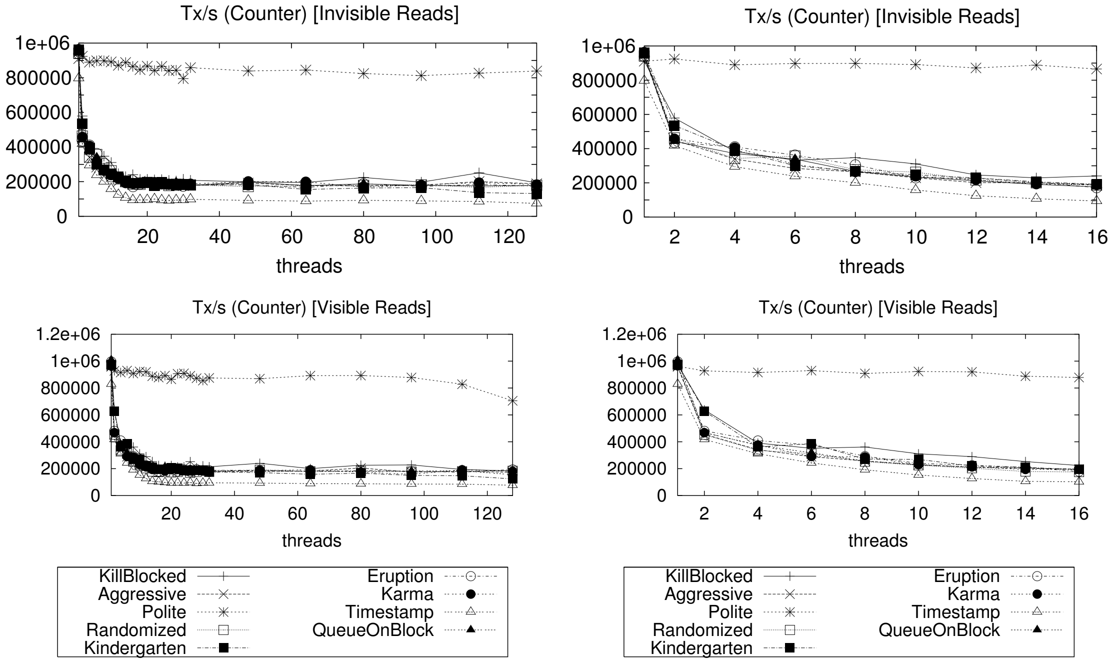


Figure 4.4: Counter benchmark performance

4.4.4 Discussion

Initial Experiment Discussion

From figures 4.4 – 4.8, we see that contention management is extremely important to achieving good performance from the DSTM system. In particular, for all benchmarks except IntSetUpgrade, the difference in performance between best- and worst-performing managers is at least a factor of 10. Upon closer inspection, we see that different contention management policies work better for different benchmark applications, and that no single manager provides all-around best results. However, the group of managers that yield top performance is fairly small: Polite does well for many benchmarks, but Karma, Eruption, and Kindergarten yield good performance in each case where Polite does less well.

The choice between visible and invisible reads, however, is not nearly as difficult. While visible reads outperform invisible reads with the IntSet benchmarks, invisible

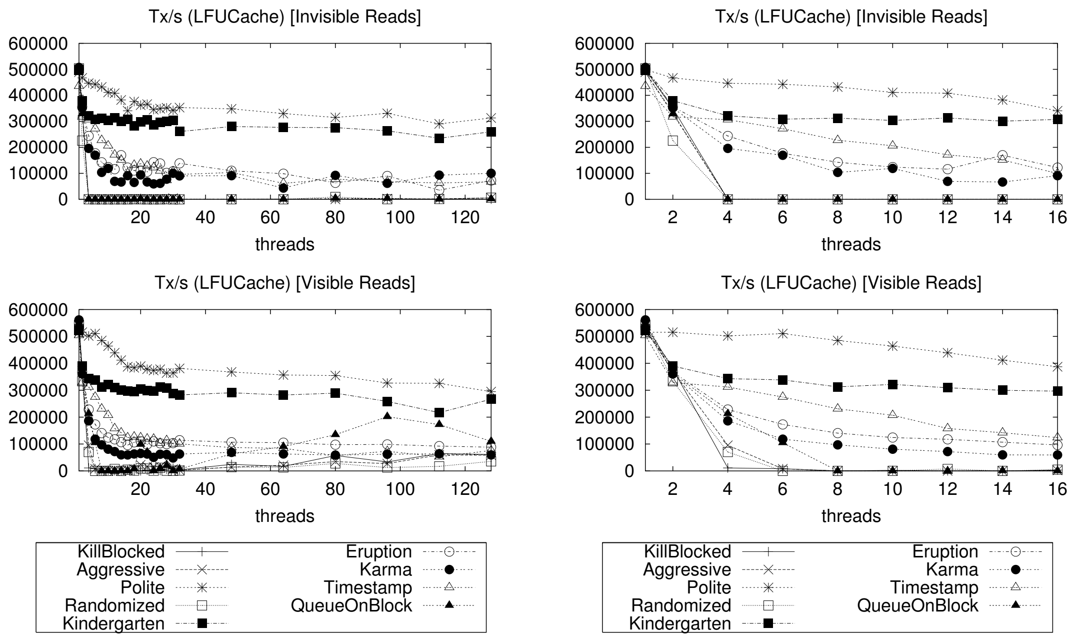


Figure 4.5: LFUCache benchmark performance

reads only outperform visible reads with very high levels of contention in the RBTree benchmark.

Second-round Experiment Discussion

The throughput graphs in Figures 4.9 and 4.10 illustrate that the choice of contention manager is crucial. Except with invisible reads in the IntSetUpgrade benchmark, the difference between a top-performing and a bottom-performing manager is at least a factor of 4.

For each of the write-access benchmarks (Figure 4.9), every pair of transactions conflict, so the best possible result is to achieve flat throughput irrespective of the number of threads. As is clearly visible, the Polka manager comes very close to achieving this goal for Stack and IntSet, and delivers by far the best performance for the Array-Counter benchmark.

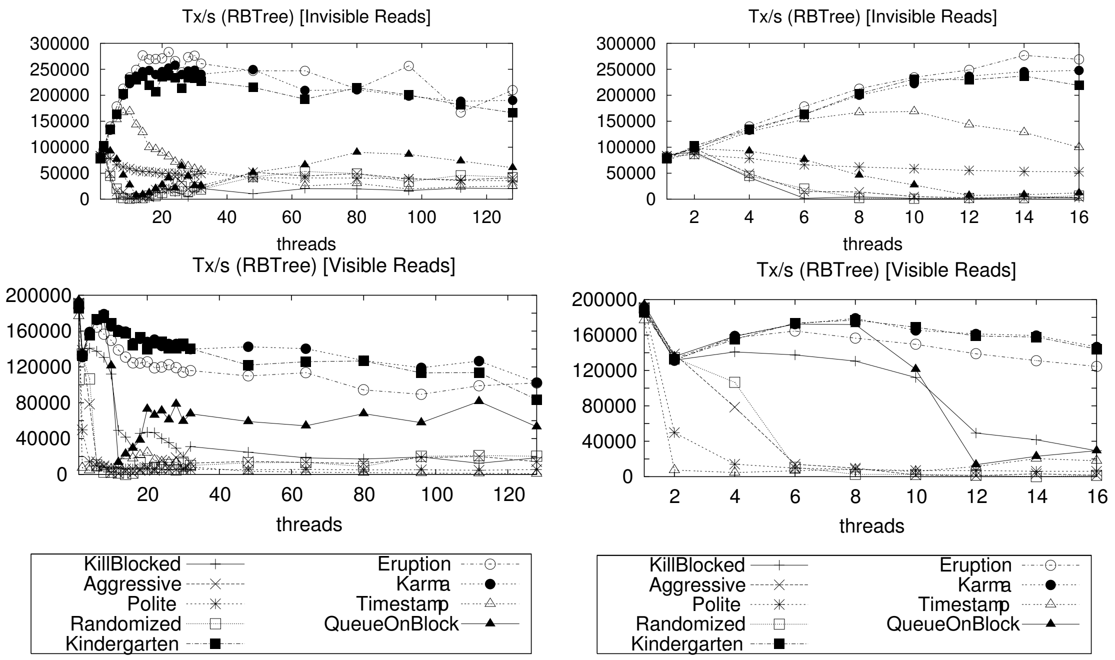


Figure 4.6: RBTree benchmark performance

For these benchmarks, good performance requires one transaction to dominate the others long enough to finish. Karma and Eruption perform well precisely because of their priority accumulation mechanisms. However, good performance also requires transactions to avoid memory interconnect contention caused by repeated writes to a cache line. Polka's increasing backoff periods effect this second requirement in a manner analogous to the way that TATAS spin locks with exponential backoff outperform those without [And90].

We confirm this hypothesis by comparing Polka to an equivalent manager (Karmexp, not shown) in which the backoff periods are fixed, but the number of them needed to "overtake" an enemy transaction increases exponentially as a function of the difference in priorities. Even though the same length of time overall must elapse before a transaction is able to abort its enemy, and all other management behavior is identical, Karmexp livelocks on the ArrayCounter benchmark.

