Comparative Performance of Memory Reclamation Strategies for Lock-free and Concurrently-readable Data Structures

by

Thomas Edward Hart

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Comparative Performance of Memory Reclamation Strategies for Lock-free and Concurrently-readable Data Structures

Thomas Edward Hart

Master of Science

Graduate Department of Computer Science

University of Toronto

2005

Despite their advantages, lock-free algorithms are often not adopted in practice, partly due to the perception that they perform poorly relative to lock-based alternatives in common situations when there is little contention for objects or the CPUs.

We show that memory reclamation can be a dominant performance cost for lock-free algorithms; therefore, choosing the most efficient memory reclamation method is essential to having lock-free algorithms perform well. We compare the costs of three memory reclamation strategies: *quiescent-state-based reclamation*, *epoch-based reclamation*, and *safe memory reclamation*. Our experiments show that changing the workload or execution environment can change which of these schemes is the most efficient. We therefore demonstrate that there is, to date, no panacea for memory reclamation for lock-free algorithms.

Using a common reclamation scheme, we fairly compare lock-free and concurrently-readable hash tables. Our evaluation shows that programmers can choose memory reclamation schemes mostly independently of the target algorithm.

Acknowledgements

First, I'd like to thank my supervisor, Angela Demke-Brown, for helping me shape my vague ideas into coherent research, and helping me to take a step back and see which questions are important and which are not.

Second, I want to thank Paul McKenney, with whom I have enjoyed a very helpful collaboration during the course of this project. Having such an experienced colleage has been immensely helpful as I start my research career.

I wish to thank Maged Michael and Keir Fraser for so readily answering questions about their respective work.

I also want to thank Vassos Hadlizacos for introducing me to practical lock-free synchronization through Greenwald's work on Cache Kernel, and my colleagues in the Systems Software Reading Group for introducing me to Read-Copy Update. Without these colleagues, I never would have become aware of this research topic.

During the course of this project, I solicited advice from many, many colleagues here at the University of Toronto, all of whom have been very generous in their help. In particular, I would like to thank Faith Fich, Alex Brodsky, Cristiana Amza, Reza Azimi, David Tam, Marc Berndl, Mathew Zaleski, Jing Su, and Gerard Baron.

Finally, I would like to thank my lab's system administrator, Norman Wilson, for keeping the machines I needed for my experiments running smoothly, and for forgiving me when I occasionally crashed them.

Contents

1 Introduction **1**

1.1 Problems with Locking 1

1.2 Memory Reclamation 3

1.3 Contributions 4

1.4 Organization of Thesis 5

2 Fundamentals **7**

2.1 Terminology 7

2.1.1 Threads 8

2.1.2 Shared Objects 8

2.1.3 Hardware Operations 12

2.2 Memory Consistency Models 15

3 Lock-free and Concurrently-readable Algorithms **17**

3.1 Theory of Non-blocking Synchronization 17

3.2 Practical Non-blocking Algorithms 18

3.2.1 Higher-level primitives 19

3.2.2 Algorithms using CAS or LL/SC 20

3.3 Concurrently-Readable Algorithms 32

4 Memory Reclamation Schemes **35**

4.1 Descriptions of Schemes 35
4.1.1 Blocking Methods 35
4.1.2 Lock-free Methods 39
4.2 Applying the Schemes 43
4.3 Analytic Comparison of Methods 49

5 Experimental Evaluation **53**
5.1 Experimental Setup 53
5.1.1 Algorithms Compared 53
5.1.2 Test Program 54
5.1.3 Operating Environment 55
5.1.4 Limitations of Experiment 57
5.2 Performance Analysis 57
5.2.1 Effects of Traversal Length 60
5.2.2 Effects of CPU Contention 63
5.2.3 Relative Severity 68
5.2.4 Low Overhead of QSBR 69
5.2.5 Lock-free Versus Concurrently-readable Linked List Algorithms . 71
5.2.6 Summary of Recommendations 75

6 Related Work **79**
6.1 Blocking Memory Reclamation for Non-blocking Algorithms 79
6.2 Vulnerabilities of Blocking Memory Reclamation Schemes 81
6.3 Performance Comparisons 82

7 Conclusions and Future Work **84**

Bibliography **87**

List of Tables

5.1 Characteristics of Machines 56

7.1 Properties of Memory Reclamation Schemes 85

List of Figures

2.1 Hierarchy of properties of concurrent objects, represented as a partial order. Any property in the hierarchy is strictly stronger than all properties below it; for example, *obstruction-free* and *almost non-blocking* are weaker than *lock-free*, but in different ways. 11

2.2 Pseudocode definition of CAS. 12

2.3 Pseudocode definitions of LL and SC. 13

2.4 Pseudocode for implementing CAS using LL/SC. 13

2.5 Illustration of the ABA problem. 14

3.1 The consensus hierarchy. 18

3.2 Pseudocode definition of DCAS. 19

3.3 Example operation of search function for lock-free linked list. 21

3.4 Pseudocode for search function for lock-free list, stripped of memory reclamation code. 22

3.5 Example operation of insert function for lock-free linked list. 24

3.6 Error which can occur in a naïve lock-free linked list implementation when insertions and deletions are interleaved. To prevent such errors, the lock-free linked list must mark a node's *next* pointer before deleting the node (Figure 3.7). 26