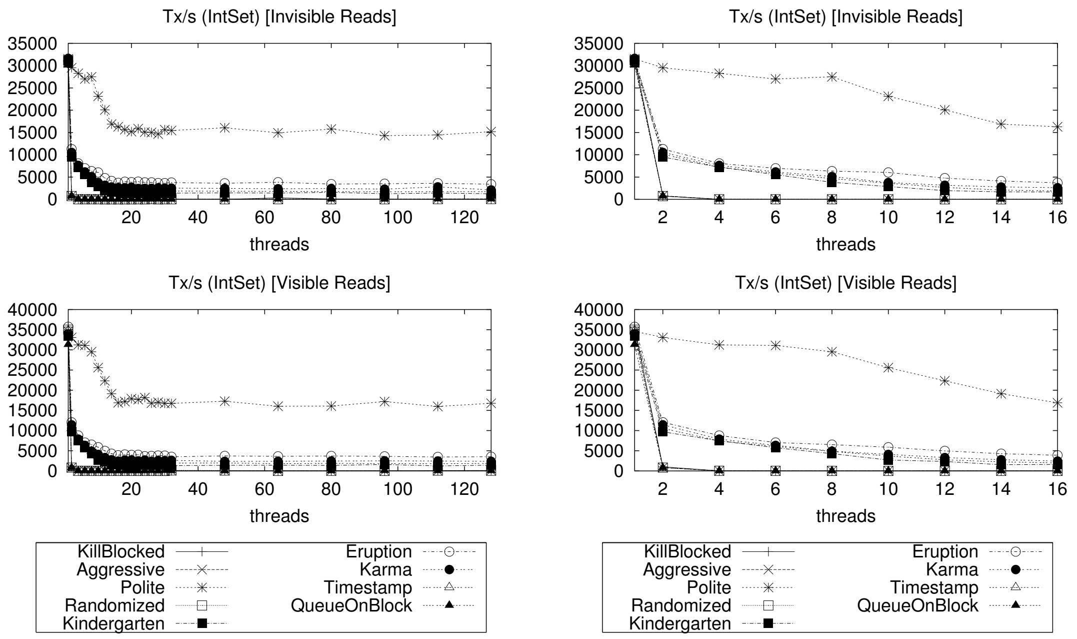


Figure 4.7: IntSet benchmark performance

LFUCache Throughput Great disparity between managers can be found in the LFUCache benchmark. In this benchmark, the vast majority of transactions consist of reading a pointer then incrementing a counter for a leaf node in the priority queue heap. As such, LFUCache is heavily write-dominated, and yields results similar to the write-access benchmarks. The results show greater "spread" however, because LFUCache offers more concurrency than the purely write-access benchmarks. The second-place finish of Kindergarten may be attributed to its strong ability to force threads to take turns accessing the leaf nodes. The difference between visible and invisible reads is very small, yielding further evidence that write performance is the dominant concern in this benchmark.

Red-Black Tree Throughput A typical transaction in the RBTree benchmark consists of accessing objects in read-only mode from the root of the tree down to an insertion/deletion point, then performing fix-ups that restore balance to the tree, working

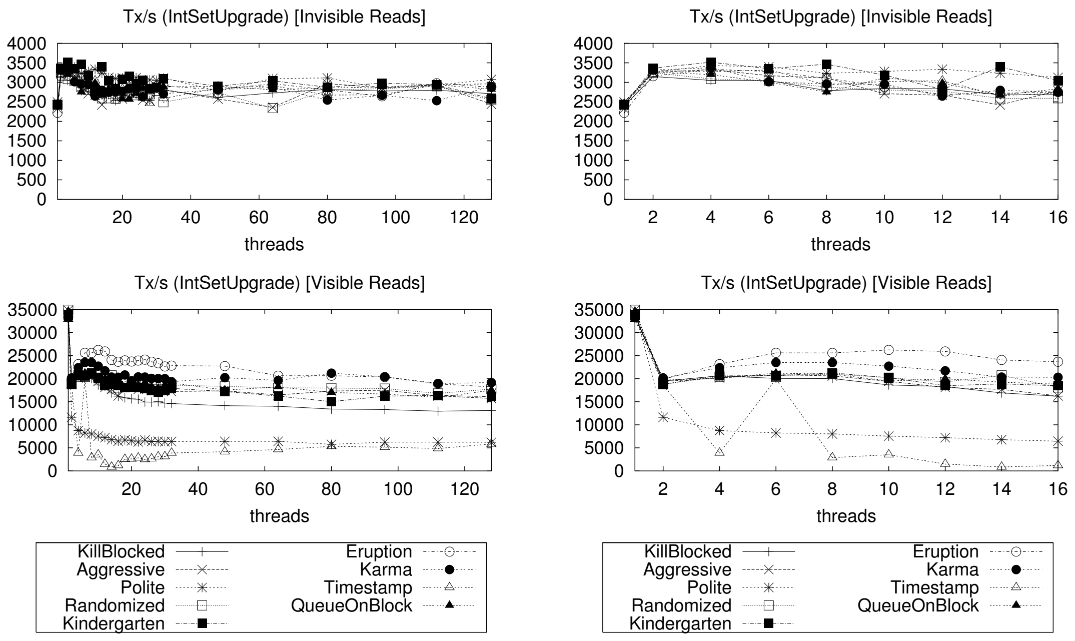


Figure 4.8: IntSetUpgrade benchmark performance

upward toward the root and acquiring objects as needed. Hence, any transaction acquiring objects is nearly done: the writes are much more important than the reads. Further, when a writer aborts a reader, it is likely to re-encounter that reader on its way back toward the root unless it finishes quickly.

Individual tree nodes tend to become localized hot-spots of contention as a transaction coming up from one child node meets another that came up from the other child or a reader working its way down the tree. This is why the Eruption manager performs so well here: not only does it have a strong mechanism for selecting one transaction over the others, but its priority transfer mechanism gives a boost to the winner for any subsequent conflicts with the loser. By comparison, Karma's priority retention allows two similarly-weighted transactions to repeatedly fight each time they meet. The Timestamp manager performs similarly to Eruption because its resolution mechanism ensures that conflict between any pair of transactions is always resolved the same way.

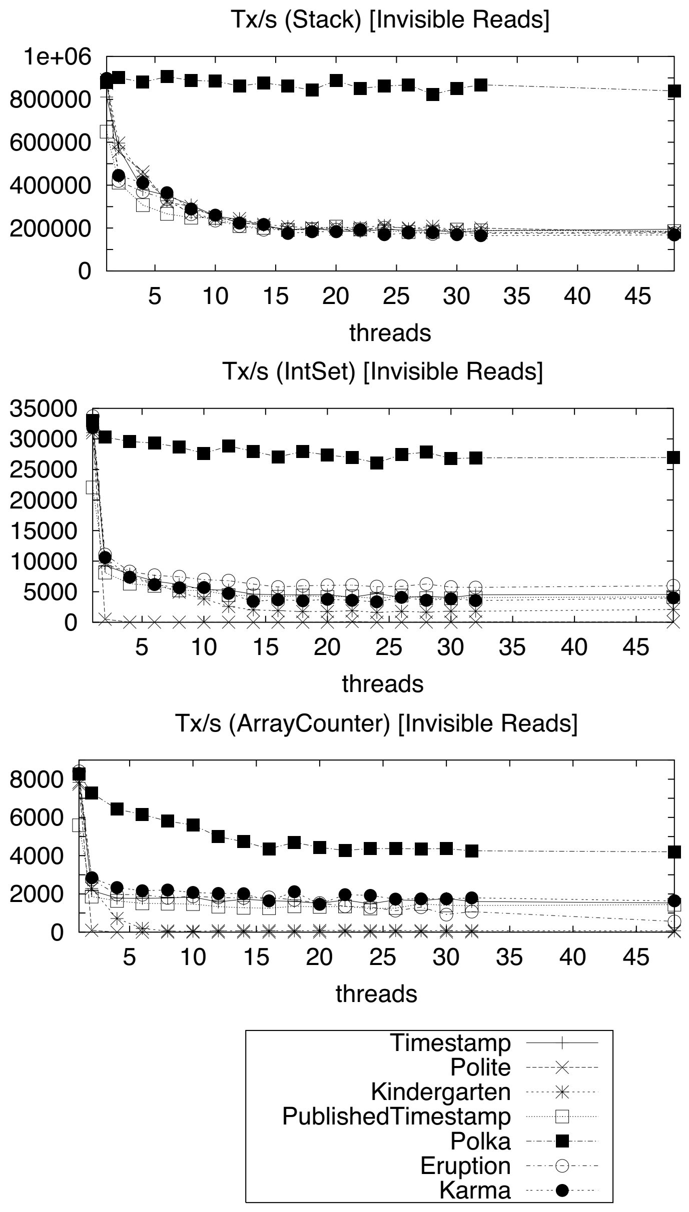


Figure 4.9: Benchmarks with only write accesses: throughput by thread count

Comparing read implementations, we observe that up through 4 threads, throughput is far stronger with visible than invisible reads. We attribute this to validation overhead: with invisible reads, each time a transaction accesses or acquires a new object, it must first re-validate each object it had previously accessed for read-only access. Hence,

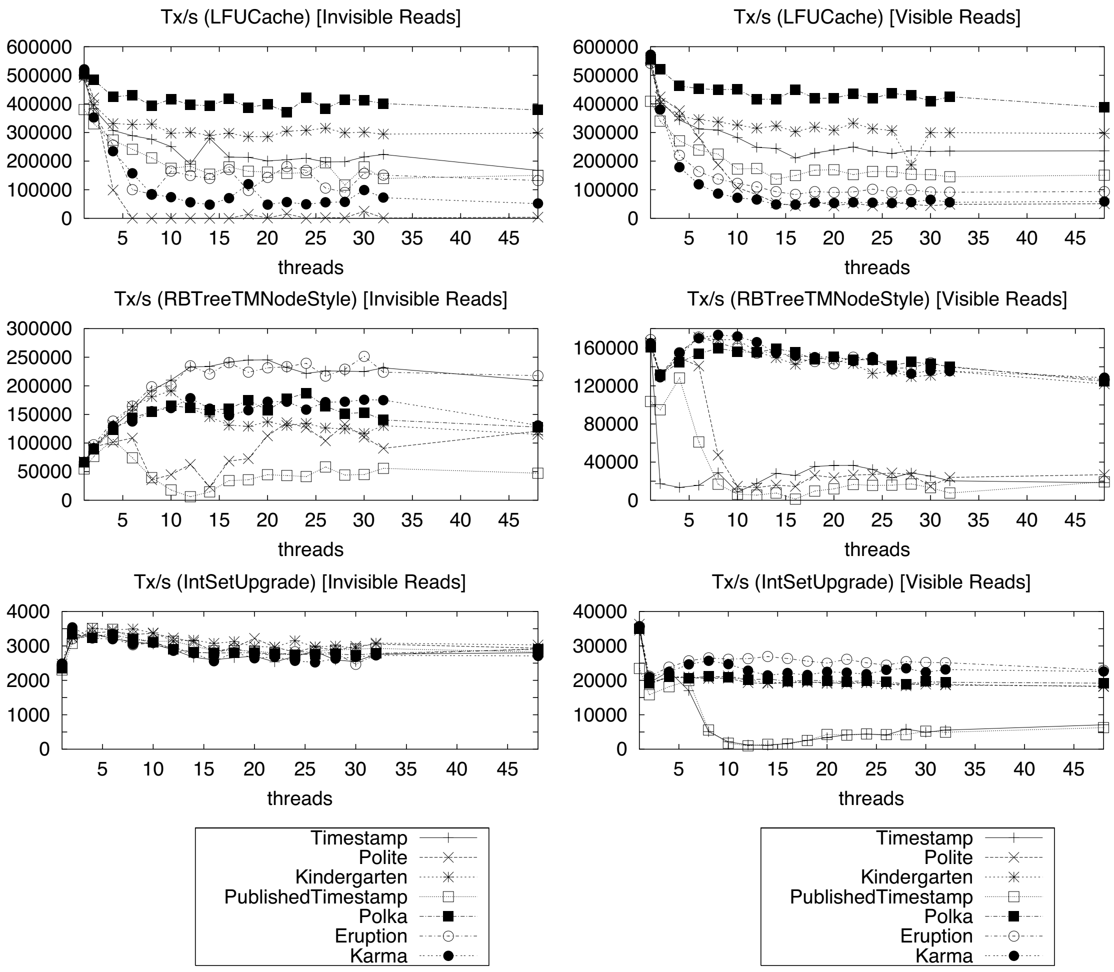


Figure 4.10: LFUCache, RBTree, IntSetUpgrade throughput results: invisible (left) and visible (right) reads. Note that Y-axis scales differ considerably for the RB-TreeTMNodeStyle and IntSetUpgrade benchmarks.

validation overhead is quadratic in the number of read objects ($V = O(R(R+W))$ for R read-access objects and W read-write access objects). By comparison, visible reads reduce this overhead to $O(R)$. Beyond 4 threads, contention increases enough that validation overhead pales in comparison .

Considering specific managers, the preeminence of writes greatly hurts the Timestamp manager in particular: With visible reads, a transaction that is nearly complete must wait behind a reader even if it needs only one final object in write mode. We confirmed this by creating a Timestamp variant that unconditionally aborts readers; it yields top-notch performance on the RBTree benchmark.

IntSetUpgrade Throughput In the IntSetUpgrade benchmark, as in the red-black tree, transactions consist of a series of reads followed by a limited number (1) of writes. Once again, we see that validation overhead incurs a large throughput penalty for invisible reads.

With invisible reads, transactions are only aware of other transactions when they attempt to open an object that the other has acquired for read-write access. Here, virtually any delay, such as that inherent to the function-call overhead of checking to see whether the other transaction should be aborted, is sufficient to allow it to finish. As expected, the difference in throughput between managers is minimal.

With visible reads, the Karma and Eruption managers allow a long transaction (e.g., one that makes a change near the end of a list) to acquire enough priority that writers are likely to wait before aborting them. This allows both transactions to complete without restarting. If shorter transactions were to complete first, longer transactions would have to restart. In summary, Karma and Eruption gain a small edge by helping to ensure that transactions complete in reverse size order, and the Timestamp variants suffer greatly from the randomness of which transaction happens to be older. Polite, Polka, and Kindergarten, meanwhile, back off for long enough to give longer transactions a better chance to complete, but do not directly ensure this ordering.

Throughput Results Summary No single manager outperforms all others in every benchmark, but Polka achieves good throughput even in the cases where it is outperformed. As the first manager we have seen that does so, it embodies a good choice for a default contention manager.

As we see from the RBTree and IntSetUpgrade benchmarks, the tradeoff between visible and visible reads remains somewhat application-specific: Visible reads greatly reduce the validation overhead when accessing or acquiring new objects, but they require bookkeeping updates to the object that can exacerbate contention for the memory interconnect.

4.4.5 Prioritized Contention Management

In this Section we introduce the *prioritized contention management problem*, wherein each thread has a *base priority* P_b that ranks its overall importance relative to other threads. Following Waldspurger and Weihl [WaW94], we aim for *proportional-share* management, where each thread's cumulative throughput is proportional to its base priority: Over any given period of time, a thread with base priority 3 should ideally complete 50% more work than one with base priority 2.

There is in general no clear way to add priorities to lock-free algorithms: The desire to guarantee progress of at least one thread and the desire to enable a higher-priority thread to "push" lower-priority threads out of its way are difficult at best to reconcile. By comparison, prioritization is a natural fit for obstruction-freedom and DSTM; prioritizing the contention managers presented below was relatively straightforward. The modularity and fine-grained control offered by the contention management interface are an inherent benefit of DSTM and obstruction freedom. Herlihy et al. argue the same point in their DSTM paper [HLM03b].

Karma, Eruption, and Polka

Prioritized variants of these managers add P_b (instead of 1) to a transaction's priority whenever it successfully opens an object. This adjusts the rate at which the transaction "catches up" to a competitor by the ratio of base priorities for the two transactions.

Timestamp and PublishedTimestamp

Prioritized versions of these managers retain their transaction's timestamp through P_b committed transactions. Essentially, they are allowed to act as the oldest extant transaction several times in a row.

Kindergarten

To prioritize the Kindergarten manager, we randomize updates to the list of transactions in favor of which a thread has aborted to probability $P_b(t)/[P_b(t)+P_b(e)]$ for a transaction with base priority $P_b(t)$ and an enemy with base priority $P_b(e)$. Intuitively, rather than "taking turns" equally for an object, this randomization biases turns in favor of the higher-priority transaction. Because a thread could infinitely often decline to update the list, this policy is probabilistically obstruction-free.

Experimental Results

To evaluate our prioritization schemes, we ran combinations of benchmarks, managers, and read implementations for 16 seconds. We present results for a single typical test run, showing the individual cumulative throughput for each of 4 or 8 threads as a function of elapsed time. We graph results for two configurations: 8 threads at different priorities from 1…8, and 4 threads initially at priorities 1…4, but inverting to priorities 4…1 midway through the test. For these tests, the ideal result would be to have throughput "fan out" keeping the gaps between adjacent threads the same and keeping all threads in priority order. In the 4-thread cases, after the midpoint, the change in priorities should ideally make the curves "fan in" and meet at a single point at the end of the test run.

Figure 4.11 shows results for the effectiveness of our prioritization adaptations on the IntSet benchmark. Figure 4.12 shows selected results for prioritized contention managers with other benchmarks.