3.7 Example operation of delete function for lock-free linked list. 27

3.8 Dequeue from non-empty lock-free queue. 28

3.9 Pseudocode for dequeue function for lock-free queue, stripped of memory reclamation code. 29

3.10 Enqueue to non-empty lock-free queue. 30

3.11 Pseudocode for enqueue function for lock-free queue, stripped of memory reclamation code. 31

3.12 Concurrently-readable node modification example from which *read-copy update* derives its name. 33

3.13 Concurrently-readable insertion. 33

3.14 Concurrently-readable deletion. 34

4.1 Illustration of EBR. Threads follow the global epoch. If a thread observes that all other threads have seen the current epoch, then it may update the global epoch. Hence, if the global epoch is e, threads in critical sections can be in either epoch $e + 1$ or e, but not $e - 1$ (all mod 3). The time period $[t_1, t_2]$ is thus a grace period for thread $T1$. 36

4.2 Illustration of QSBR. Thick lines represent quiescent states. The time interval $[t_1, t_2]$ is a grace period: at time t_2, each thread has passed through a quiescent state since t_1, so all nodes logically removed before time t_1 can be physically deleted. 38

4.3 Illustration of SMR. q is logically removed from the linked list on the right of the diagram, but cannot be physically deleted while $T3$'s hazard pointer $HP[4]$ is still pointing to it. 41

4.4 Lock-free queue's `dequeue()` function, using SMR. 45

4.5 Lock-free queue's `dequeue()` function, using QSBR. 46

4.6 Lock-free queue's `dequeue()` function, using EBR. 47

4.7 Illustration of why QSBR is inherently blocking. Here, thread $T2$ does not go through a quiescent state for a long period of execution time; hence, threads $T1$ and $T3$ must wait to reclaim memory. A similar argument holds for EBR. 51

5.1 High-level pseudocode for the test loop of our program. Each thread executes this loop. The call to QUIESCENT_STATE() is ignored unless we are using QSBR. 55

5.2 Single-threaded memory reclamation costs on PowerPC. Hash table statistics are for a 32-bucket hash table with a load factor of 1. Queue statistics are for a single non-empty queue. 58

5.3 Hash table, 32 buckets, load factor 1, read-only workload. Spinlocks scale poorly as the number of threads increases. 59

5.4 Hash table, 32 buckets, one thread, read-only workload, varying load factor. 60

5.5 Hash table, 32 buckets, one thread, write-only workload, varying load factor. 61

5.6 Hash table, 32 buckets, one thread, write-only workload, varying load factor. 62

5.7 100 queues, variable number of threads, Darwin/PPC. 63

5.8 Hash table, 32 buckets, load factor 1, write-only workload, variable number of threads, Darwin/PPC. 64

5.9 100 queues, variable number of threads, Linux/IA-32. 64

5.10 Hash table, 32 buckets, 16 threads, write-only workload, varying load factor. 66

5.11 Hash table, 32 buckets, load factor 10, write-only workload, varying number of threads. 66
5.12 Hash table, 32 buckets, load factor 20, write-only workload, varying number of threads. 67

5.13 Queues, two threads, varying number of queues. 69
5.14 Hash table, 32 buckets, two threads, load factor 5, varying update fraction. 70

5.15 Hash table, 32 buckets, two threads, load factor 5, varying update fraction. QSBR allows the lock-free algorithm to out-perform RCU for almost any workload; neither SMR nor EBR achieve this. 71

5.16 Code for fast searches of lock-free list; compare to pseudocode of Figure 3.4. 72
5.17 Hash table, 32 buckets, two threads, load factor 5, varying update fraction between 0 and 0.1. 72
5.18 Hash table, 32 buckets, two threads, read-only workload, varying load factor. 73
5.19 Hash table, 32 buckets, two threads, write-only workload, varying load factor. 73
5.20 Decision tree for choosing a memory reclamation scheme. 76
5.21 Decision tree for choosing whether to use the lock-free or concurrently-readable linked list algorithm. 78

Chapter 1

Introduction

Shared memory multiprocessing is becoming important outside the traditional areas of high-performance computing such as scientific computation, graphics rendering, and web servers. New technologies such as *simultaneous multithreading (SMT)* and *chip multiprocessing (CMP)* are bringing multiprocessing to commodity desktops. Furthermore, many applications running on these systems, such as web browsers and operating system kernels, are already multithreaded. It is essential that these concurrent applications perform well; we expect that technologies like SMT and CMP will make this requirement even more important in the future.

The rest of this chapter describes the problems associated with locking, explains the need for memory reclamation, summarizes the contributions of this thesis, and outlines the organization of the remainder of this document.

1.1 Problems with Locking

Concurrent applications require a means to coordinate accesses to shared data structures. Mutual exclusion, implemented via locks or semaphores, is the most common solution to the synchronization problem. Mutual exclusion is intuitive; however, it has several drawbacks. Locks, in particular, can be bottlenecks in high-performance shared memory

programs, and suffer from several problems:

- **Poor reliability:** a thread that crashes while holding a lock will make the resource associated with the lock unavailable to all other threads. This is the most cited problem with locks in theoretical literature, since fault tolerance is a favorite problem in the theory of distributed computing; however, few systems researchers consider this the most compelling disadvantage of locks.

- **Poor performance in the face of preemption:** a thread that is pre-empted while holding a lock will make the associated resource unavailable until the thread is rescheduled and can complete its work. This can result in *convoying*, which occurs when a pre-empted or otherwise stalled thread holds a lock, and other threads form a "convoy" waiting for the lock.

- **Priority inversion:** a low priority thread may be pre-empted while holding a lock on a shared resource; a high priority thread requiring the shared resource may then be scheduled, and have to wait for the lower priority thread.

- **Vulnerability to deadlock:** if threads must acquire locks on two or more resources, deadlock is possible if those locks are not acquired in the same order. A thread may even deadlock with itself if it attempts to acquire a lock which it already holds..

In addition to these deficiencies, lock-based approaches require expensive operations, such as compare-and-swap, to acquire and release locks — even when the locks are not contended.

Many researchers are therefore interested in ways to avoid using locks [42, 18, 54, 55]. Using reader-writer locks instead of normal spinlocks increases concurrency by allowing either multiple readers or a single writer at any given time. Using a *concurrently-readable* [34] data structure also permits multiple readers, but in this case they can run concurrently with a single writer, and do not need to acquire a lock. Using a *lock-free* data

structure is more complicated, but allows readers and writers both to run concurrently when logically possible, and sidesteps the above-mentioned disadvantages of locks; however, lock-free data structures typically require expensive atomic operations. Some recent lock-free algorithms, however, require few such expensive operations, and therefore provide an attractive alternative to lock-based designs, as they have been shown to have lower overhead, even when contention for objects and the CPUs is low [47].

1.2 Memory Reclamation

Memory reclamation is required for all dynamic lock-free and concurrently-readable data structures, such as linked lists and queues. We distinguish logical deletion of a node, N (removing it from a shared data structure so that no new references to N may be created) from physical deletion of that node (allowing the memory used for N to be reclaimed for arbitrary reuse). If a thread T_1 logically deletes a node N from a lock-free data structure, it cannot physically delete N until no other thread T_2 holds a reference to N, since physically deleting N may cause T_2 to crash or execute incorrectly. Never physically deleting logically deleted nodes is also unacceptable, since this will eventually lead to out-of-memory errors which will stop threads from making progress.

Choosing an inefficient memory reclamation scheme can ruin the performance of a lock-free or concurrently-readable algorithm. Reference counting [61, 12], for example, has high overhead in the base case, and scales poorly in structures for which long chains of nodes must be traversed [47]. Little work has been done on comparing different memory reclamation strategies; we address this deficiency, showing that the performance of memory reclamation schemes depends both on their base costs, and on the target workload and execution environment. We expect our results can provide some guidance to implementers of lock-free and concurrently-readable algorithms in choosing an appropriate scheme for their specific applications.

Current memory reclamation strategies suggested for lock-free data structures [44, 47, 28, 17, 11] have high per-operation runtime overhead. We believe that imposing large overhead on an algorithm in order to manage memory reclamation is unacceptable. We therefore propose using *quiescent-state-based reclamation* (QSBR) [42, 39, 18, 6, 38, 40], an efficient scheme with negligible overhead pioneered in the domain of operating system kernels, to manage memory for lock-free data structures whenever CPU contention is low. QSBR is normally used with concurrently-readable algorithms; however, we have found that it also works well with lock-free ones. We show by example that QSBR can be implemented in applications other than operating system kernels, and that using this memory reclamation scheme can make lock-free algorithms significantly more efficient. The cost of QSBR's increased performance is an increased burden on the application programmer, and the risk of out-of-memory errors. We therefore believe that, despite the performance advantage of QSBR, these other memory reclamation algorithms have a place in situations where this trade-off is unacceptable.

The other memory reclamation schemes, safe memory reclamation (SMR) and epoch-based reclamation (EBR), are intended for use with lock-free algorithms. We demonstrate that just as QSBR can be used with lock-free schemes, SMR and EBR can be used with concurrently-readable ones, therefore showing that the choice of memory reclamation scheme is mostly independent of the target algorithm.

1.3 Contributions

Although lock-free and concurrently-readable algorithms each have their respective adherents, we are not aware of any work which has examined the tradeoffs between comparable lock-free and concurrently-readable algorithms. Were one to make such a comparison, one might naïvely use each algorithm with the reclamation scheme suggested by its creator, and compare, for example, a lock-free algorithm using SMR to a concurrently-

readable algorithm using QSBR. One might even be unaware that the algorithm and its reclamation scheme *can* be separated. Since the choice of memory reclamation scheme has a profound impact on performance, any such comparison would be unfair. Our decoupling of memory reclamation schemes from the algorithms for which they were originally designed therefore allows us to make the first fair and detailed comparison between lock-free and concurrently-readable chaining hash tables.

In summary, the contributions of this thesis are as follows:

- We demonstrate that the choice of algorithm and memory reclamation scheme are mostly independent.

- We analyze the strengths and weaknesses of each of three memory reclamation strategies — QSBR, SMR, and EBR.

- We make the first comparison of the performance of a lock-free algorithm to a concurrently-readable alternative.

Our experiments were conducted on commodity dual-processor PowerPC and IA-32 machines. Our results show that changing the workload or execution environment can change which memory reclamation scheme is the most efficient; we therefore demonstrate that there is, to date, no panacea for memory reclamation for lock-free and concurrently-readable algorithms.

1.4 Organization of Thesis

This thesis is organized as follows. Chapter 2 discusses the terminology we use and our system model. Chapter 3 provides background on lock-free and concurrently-readable algorithms, with an emphasis on the algorithms on which we performed our experiments. Chapter 4 presents the memory reclamation schemes we consider, as well as some which

we did not. Chapter 5 presents our experimental results, and Chapter 6 discusses related work. Chapter 7 concludes the thesis.

Chapter 2

Fundamentals

In this chapter, we define terminology for discussing lock-free and concurrently-readable algorithms. We then discuss weak memory consistency models, which, we show in chapter 5, have an important effect on the performance of memory reclamation schemes.