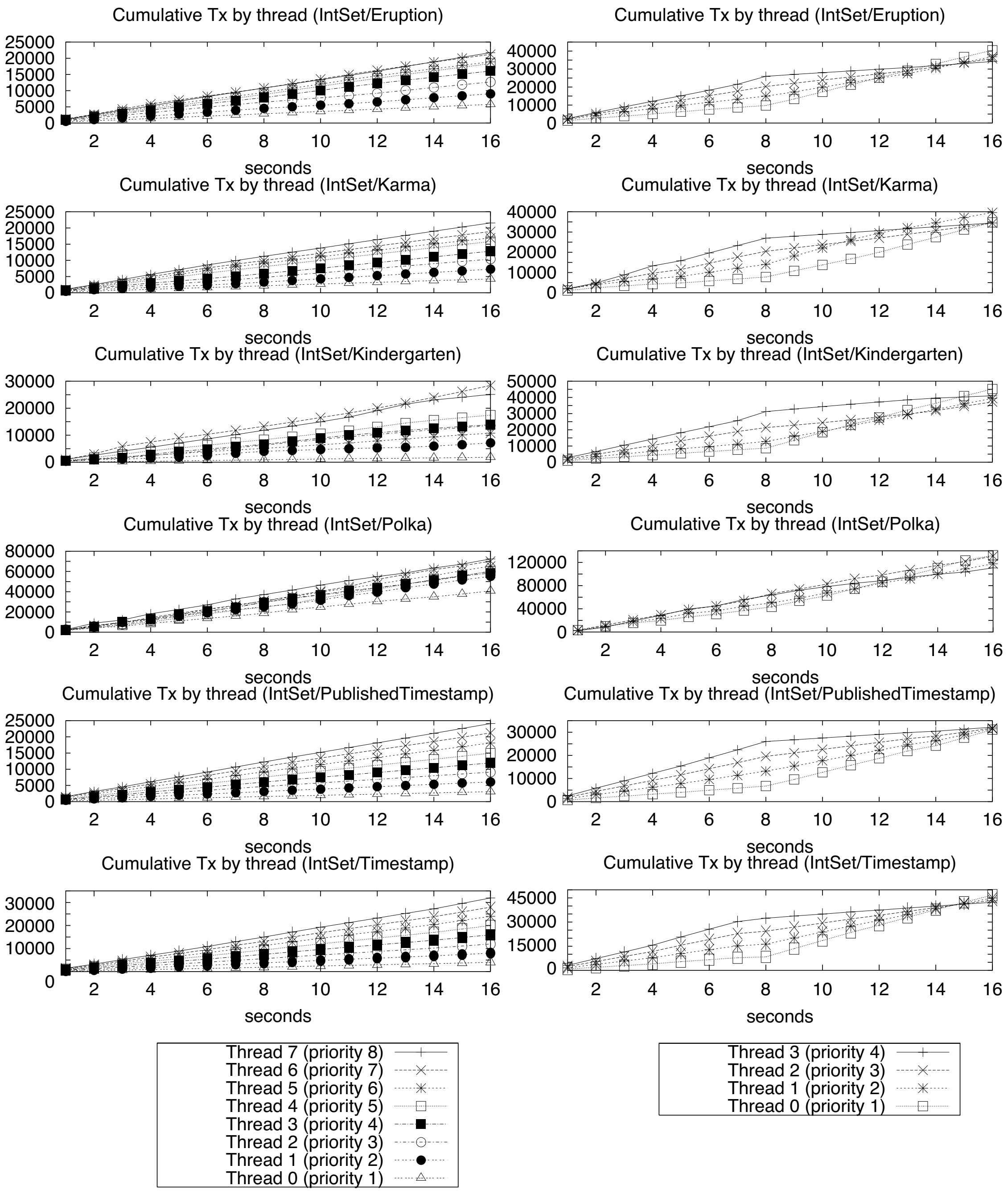


Figure 4.11: Prioritization of the IntSet benchmark: Thread throughput by time. Left: 8 threads with priorities 1…8. Right: 4 threads with priorities 1…4, inverted to 4…1 halfway through.

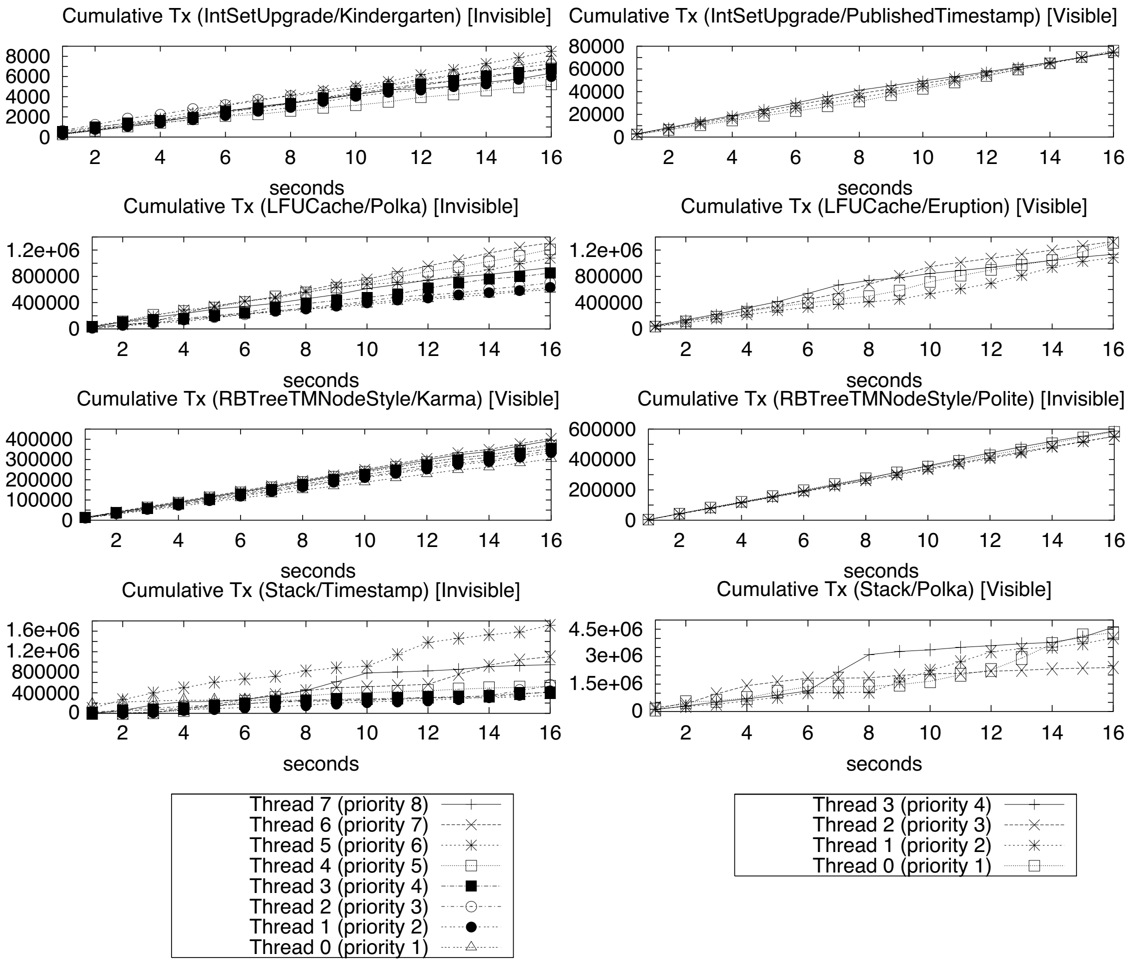


Figure 4.12: Prioritized contention management: other benchmarks. Left: 8 threads with priorities 1…8. Right: 4 threads with priorities 1…4, inverted to 4…1 halfway through.

Discussion

Examining Figure 4.11, we see that the absolute prioritization of the Timestamp protocols yields almost perfect fairness. Eruption, Karma, and Kindergarten, however, only effect their priorities when they directly interact with other threads; consequently, they are dependent on the sequence of transactions that they happen to encounter. Yet all of these managers do very well at prioritization in comparison to Polka. Ironically, the same ability to separate transactions temporally that gives Polka its strong performance on many of the benchmarks limits the extent to which transactions encounter (and thus can influence) each other. This manifests as smaller spread but much higher throughput with Polka: note the difference in Y-axis scale.

We present selected prioritization results in Figure 4.12. A fundamental limitation to our techniques for prioritization is that by relying on contention management to effect priorities, we have no ability to discriminate between transactions that do not encounter each other. Hence, it makes sense that the results for IntSetUpgrade and RBTree (which have an inherently higher level of concurrency than the IntSet benchmark) do not show the same spreading of throughput, though individual thread priority trends are somewhat apparent. The behavior of the Polite manager with RBTree is typical of reasonably fair, but unprioritized managers. We speculate that using priorities to control when a transaction may begin its attempt might improve priority adherence.

For the LFUCache and Stack benchmarks, individual transactions can be so short that lower-priority transactions may complete in the time it takes for a higher-priority transaction to notice and decide to abort them. This tendency manifests as a large deviation from the desired priorities.

4.4.6 Randomized Contention Management

Many researchers have found randomization to be a powerful technique for breaking up repetitive patterns of pathological behaviors that hinder performance [MoR95]. We evaluate this potential by randomizing facets of the Karma manager from Section 4.4.2; we chose Karma for this study because it frequently yields strong performance and because it has multiple facets that can be randomized.

Randomized Abortion

In response to a `shouldAbort` query, the basic Karma manager returns `true` when the difference Δ between the current and enemy transactions' accumulated priorities is less than the number of times the current transaction has attempted to open an object. We randomize this abortion decision with a sigmoid function that returns `true` with probability biased to the higher-priority transaction: $(1 + e^{-\frac{1}{2}\Delta})^{-1}$.

Randomized Backoff

The original Karma scheme backs off for a fixed period of time T between attempts to acquire an object. Randomized, we instead sleep for a uniform random amount of time between 0 and $2T$.

Randomized Gain

The basic Karma manager gains one point of priority with each object that it successfully opens. Randomized, we instead gain as priority an integer randomly selected from the uniform interval $0 \ldots 200$.

Experimental Results

For these tests, we implemented all eight combinations of randomizing three facets of the Karma manager. We ran each variant for a total of 10 seconds. Tests in this section were run with Sun's Java SE 5.0 HotSpot VM. We display throughput results for eight threads: Previous experiments suggest that eight threads is enough for inter-thread contention to affect scalability in the benchmarks, yet few enough that limited scalability of the benchmarks themselves does not skew the results. Figure 4.13 displays throughput results for the various benchmarks.

Discussion

As we see from Figure 4.13, some combination of randomization improves throughput in every benchmark. In the ArrayCounter, IntSetUpgrade, and IntSet benchmarks, randomizing just abortion decisions yields the best performance. Randomizing both

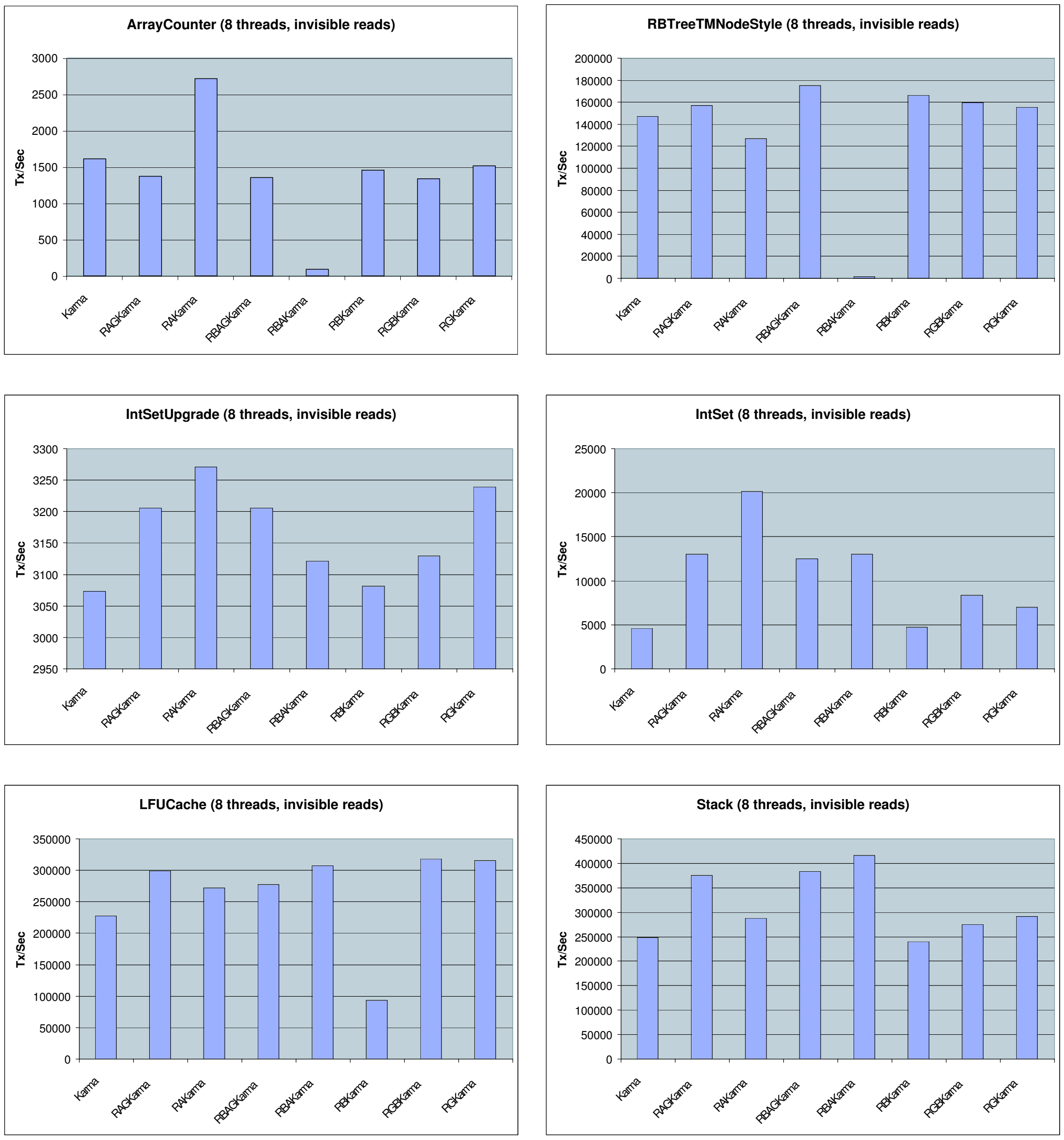


Figure 4.13: Randomization of the “Karma” contention manager: Throughput results for 8 threads and each combination of randomizing backoff (B), abortion (A) decisions, and/or gain (G) upon opening an object (ordered alphabetically)

abortion and backoff gives very poor performance in ArrayCounter and RBTree; yet, it improves performance for LFUCache and Stack. Randomizing gain improves performance both alone, and in every combination with other types of randomization, for LFUCache and RBTree.

Randomizing abortion is particularly helpful for the ArrayCounter, IntSet, and IntSetUpgrade benchmarks. Why is this the case? One possible explanation is that randomizing abortion decisions is very powerful for breaking up semi-deterministic livelock patterns. Such livelocking patterns are particularly visible in ArrayCounter: an increment and a decrement that start at roughly the same time are very likely to have similar or identical priorities when they meet; they are thus prone to mutual abortion.

The combination of randomizing backoff and abortion produces great variance in how long a thread waits to abort an enemy transaction. In times when this wait period is shortened, a longer enemy transaction will have less of a chance to complete; transactions in RBTree and ArrayCounter are particularly long. In times when this wait period is lengthened, multiple shorter enemy transactions can complete, competing with one fewer enemy. Indeed, LFUCache and Stack transactions are very short; and with higher thread counts (not shown due to space limitations), this combination is less effective.

There is no obvious analogous deterministic pathology associated with transaction priority levels. While backoff randomization helps in locking algorithms that have multiple contenders (which can get into simultaneous retry pathology), this problem does not arise in the 2-transaction case. Instead, one continues oblivious to the conflict and the other backs off. This is why randomizing backoff yields comparatively little direct benefit.

4.5 Conclusions

The dynamic software transactional memory (DSTM) system of Herlihy et al. is a Java-based system that supports relatively straightforward programming of a wide variety of dynamic-sized data structures. An attractive feature of DSTM is the ability for one transaction to detect conflict with another and decide whether to proceed immediately or to pause briefly that the other transaction might complete. We implemented a modular interface that allows these policy decisions to be made by modular contention managers that can be "plugged in" without affecting the transaction code or its correctness.

We have further experimented with a wide variety of contention management policies embodied in contention managers. Testing with a variety of benchmarks and two different implementations of read-only transactional access to objects, we have found that different contention management policies work better for different benchmark applications, and that the performance they yield differs by at least a factor of ten. This clearly demonstrates the importance of contention management.

We have examined in greater detail several managers that we found to be frequent top performers, and leveraged this analysis to create a single default contention management scheme, Polka, that gives top or near-top performance across a wide variety of benchmarks. We have also introduced the study of fairness and demonstrated simple techniques that enable proportional-share variants of previous contention managers.

From our analysis of the behavior of contention managers with various benchmarks, we conclude that visible reads yield a large performance benefit due to the lower cost of read-object validation when write contention is low. With sufficiently high write contention, however, the writes to shared memory cache lines needed to add a transaction to a object's readers list degrade performance by introducing memory interconnect contention, and invisible reads become desirable.

5 Contributions and Future Work

Parallelism and synchronization, once the domain of high-end scientific and systems programming, have moved into mainstream computing. Having hit the power wall, processor manufacturers are moving to multiple cores per processor and multiple processors per machine in order to slake users' thirst for faster and faster hardware. This change in runtime environments forces a corresponding change in user-level software: High-performance synchronization is critical to software performance. Where scientific computing, dedicated servers, and operating systems could generally count on preemption being limited or nonexistent, users typically run multiple programs concurrently. Preemption tolerance is thus far more necessary in user-level concurrent software systems than in domains where synchronization has been used historically. At the same time, however, advanced synchronization techniques have historically been too difficult for most programmers to use.

This dissertation explored a variety of techniques to improve the degree of preemption tolerance and ease of use of traditional lock-based systems. These contributions can be broadly divided into three categories:

(I) Improving the preemption tolerance of lock-based software systems by developing high-performance mutual exclusion algorithms that are less sensitive to preemption than traditional algorithms.

(II) Extending the theory and practice of nonblocking algorithms to include algorithms that require condition synchronization.

(III) Improving the performance and general utility of transactional memory systems by introducing explicit, orthogonal contention management that can be adapted to the current runtime environment without requiring modifications to the underlying code for transactions or the transaction system.