2.1 Terminology

This work attempts to use results from the theory of distributed computing, specifically non-blocking synchronization, in a practical setting. As such, we shall use terminology which allows us to discuss non-blocking synchronization in a cogent matter. We shall define our concepts of threads, shared objects and their properties, and the hardware operations required to implement these shared objects.

The level of formality of this terminology is lower than that used in pure theory literature, but higher than that of systems literature; we hope that the terminology will be acceptable to people in both groups.

2.1.1 Threads

The terms *process* and *thread* are, for the purposes of our discussion, synonymous. Both refer to a unit of execution which may access shared objects. Since the additional denotations of process in operating systems literature have no bearing on our analysis of the algorithms under consideration, we prefer the term "thread."

Typically, for the purposes of any theoretical analysis, a thread may *crash*. This means only that the crashed thread ceases to perform any actions, and does not include any of the other denotations of the term "crash" in systems literature. A thread that does not crash is said to be *correct*. We assume that the actions of any thread occur over a sequence of discrete *time steps*.

2.1.2 Shared Objects

A *shared object* is an entity in memory which can be accessed and modified only through a pre-defined set of *operations*. Conceptually, these operations correspond to *methods* in object-oriented programming; for example, the operations on a shared queue are *enqueue* and *dequeue*. Multiple threads may access a shared object. A *dynamic shared object* is a shared object which is made up of *nodes* which are dynamically allocated in memory.

In discussing the various forms of non-blocking synchronization, we shall use terminology which is becoming standard in the literature [17]. A shared object is *non-blocking* if and only if it is *wait-free*, *lock-free*, or *obstruction-free*. A shared object O is wait-free, lock-free, or obstruction-free, respectively, if and only if the following properties hold:

- *wait-free*: if a correct thread is performing an operation on O, this operation must complete after a finite number of time steps.

- *lock-free*: if a correct thread is performing an operation on O, *some* operation invoked by *some* thread on O must complete after a finite number of time steps.

- *obstruction-free*: if a correct thread is performing an operation on O, *some* operation invoked by *some* thread on O must complete after a finite number of time steps *unless* another thread's operation conflicts with it.

Those who are used to the terminology of operating systems may find the following summary of the different non-blocking synchronization properties more helpful. A lock-free shared object guarantees that it will not, under any circumstances, cause any of the threads accessing it to deadlock, even if some of those threads crash. An obstruction-free shared object makes almost the same guarantee, except that it can allow livelock. A wait-free shared object is a lock-free shared object that also guarantees that it will not starve any thread accessing it.

We note that designating wait-freedom, lock-freedom, and obstruction-freedom as the only non-blocking properties is only a matter of convention. Originally, *lock-free* was a synonym for *non-blocking*. The intuition behind the latter term is that threads may block while waiting to access a lock-protected object if another thread holds the object's lock, but threads will never block on accessing a non-blocking object. Obstruction-freedom was later introduced as a non-blocking property, and the literature continues to refer to it as such.

Shared objects may also be *almost non-blocking* or *concurrently-readable*. These properties are similar to the three non-blocking properties, but are not defined as being non-blocking. Roughly, a concurrently-readable shared object allows concurrent reads, but may use locks for updates, and an almost non-blocking shared object allows concurrent reads, but only a bounded number of threads may simultaneously attempt to update the object. In both cases, writers could block on attempting to update the shared object, so it intuitively makes sense not to consider these properties non-blocking.

We owe the term *concurrently-readable* to Lea [34], whose definition we attempt to formalize. A shared object O is *concurrently-readable* if multiple threads may read O concurrently, and any read operation on O begun by a correct thread must complete

after a finite number of time steps. The definition of *concurrently-readable* does not rule out the use of locks for updates; however, a conventional design based on reader-writer locks is not concurrently-readable, since a thread that crashes after write-acquiring the lock will stop other threads from reading the shared object.

By our definition, any wait-free object is concurrently-readable, but a lock-free or obstruction-free object is *not* necessarily concurrently-readable. These non-blocking properties both allow readers to be starved, which violates the definition of concurrently-readable. Furthermore, technically, the definition of obstruction-freedom allows livelock between concurrent reads (although we would consider a design which allows this to be strange), while our definition of concurrently-readable does not.

We note that the literature on concurrently-readable algorithms often refers to the algorithms' read operations as *lock-free*. This use of the term is different than the definition of lock-free concurrent objects. Therefore, we instead refer to operations which do not use locks as *lockless*.

The idea of almost non-blocking objects is due to Boehm [8]. An *almost non-blocking* shared object is one that is *N-non-blocking* for some $N > 0$. The definition of *N-non-blocking* requires the concept of *inactive* threads. We say that a thread is *inactive* if it fails to execute instructions at some pre-determined minimum rate while trying to update a data structure; this is an attempt to model threads which are descheduled, blocked, or crashed altogether. Then, a shared object O is *N-non-blocking* if and only if:

- any number of processes may concurrently access O, and,
- if at most N *inactive* threads are trying to update O, and at least one active thread is trying to access or update O, then some thread will succeed in accessing or updating O in a bounded amount of time.