5.1 Contributions in Preemption Tolerant Locks

Key conceptual contributions of this dissertation in the area of preemption tolerance in lock-based systems, described in Chapter 2, include:

- Queue-based mutual exclusion algorithms based on the MCS and CLH locks that support timeout, and a queue-based mutual exclusion algorithm based on the MCS lock that supports nonblocking timeout.
- A time-publishing heuristic for effectively estimating whether another thread is currently preempted and that requires no special hardware or operating system support.
- Queue-based mutual exclusion algorithms based on the MCS and CLH locks that use this heuristic to effect strong levels of preemption tolerance.

To evaluate these conceptual contributions, we developed concrete implementations of each lock algorithm on multiple hardware platforms. These implementations are publicly available online from `http://www.cs.rochester.edu/u/scott/synchronization`. To assess their performance, we additionally developed a parameterized test harness that simulates real systems' lock usage.

Before this work, locking was divided into two camps: TATAS locks that do not scale beyond 16-20 processors; and queue-based locks that cannot tolerate preemption and did not support timeout. This work was the first to demonstrate scalable try locks that support timeout on large machines, especially in the presence of preemption.

5.2 Contributions in Nonblocking Synchronization

Key conceptual contributions of this dissertation in the area of nonblocking synchronization, described in Chapter 3, include:

- A design methodology, dual data structures, that supports concurrent objects with condition synchronization via standard linearizability theory.
- Lock-free dual stack and dual queue implementations that outperform the best previously known algorithms under conditions of heavy consumption coupled to limited production.
- Lock-free exchangers and synchronous queues that build on our lock-free dual stack and dual queue implementations to provide high-performance implementations of standard concurrency constructs.

To evaluate these conceptual contributions, we developed concrete implementations of each algorithm on multiple hardware platforms. The dual stack and dual queue are publicly available online from `http://www.cs.rochester.edu/u/scott/synchronization`; the exchanger and synchronous queues have been adopted as part of the Java 6 concurrency library. We also developed benchmark applications to evaluate the performance of these algorithms.

Before this work, there was no meaningful characterization of nonblocking concurrent objects with partial methods. The best-performing algorithms known for synchronous queues and exchange channels achieved little parallel speedup; our nonblocking algorithms outperform them by factors of 14 and 50, respectively, at only moderate levels of concurrency.

5.3 Contributions in Transactional Memory

Key conceptual contributions of this dissertation in the area of transactional memory, described in Chapter 4, include:

- A library of contention managers that demonstrate the breadth of the design space for contention management policies, which allow conflict resolution in nonblocking algorithms to be tailored to the current runtime environment. Of particular note are:

  Karma, which introduces a mechanism that couples runtime priority to the amount of energy (time) the system has invested in a transaction.

  Eruption, which prioritizes operations that hold resources that are bottlenecks in the critical path across many transactions.

  Kindergarten, which forces strict turn-taking between threads that execute transactions that require common resources.

  Polka, which adds exponential backoff to “Karma” and yields very good performance in practical systems.

  PublishedTimestamp, which uses our preemption-detection heuristic to effect oldest-active-operation-first prioritization for conflict resolution.

- A case study on randomization in the “Karma” policy.
- A demonstration of proportional-share prioritization in contention management.

To evaluate these conceptual contributions, we extended the dynamic software transactional memory (DSTM) system of Herlihy et al.[HLM03b] with a modular plug-in interface for contention management. We also produced concrete realizations of each policy and a test harness and a series of microbenchmarks by which to evaluate their performance. Attesting to the importance of conflict resolution, the difference in performance between top-tier and lesser contention management policies exceeds an order of magnitude in every benchmark we have examined. Since this work, modular contention managers have been used in several STM systems [MSS05; HMS06; SMD05; GHP05].

Before this work, research attention was focused on correctness and performance of transactional memory systems; contention management was at best an afterthought in them.

5.4 Future Research Directions

The work conducted for this dissertation suggests many opportunities for further research in several directions. We highlight several such opportunities in the remainder of this Section.

5.4.1 Expanding Algorithmic Techniques

Our work in dual data structures has produced several new algorithms, but highly scalable versions of many more data structures could be useful in practice. In particular, semaphores, skip lists, hash tables and tree-based priority queues all seem to be strong candidates for adaptation to the dual data structure modality. Similarly, the elimination technique seems particularly well-suited for use in synchronous queues.

5.4.2 New Algorithms for New Hardware

Hardware designers are increasingly adopting simultaneous multithreading (SMT) and chip multiprocessing (CMP) to improve performance of hardware systems. However, systems with multiple such chips have highly non-uniform communication overhead between pairs of execution contexts. Where in previous hardware environments a factor of five difference in communication time might have been observed, SMT and CMP systems can easily have an order of magnitude greater differential. This in turn provides an opportunity to create new algorithms that carefully restrict communication dependencies so as to minimize the total communication time involved in executing operations. Particularly in synchronization algorithms, where communication time is the main overhead for a well-tuned implementation, this opens a whole new field for exploration.

5.4.3 Hardware/Software Interaction

The vast majority of research in algorithms published in the last 40 years has been restricted to algorithms implementable in current hardware environments, with existing primitives. Yet, virtually infinite possibilities exist for what manner of primitives can be supported in hardware. Exploring the improvements in algorithm efficiency and complexity that could be obtained if, for example, a double-`compare_and_swap` (DCAS) primitive were available seems a fertile area for additional investigation. For a queue-based lock with timeout, for example, DCAS could potentially reduce the complexity of timeout to just a few lines of code. Similarly, in the DSTM system from Section 4.3, several candidate optimizations are possible given a high-performance DCAS operation:

- Reuse of committed Locators and theft of Locators from active transactions in order to reduce memory churn and allocation overhead.
- Joining a list of visible readers for a transactional object without needing to allocate a new Locator.
- Simplified early release from the list of visible readers for a transactional object.
- Parallel acquisition of transactional objects in lazy acquire mode.

- Parallel abortion of visible readers for a transactional object when acquiring it for read/write access.

5.5 Concluding Remarks

Over about the last 40 years, concurrency has evolved from a protective mechanism in the design of device drivers in operating systems, to an academic curiosity, to a mechanism for supporting multitasking and multiprogramming in operating systems, to a tool for obtaining parallel speedup in server applications and scientific computing. It is now in the process of becoming fundamental to everyday end-user software. This evolution is rife with challenges and opportunities to design high-performance concurrent algorithms. Although software is increasingly concurrent, most software today relies strictly on traditional locks for synchronization, and years of programmer effort have been devoted to fine-tuning intricate systems of locks. Although advanced synchronization techniques are notoriously difficult for most programmers, the algorithms and systems presented in this dissertation provide a toolbox of “off-the-shelf” components that can be readily used. Nonetheless, producing the next generation of high-performance concurrent systems remains an open challenge.

Bibliography

[ADT95] Yehuda Afek, D. Dauber, and Dan Touitou. Wait-Free Made Fast. In *Proceedings of the Twenty-Seventh ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[AMT97] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. Disentangling Multi-Object Operations. In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing*, pages 111–120., August 1997.

[ADF00] Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele, Jr. DCAS-Based Concurrent Deques. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146, Bar Harbor, Maine, United States, 2000.

[ADG99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knioppel, Y. S. Ramakrishna, and Derek White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization. In *OOPSLA '99 Conference Proceedings*, pages 207–222, Denver, CO, November 1999. In *ACM SIGPLAN Notices 34*:10 (October 1999).

[ABH00] William Aiello, Costas Busch, Maurice Herlihy, Marios Mavronicolas, Nir Shavit, and Dan Touitou. Supporting Increment and Decrement Operations in Balancing Networks. In *Chicago Journal of Theoretical Computer Science*, December 2000. Originally presented at the Sixteenth International Symposium on

Theoretical Aspects of Computer Science, Trier, Germany, March 1999, and published in *Lecture Notes in Computer Science*, Vol. 1563, pp. 393-403.

[ACS99] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan J. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Static Analysis Symposium*, pages 19–38, 1999.

[AlF92] J. Alemany and E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, Vancouver, BC, Canada, August 1992.

[AnK01] James Anderson and Yong-Jik Kim. Shared-Memory Mutual Exclusion: Major Research Trends Since 1986. June 2001. submitted for publication.

[And90] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[ABL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[AnM95a] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. In *Proceedings of the Ninth International Workshop on Distributed Algorithms*, pages 168–182, Mont Saint-Michel, France, September, 1995. Lecture Notes in Computer Science #972, Springer-Verlag.

[AnM95b] James H. Anderson and Mark Moir. Universal Constructions for Multi-Object Operations. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 184–193, Ottawa, Canada, August 1995.

[ARJ97] James Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing Wait-Free Objects on Priority-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing*, pages 229–238, August 1997.

[AnM97] James Anderson and Mark Moir. Using Local-Spin *k*-Exclusion Algorithms to Improve Wait-Free Object Implementations. *Distributed Computing*, 11(1):1–20, December 1997.

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, CA, 1991.

[ABD92] J. Arnold, D. Buell, and E. Davis. SPLASH II. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June – July 1992.

[AtD01] Hagit Attiya and Eyal Dagan. Improved Implementations of Binary Universal Operations. *Journal of the ACM*, 48(5):1013–1037, 2001.