The relationship between these five properties may be confusing. We show the hierarchy of these properties in Figure 2.1, along with their relation to conventional reader-

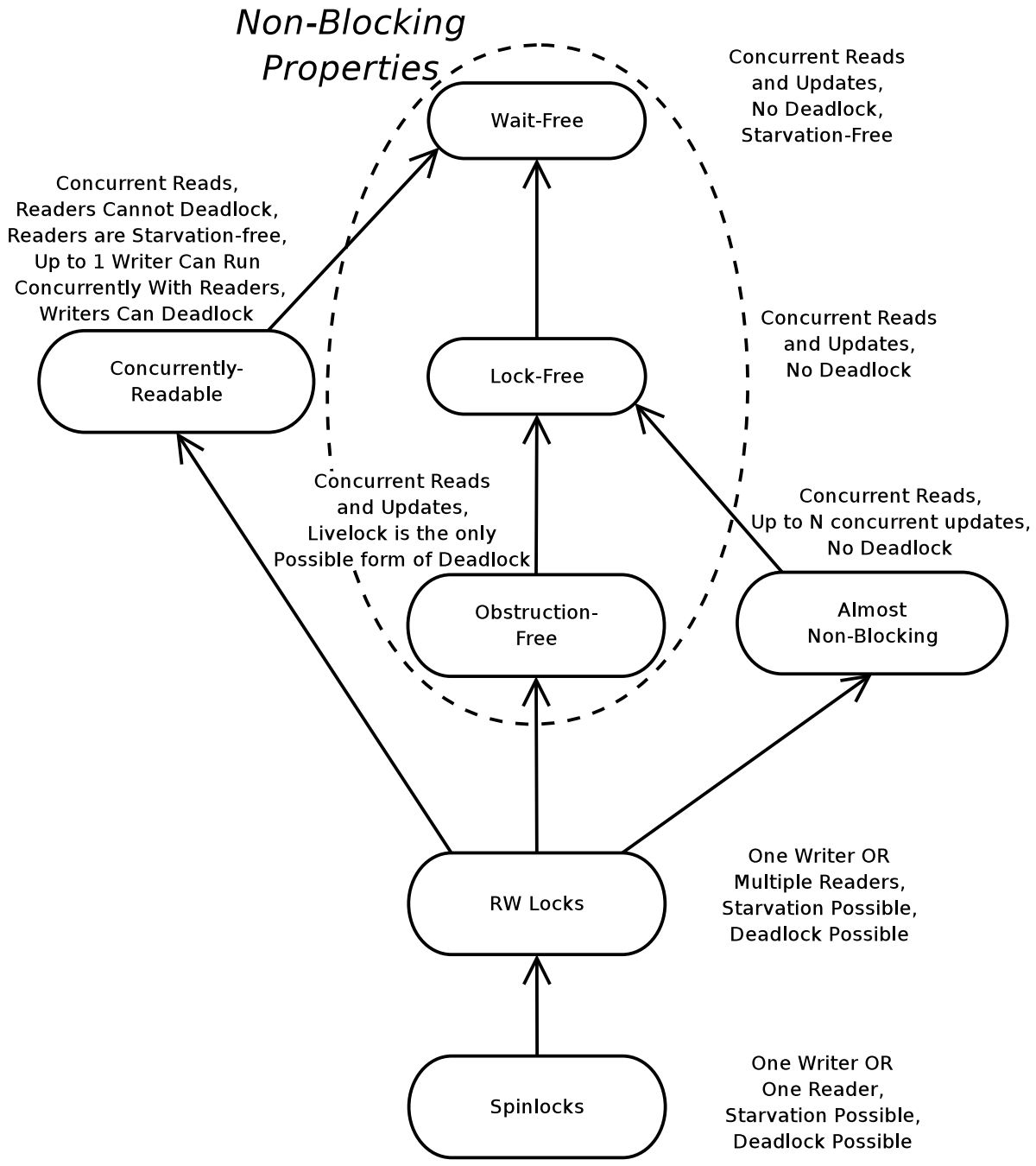

Figure 2.1: Hierarchy of properties of concurrent objects, represented as a partial order. Any property in the hierarchy is strictly stronger than all properties below it; for example, *obstruction-free* and *almost non-blocking* are weaker than *lock-free*, but in different ways.

```
BOOL CAS (void *A, long B, long C)
{
    atomically do {
        if (*A == B) {
            *A = C;
            return TRUE;
        } else {
            return FALSE;
        }
    }
}
```

Figure 2.2: Pseudocode definition of CAS.

writer locking and spinlocking, both of which may be more familiar to the reader.[1] We are primarily interested in lock-free and concurrently-readable shared objects; the other properties are noted only for completeness.

2.1.3 Hardware Operations

Non-blocking shared object implementations typically use either the *compare-and-swap (CAS)* operation or the *load-linked* and *store-conditional (LL/SC)* pair of instructions to update shared pointers. Both of these instructions are conditional synchronization primitives which atomically both check if a certain condition has been met, and update a word in memory if the check succeeds. CAS and LL/SC are defined as shown in Figures 2.2 and 2.3, respectively.

Typically, processors built according to the complex instruction set (CISC) paradigm will provide a CAS operation, while those built according to the reduced instruction set (RISC) paradigm will provide (restricted) LL/SC. Each of CAS and LL/SC can be used to implement the other, and both may be desirable. LL/SC is often used to implement CAS,

[1] This is a minor abuse of terminology, since reader-writer locks and spinlocks are mechanisms, not properties.

```
WORD LL (void *A)
{
    return *A;
}

BOOL SC (void *A, WORD w)
{
    atomically do {
        if (A has not been written to since this thread last called LL(A)) {
            *A = w;
        }
    }
}
```

Figure 2.3: Pseudocode definitions of LL and SC.

```
BOOL CAS (void *A, long B, long C)
{
    do {
        if (LL(A) ≠ B) {
            return FALSE;
        }
    } while (SC(A,C) == FALSE);
    return TRUE;
}
```

Figure 2.4: Pseudocode for implementing CAS using LL/SC.

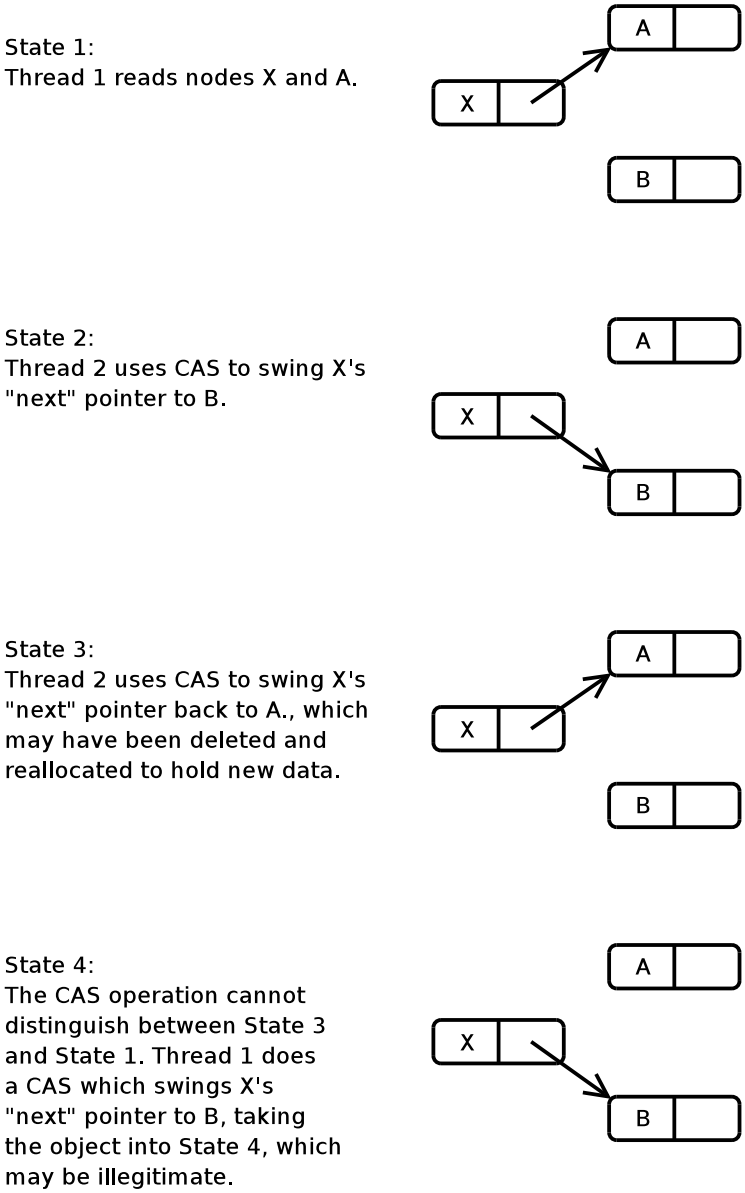

Figure 2.5: Illustration of the ABA problem.

since CAS is intuitive, and convenient for implementing many non-blocking algorithms. Implementing CAS using LL/SC is quite trivial, as shown in Figure 2.4.

Implementations of LL/SC using CAS are more involved [14, 48], but may be desirable in order to avoid the ABA problem. This problem occurs when CAS is used to implement a lock-free algorithm, and is illustrated in Figure 2.5. The effect of two CAS operations can make two states of a data structure indistinguishable to a third CAS operation, which could then succeed and take the data structure into an illegal state.

2.2 Memory Consistency Models

Current literature on lock-free algorithms generally assumes a sequentially-consistent [33] memory model. However, for performance reasons, modern architectures provide a weakly-consistent memory model; sequential consistency can be enforced when needed by using special *fence* instructions. We formalize the key properties of the weakly-consistent memory models used by the processor architectures in this study, using the terminology given in [17].

Let A and B be instructions, let M be a word in memory, and let $<_p$ and $<_m$ represent partial orders. We say that $A <_p B$ if and only if A and B are executed on the same processor, and A precedes B in the program order, and that $A <_m B$ if and only if A is executed before B in all valid program execution orders. The processors in this study provide guarantees of *coherency*, *self-consistency*, and *dependency consistency*, defined as follows[2]:

- Coherency: At any given point in execution time, M has only one value, and this value is eventually visible to all processors.

- Self-consistency: If A and B both access M, and $A <_p B$, then $A <_m B$.

[2]Most current processors provide these guarantees; we do not consider processors such as the DEC Alpha 21264 that do not provide dependency consistency.

- Dependency consistency: If A and B are executed on the same processor and B depends on a control decision by A or a state written by A, then $A <_m B$.

Fence instructions can enforce particular orderings when necessary as follows. If A and B are instructions executed on processor P, X is a fence executed on P, and $A <_p X <_p B$, then $A <_m B$. Distinct *write fences* and *read fences*, which impose orderings on writes and reads, respectively, may also be provided.

Our work involves performance measurement on current commodity machines; specifically, IA-32 and PowerPC processors. Since fence instructions are expensive operations, we cannot ignore them in our analysis. Indeed, we will show that the base costs of the reclamation strategies considered depend almost entirely on the number of fences they require. In addition, they are an often-hidden factor in the performance of several lock-free algorithm implementations.

Chapter 3

Lock-free and Concurrently-readable Algorithms

In this chapter, we present an overview of the literature on lock-free and concurrently-readable algorithms. We present in more depth the algorithms used in our analysis.