[BBN87] BBN Advanced Computers Inc. Chrysalis Programmers Manual. Version 3.0, Cambridge, MA, April 1987.

[BBN86] BBN Laboratories. Butterfly Parallel Processor Overview. BBN Report #6148, Version 1, Cambridge, MA, March 1986.

[BKM98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 258–268, Montréal, PQ, Canada, June 1998. In *ACM SIGPLAN Notices 33*:5 (May 1998).

[Bar93] Greg Barnes. A Method for Implementing Lock-Free Shared Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, Velen, Germany, June – July 1993.

[Ber80] A. J. Bernstein. Output Guards and Nondeterminism in 'Communicating Sequential Processes'. *ACM Transactions on Programming Languages and Systems*, 2(2):234–238, April 1980.

[BeG80] Philip A. Bernstein and Nathan Goodman. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *Proceedings of the Sixth VLDB*, pages 285–300, Montreal, Canada, October 1980.

[Bla90] D. L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *Computer*, 23(5):35–43, May 1990.

[BoH99] Jeff Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *OOPSLA '99 Conference Proceedings*, pages 35–46, Denver, CO, November 1999. In *ACM SIGPLAN Notices 34*:10 (October 1999).

[BLS96] P. Bohannon, D. Lieuwen, and A. Silberschatz. Recovering Scalable Spin Locks. Technical Report 112580-960626-04, Bell Laboratories, June 1996.

[Bur81] J. E. Burns. Symmetry in Systems of Asynchronous Processes. In *Proceedings of the 1981 Symposium on Foundations of Computer Science*, pages 179–184, 1981.

[ChM84] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[ChM88] A. Chang and M. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988. Originally presented at the *Eleventh ACM Symposium on Operating Systems Principles*, November 1987.

[CoV86] R. Cole and U. Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 70(1):32–53, July 1986.

[Cra93a] Travis S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap. TR 93-02-02, Department of Computer Science, University of Washington, February 1993.

[Cra93b] Travis S. Craig. Queuing Spin Lock Algorithms to Support Timing Predictability. In *Proceedings of the Fourteenth IEEE Real-Time Systems Symposium*, pages 148–157, Raleigh-Durham, NC, December 1993.

[DFG00] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. Even Better DCAS-Based Concurrent Deques. In *Proceedings of the Fourteenth International Symposium on Distributed Computing on Distributed Computing*, pages 59–73, 2000.

[Dic01] D. Dice. Implementing Fast Java Monitors with Relaxed Locks. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 79–90, Monterey, CA, April 2001.

[Dij65] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, September 1965.

[Dij72] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. In C. A. R. Hoare and R. H. Perrott, editor, *Operating Systems Techniques*, number 9 in A. P. I. C. Studies in Data Processing, pages 72–93. Academic Press, London, 1972. Also *Acta Informatica* 1 (1971), pp 115-138.

[DiR99] Pedro C. Diniz and Martin C. Rinard. Synchronization Transformations for Parallel Computing. *Concurrency — Practice and Experience*, 11(13):773–802, 1999.

[DDG04] Simon Doherty, David L. Detlefs, Lindsay Grove, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. DCAS Is Not a Silver Bullet for Nonblocking Algorithm Design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 216–224, Barcelona, Spain, June 2004.

[ELS88] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, Pittsburgh, PA, September 1988.

[FKR00] Robert P. Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An Optimizing Compiler for Java. *Software — Practice and Experience*, 30(3):199–232, 2000.

[FRK02] H. Franke, R. Russel, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, June 2002.

[Fra04] Keir Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, February 2004.

[Fu97] Shiwa S. Fu and Nian-Feng Tzeng. A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):628–639, 1997.

[GRS00] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field Analysis: Getting Useful and Low-cost Interprocedural Information. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 334–344, Vancouver, BC, Canada, June 2000.

[GVW89] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, Boston, MA, April 1989.

[GrT90] G. Graunke and S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *Computer*, 23(6):60–69, June 1990.

[GHP05] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic Contention Management in SXM. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[Han97] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, Menlo Park, CA, 1997.

[Har01] Timothy L. Harris. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the Fifteenth International Symposium on Distributed Computing*, pages 300–314, Lisboa, Portugal, October 2001.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-Word Compare-And-Swap Operation. In *Proceedings of the Sixteenth International Symposium on Distributed Computing*, 2002.

[HaF03] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA 2003 Conference Proceedings*, Anaheim, CA, October 2003.

[HS05] Bijun He, William N. Scherer III, and Michael L. Scott. Preemption Adaptivity in Time-Published Queue-Based Spin locks. In *Proceedings of the 2005 International Conference on High Performance Computing*, Goa, India, December 2005. Earlier but expanded version available as TR 867, URCS Department, University of Rochester, May 2005.

[HHL05] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Nir Shavit, and William N. Scherer, III. A Lazy Concurrent List-based Set Algorithm. In *Ninth International Conference on Principles of Distributed Systems*, Pisa, Italy, December 2005.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-Free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 206–215, Barcelona, Spain, June 2004.

[HMS06] Christopher Heriot, Virendra Marathe, Michael F. Spear, Athul Acharya, Sandhya Dwarkadas, David Eisenstat, William N. Scherer III, Michael L. Scott, and Arrvindh Shriraman. Low-Overhead Software Transactional Memory for C++. Technical Report, Department of Computer Science, University of Rochester, January 2006. In preparation.

[HLM02] Maurice Herlihy, Victor Luchangco, and and Mark Moir. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *Proceedings of the Sixteenth International Symposium on Distributed Computing*, Toulouse, France, October 2002.

[HLM03a] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems*, Providence, RI, May, 2003.

[HLM03b] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

[Her90] M. Herlihy. 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, Seattle, WA, March 1990.

[HeW90] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Her91] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[Her93] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[HeM93] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, December 1992.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hol97] Gerald J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[HuS98] T.-L. Huang and C. H. Shann. A comment on 'A Circular List-based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors'. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):414–415, 1998.

[Hua99] Ting-Lu Huang. Fast and Fair Mutual Exclusion for Shared Memory Systems. In *Proceedings of the 1999 International Conference on Distributed Computing Systems*, pages 224–231, Los Alamitos, CA, May/June 1999.

[IBM01] IBM Corporation. AIX 5L Differences Guide, Version 5.0. 2001.

[IsR94] Amos Israeli and Lihu Rappoport. Disjoint-Access Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 151–160, Los Angeles, CA, August 1994.

[Jay03] Prasad Jayanti. Adaptive and Efficient Abortable Mutual Exclusion. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 295–304, Boston, MA, July 2003.

[JoH97] T. Johnson and K. Harathi. A Prioritized Multiprocessor Spin Lock. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):926–933, 1997.

[KBG97] Alain Kägi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the Twenty-Fourth International Symposium on Computer Architecture*, Denver, CO, June 1997.

[KLM91] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 41–55, Pacific Grove, CA, October 1991.

[Ken92] Kendall Square Research. KSR1 Principles of Operation. Waltham MA, 1992.

[Kni86] Thomas F. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 105–112, Cambridge, MA, August 1986.

[Knu66] D. E. Knuth. Additional Comments on a Problem in Concurrent Programming Control. *Communications of the ACM*, 9(5):321–322, May 1966.

[KWS97] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-Conscious Synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[KSU93] O. Krieger, M. Stumm, and R. Unrau. A Fair Fast Scalable Reader-Writer Lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:201–204, St. Charles, IL, August 1993.

[KJC99] Sanjeev Kumar, Dongming Jiang, Rohit Chandra, and Jaswinder Pal Singh. Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 23–34, Atlanta, GA, May 1999.

[LaS04] Edya Ladan-Mozes and Nir Shavit. An Optimistic Approach to Lock-Free FIFO Queues. In *Proceedings of the Eighteenth International Symposium on Distributed Computing*, Amsterdam, The Netherlands, October 2004.

[Lam74] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[Lam87] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[Lea05] Doug Lea. The java.util.concurrent Synchronizer Framework. *Science of Computer Programming*, 58(3):293–309, December 2005.

[Lea] Doug Lea. Concurrency JSR-166 Interest Site. http://gee.cs.oswego.edu/dl/concurrency-interest/.

[LLG90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

[LLG92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[LiA94] Beng-Hong Lim and Anant Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, San Jose, CA, October 1994.

[Luc02] Victor Luchangco. *Personal communication*. Sun Microsystems Laboratories, Boston, MA, January, 2002.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[MLH94] Peter Magnussen, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 165–171, Cancun, Mexico, April 1994. Expanded version available as “Efficient Software Synchronization on Large Cache Coherent Multiprocessors”, SICS Research Report T94:07, Swedish Institute of Computer Science, February 1994.

[MSS04] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Houston, TX, October 2004.

[MSS05] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[Mar91] Evangelos P. Markatos. Multiprocessor Synchronization Primitives with Priorities. In *Proceedings of the Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 1–7, Atlanta, GA, May 1991. In conjunction with the *Seventeenth IFAC/IFIP Workshop on Real-Time Programming*, and published in the *Newsletter of the IEEE Computer Society Technical Committee on Real-Time Systems 7*:4 (Fall 1991).

[MSL91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, October 1991.

[MaT02] José F. Martínez and Josep Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, San Jose, CA, October 2002.