3.1 Theory of Non-blocking Synchronization

Lamport [32] introduced the idea of concurrent computing without mutual exclusion. Herlihy deepened the theory of non-blocking synchronization by showing that the power of concurrent objects and operations can be characterized by their ability to solve the *consensus* problem [25]. There exists a consensus hierarchy, partially shown in Figure 3.1, such that any object or operation on level n of the hierarchy can, in combination with atomic read/write registers, be used to solve the consensus problem for n processes, but not $n+1$ processes. This result established that strong synchronization primitives such as CAS and LL/SC are required for implementing practical non-blocking synchronization, while weaker primitives such as Test&Set are insufficient.

Herlihy presented a *universal* method to transform any sequential object into an n-process wait-free concurrent one [25] using n-process consensus objects. He later pre-

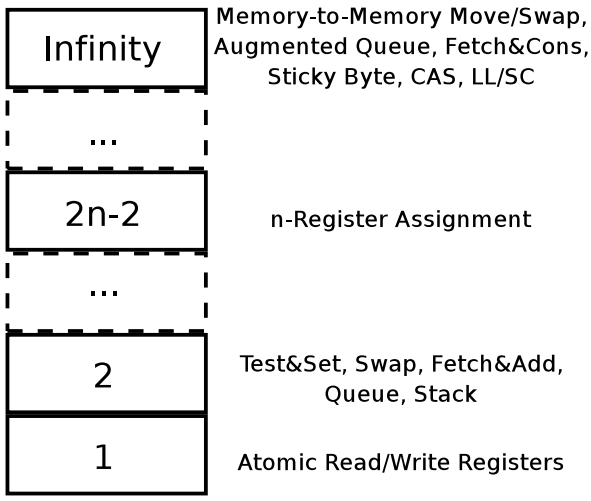

Figure 3.1: The consensus hierarchy.

sented a more efficient transformation based on LL/SC; however, even this more efficient transformation yields implementations which usually have higher overhead than their lock-based counterparts [26]. This poor performance of universal transformations provided a motivation for less general, but more high-performance, non-blocking object implementations.

3.2 Practical Non-blocking Algorithms

Herlihy's universal transformations are too inefficient to be practical. Alemany and Felten [2] identified useless parallelism and unnecessary copying as partial causes of this inefficiency. Several others have attempted to design more efficient universal transformations [4, 3, 7, 52]. However, no universal transformation yet devised can match the performance of custom non-blocking algorithms [19].

```
BOOL DCAS (void *A1, void *A2, long B1, long B2, long C1, long C2)
{
    atomically do {
        if (*A1 == B1 && *A2 == B2) {
            *A1 = C1;
            *A2 = C2;
            return TRUE;
        } else {
            return FALSE;
        }
    }
}
```

Figure 3.2: Pseudocode definition of DCAS.

3.2.1 Higher-level primitives

Many early attempts to build lock-free data structures using techniques more efficient than expensive universal constructions used the *double compare-and-swap (DCAS)* operation shown in Figure 3.2. This operation was useful, for example, to update both a pointer and a version number simultaneously (see example given in [20]). The only two lock-free operating system kernels yet developed, Synthesis and Cache Kernel, both used DCAS [37, 20]. Unfortunately, DCAS was not supported by any processor other than the Motorola 680x0 series, and hardware implementations of DCAS are considered impractical with current processor technology.

DCAS is not a convenient enough primitive to make the design of lock-free algorithms easy [13]. Although it is easier to design a lock-free algorithm using DCAS than it is with CAS or LL/SC, constructing DCAS-based lock-free algorithms is still difficult. Many researchers believe that *multi-word compare-and-swap (MCAS)*, which operates on an arbitrary number of words independently, is the correct primitive for easy construction of arbitrary non-blocking shared objects [17, 19, 23]. Fraser showed that MCAS is substantially easier to use than CAS, but has moderate overhead and performs poorly

when contention for a data structure is high [17].

Another approach to easy non-blocking synchronization is *transactional memory*, implemented either in software [30, 17], or hardware [55, 21]. Transactional memory allows operations to be grouped into transactions which atomically succeed or fail. Fraser [17] and Herlihy et. al. [30] showed that transactional memory makes lock-free algorithm design relatively simple. However, software transactional memory has very high overhead [17], and whether or not hardware transactional memory will be practical in future processors in still unknown.

The data structures we consider can be implemented efficiently using CAS, so MCAS and transactional memory are unnecessary. Nevertheless, we believe that our results would also be applicable to more complex data structures which currently require these mechanisms.

3.2.2 Algorithms using CAS or LL/SC

Lock-free algorithms using CAS or LL/SC out-perform versions using MCAS or software transactional memory, but are more difficult to construct. Only a handful of data structures have known lock-free implementations that do not require higher-level abstractions or universal transformations. Among them are linked lists [22, 43], chaining hash tables [43], skip lists [15, 17], doubly-linked lists [58], queues [51, 47], priority queues [57], stacks [60, 47, 24], and deques [45, 58]. More complex data structures, such as binary search trees and red-black trees, currently require higher-level primitives [17, 30].

We note that, despite the relatively-small number of lock-free algorithms based on CAS or LL/SC, many of them appeared after DCAS-based versions (compare [22] to [20], and [45] to [10], for example). In the future, we may see lock-free implementations of other data structures requiring only CAS or LL/SC.