[MAK01] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russel, D. Sarma, and M. Soni. Read-Copy Update. In *Proceedings of the Ottawa Linux Symposium*, July 2001.

[MSA04] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proceedings of the Twentieth International Conference on Data Engineering*, Boston, MA, March/April 2004.

[MeS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[Mic02] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, Winnipeg, MB, Canada, August 2002.

[Mic03] Maged M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. *Proceedings of the Ninth European Conference on Parallel Processing (EURO-PAR)*, 2790:651–660, Springer-Verlag, August 2003.

[Mic04] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), August 2004.

[MiS96] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, May 1996.

[MiS98] Maged M. Michael and Michael L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.

[Moi00] Mark Moir. Laziness Pays! Using Lazy Synchronization Mechanisms to Improve Non-Blocking Constructions. In *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing*, Portland, OR, July 2000.

[MNS05] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *Proceedings of the*

Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pages 253–262, Las Vegas, NV, July 2005.

[Moi97] Mark Moir. Transparent Support for Wait-Free Transactions. In *Proceedings of the Eleventh International Workshop on Distributed Algorithms*, 1997.

[MoR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, August 1995.

[OpL02] Jeffrey Oplinger and Monica S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 184–196, San Jose, CA, October 2002.

[OrO00] Vitaly Oratovsky and Michael O'Donnell. *Personal communication*. Mercury Computer Corp., February 2000.

[Ous82] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22–30, Miami/Ft. Lauderdale, FL, October 1982.

[P1590] P1596 Working Group of the IEEE Computer Society Microprocessor Standards Committee. SCI (Scalable Coherent Interface): An Overview of Extended Cache-Coherence Protocols. Draft 0.59 P1596/Part III-D, February 1990.

[Pap79] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[Pet81] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[PfN85] G. F. Pfister and V. Alan Norton. 'Hot Spot' Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[PLJ94] Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers*, 43(5):548–559, May 1994.

[Pug90] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[RaH03] Zoran Radović and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proc of the Ninth International Symposium on High Performance Computer Architecture*, pages 241–252, Anaheim, CA, February 2003.

[RaG01] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the Thirty-Fourth International Symposium on Microarchitecture*, Austin, TX, December 2001.

[RaG02] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, San Jose, CA, October 2002.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion*, MIT Press Series in Scientific Computation. MIT Press, Cambridge, MA, 1986. Translated from the French by D. Beeson.

[RoD90] John T. Robinson and Murthy V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 134–142, Boulder, CO, May 1990.

[Ruf00] Erik Ruf. Effective Synchronization Removal for Java. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 208–218, Vancouver, BC, Canada, June 2000.

[ScS04a] William N. Scherer III and Michael L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.

[ScS04b] William N. Scherer III and Michael L. Scott. Nonblocking Concurrent Objects with Condition Synchronization. In *Proceedings of the Eighteenth International Symposium on Distributed Computing*, Amsterdam, The Netherlands, October 2004.

[ScS05] William N. Scherer III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[ScS01] Michael L. Scott and William N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. In *Proceedings of the Eighth ACM Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.

[Sco02] Michael L. Scott. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 31–40, Monterey, CA, July 2002.

[ShS03] Ori Shalev and Nir Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 102–111, Boston, Massachusetts, 2003.

[SHC00] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A Practical Nonblocking Queue Algorithm Using Compare-and-Swap. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, page 470ff, Iwate, Japan, July 2000.

[ShT95] Nir Shavit and Dan Touitou. Elimination Trees and the Construction of Pools and Stacks. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995.

[ShT97] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, February 1997. Originally presented at the *Fourteenth ACM Symposium on Principles of Distributed Computing*, August 1995.

[ShZ99] Nir Shavit and Asaph Zemach. Scalable Concurrent Priority Queue Algorithms. In *Proceedings of the Eighteenth ACM Symposium on Principles of Distributed Computing*, pages 113–122, Atlanta, Georgia, United States, 1999.

[SMD05] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear. Hardware Acceleration of Software Transactional Memory. TR 887, Department of Computer Science, University of Rochester, December 2005. Condensed version submitted for publication.

[Sto92] J. M. Stone. A Non-Blocking Compare-and-Swap Algorithm for a Shared Circular Queue. In S. Txafestas and others, editors, *Parallel and Distributed Computing in Engineering Systems*, pages 147–152. Elsevier Science Publishers, 1992.

[SSH93] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.

[Sto84] Michael Stonebraker. Virtual Memory Transaction Management. *ACM SIGOPS Operating Systems Review*, 18(2):8–16, 1984.

[SuT02] H. Sundell and P. Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In *Proceedings of the Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington, DC, March 2002.

Also TR 2002-02, Chalmers University of Technology and Göteborg University, Göteborg, Sweden.

[SuT03] Hakan Sundell and Philippas Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In *Proceedings of the 2003 International Parallel and Distributed Processing Symposium*, page 84ff, Nice, France, April 2003. Extended version available as Technical Report 2003-01, Chalmers University of Technology and Göteborg University, Göteborg, Sweden.

[TaS97] H. Takada and K. Sakamura. A Novel Approach to Multiprogrammed Multiprocessor Synchronization for Real-Time Kernels. In *Proceedings of the Eighteenth IEEE Real-Time Systems Symposium*, pages 134–143, San Francisco, CA, December 1997.

[Tre86] R. Kent Treiber. Systems Programming: Coping with Parallelism. RJ 5118, IBM Almaden Research Center, April 1986.

[TsZ01] Philippas Tsigas and Yi Zhang. A Simple, Fast and Scalable Non-Blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143, Aldemar, Crete, Greece, July 2001.

[TLT01] Eric Tune, Dongning Liang, Dean M. Tullsen, and Brad Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 185–196, January 2001.

[TSP92] John Turek, Dennis Shasha, and Sundeep Prakash. Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 212–222, Vancouver, BC, Canada, August 1992.

[Val94] John D. Valois. Implementing Lock-Free Queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, October 1994.

[Val95] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Canada, August 1995.

[WaW94] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[WTS96] Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura. Priority Inheritance Spin Locks for Multiprocessor Real-Time Systems. In *Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 70–76, Bejing, China, June 1996.

[WKS94] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael L. Scott. Scalable Spin Locks for Multiprogrammed Systems. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 583–589, Cancun, Mexico, April 1994.

[WKS95] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. In *Proceedings of the Fifth ACM Symposium on Principles and Practice of Parallel Programming*, pages 199–206, Santa Barbara, CA, July 1995.

[WOT95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the Twenty-Second In-*

ternational Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995.

[ZhN04] Haoqiang Zheng and Jason Nieh. SWAP: A Scheduler With Automatic Process Dependency Detection. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, pages 183–196, San Francisco, CA, March 2004.

A Publications

1. W. N. Scherer III, D. Lea, and M. L. Scott. Scalable Synchronous Queues. In *11th ACM Symposium on Principles and Practice of Parallel Programming* (PPoPP 2006), Manhattan, NY, March 2006.

2. S. Heller, M. Herlihy, V. Luchangco, M. Moir, N. Shavit, and W. N. Scherer III. A Lazy Concurrent List-Based Set Algorithm. In *9th International Conference of Principles of Distributed Systems* (OPODIS 2005), Pisa, Italy, December 2005.

3. W. N. Scherer III, D. Lea, and M. L. Scott. A Scalable Elimination-based Exchange Channel. In *OOPSLA Workshop on Synchronization and Concurrency in Object Oriented Languages* (SCOOL 2005) held in conjunction with the *20th ACM Symp. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA 2005), San Diego, CA, October 2005.

4. V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *18th Annual Conference on DIStributed Computing* (DISC 2005), Cracow, Poland, September, 2005. An earlier version appears as Technical Report URCS-TR868, Department of Computer Science, University of Rochester, May 2005.

5. B. He, W. N. Scherer III, and M. L. Scott. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. In *12th Annual IEEE International Conference on High Performance Computing* (HiPC 2005), Goa, India, December 2005. An earlier version appears as Technical Report URCS-TR867, Department of Computer Science, University of Rochester, May 2005.

6. W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *24th ACM Symposium on Principles of Distributed Computing* (PODC'05), Las Vegas, NV, July 2005.

7. W. N. Scherer III and M. L. Scott. Randomization in STM Contention Management (poster paper). In *24th ACM Symposium on Principles of Distributed Computing* (PODC'05), Las Vegas, NV, July 2005.

8. V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems* (LCR'04), Houston, TX, October 2004.

9. W. N. Scherer III and M. L. Scott. Nonblocking Concurrent Objects with Condition Synchronization. In *18th Annual Conference on DIStributed Computing* (DISC'04), Amsterdam, The Netherlands, October 2004.

10. W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs* held in conjunction with the *23rd ACM Symposium on Principles of Distributed Computing* (PODC'04), St. Johns, NL, Canada, July 2004.

11. M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Supporting Dynamic-Sized Data Structures. In *22nd ACM*

Symposium on Principles of Distributed Computing (PODC'03), Boston, MA, July 2003.

12. M. L. Scott and W. N. Scherer III. Scalable Queue-Based Spin Locks with Time-out. In *8th ACM Symposium on Principles and Practice of Parallel Programming* (PPoPP'01), Snowbird, UT, June 2001.