In our experiments, we focus on the lock-free queue and chaining hash table implementations presented in [47]; the latter is simply an array of lock-free linked lists. We

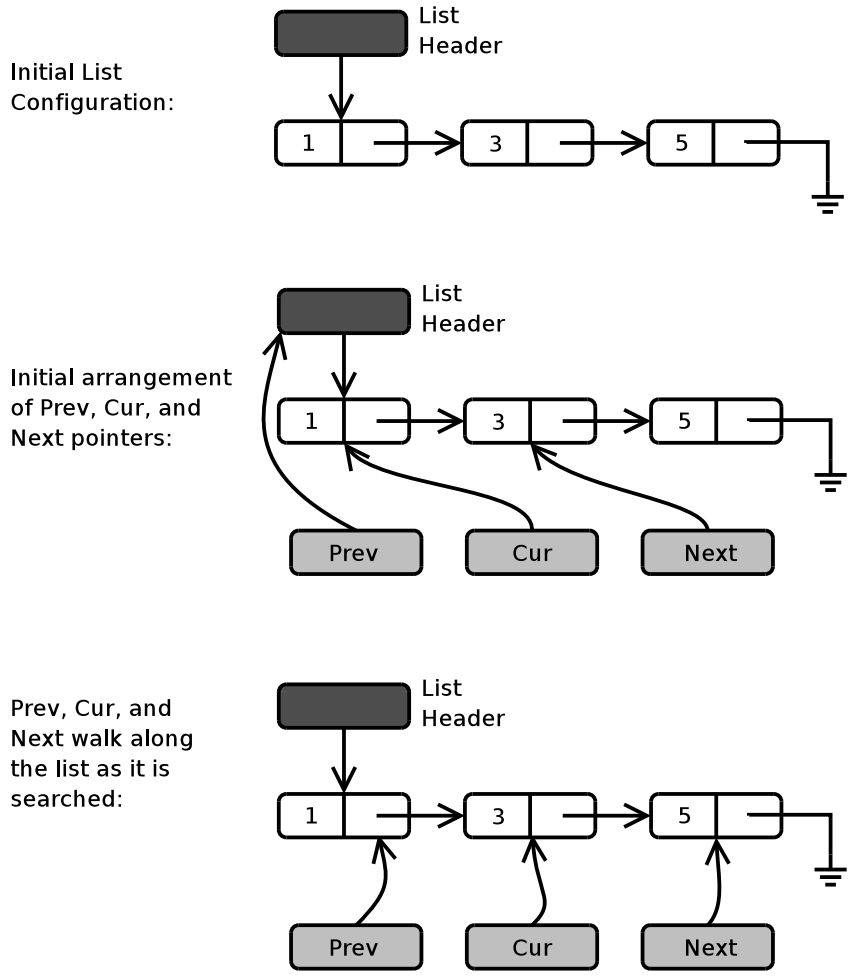

Figure 3.3: Example operation of search function for lock-free linked list.

therefore present outlines of the algorithms for lock-free linked lists and queues.

Lock-free Linked Lists

We use the lock-free linked list presented in [47], which is an improvement by Michael on an original design by Harris [22]. The list stores its keys in sorted order, and does not allow duplicate keys. It is singly-linked, $NULL$-terminated, and has a head node. The supported operations are searching the list for a node with a given key, deleting a node with a given key, and inserting a new node with a given key.

Figure 3.3 shows an example of a search for the key 2 in such a list, and Figure 3.4

```
node **prev;
node *next;
node *cur;

int search (node **head, long key)                                    5
{
try_again:
        prev = head;
        cur = *prev;
        while (cur != NULL) {                                        10
                if (*prev != cur) goto try_again;
                next = cur->next;
                /* If the low-order bit is a 1, the node is marked to be logically deleted. */
                if (next & 1) {
                        /* Update the link and logically delete the node. */   15
                        if (!CAS(prev, cur, next-1)) goto try_again;
                        schedule_for_deletion(cur);
                        cur = next-1;
                } else {
                        if (*prev != cur) goto try_again;            20
                        if (cur->key >= key) {
                                return (cur->key == key);
                        }
                        prev = &cur->next;
                }                                                    25
        }
        return (0);
}
```

Figure 3.4: Pseudocode for search function for lock-free list, stripped of memory reclamation code.

shows the associated pseudocode. The searching thread walks the list, and as it does so, it keeps track of the current node, the *next* pointer of the current node's predecessor (or the *head* pointer if the current node is the first in the list), and the current node's successor. These three references are used by CAS operations in the insertion and deletion routines. The thread stops once it encounters the first key greater than or equal to the key for which it is searching (see lines 21-23 of Figure 3.4).

If a searching thread encounters a partially-deleted node, it must attempt to help complete this deletion, and then restart its traversal (lines 13-18; see the explanation of the deletion function, below). The thread also restarts its traversal if it detects that a node has been inserted between the current node and its predecessor (line 20). If such an insertion takes place after the check in line 20, and the caller of the search function is an inserting or deleting thread, the calling thread will have to invoke the search function again; the check serves to decrease the chances of this occurring.

More formally, the search function finds a pointer to a pointer to a node, $prev$, and pointers to nodes cur and $next$ such that:

- If list is empty, $prev = head$, $cur = NULL$, and the search returns $false$.
- If the key is not in the list, then $prev$ points to the $next$ pointer of the last node in the list, $cur = NULL$, $next = NULL$, and the search function returns $false$.
- If there exists some node q in the list such that:
  - q has not been marked for deletion (see the explanation of the delete function, below), and
  - $q \rightarrow key >= key$ and either q is the first node in the list or $q \rightarrow prev \rightarrow key < key$,

  then $*prev = cur$, $cur = q$, $next = q \rightarrow next$, and the search returns $true$ if and only if $q \rightarrow key = key$.

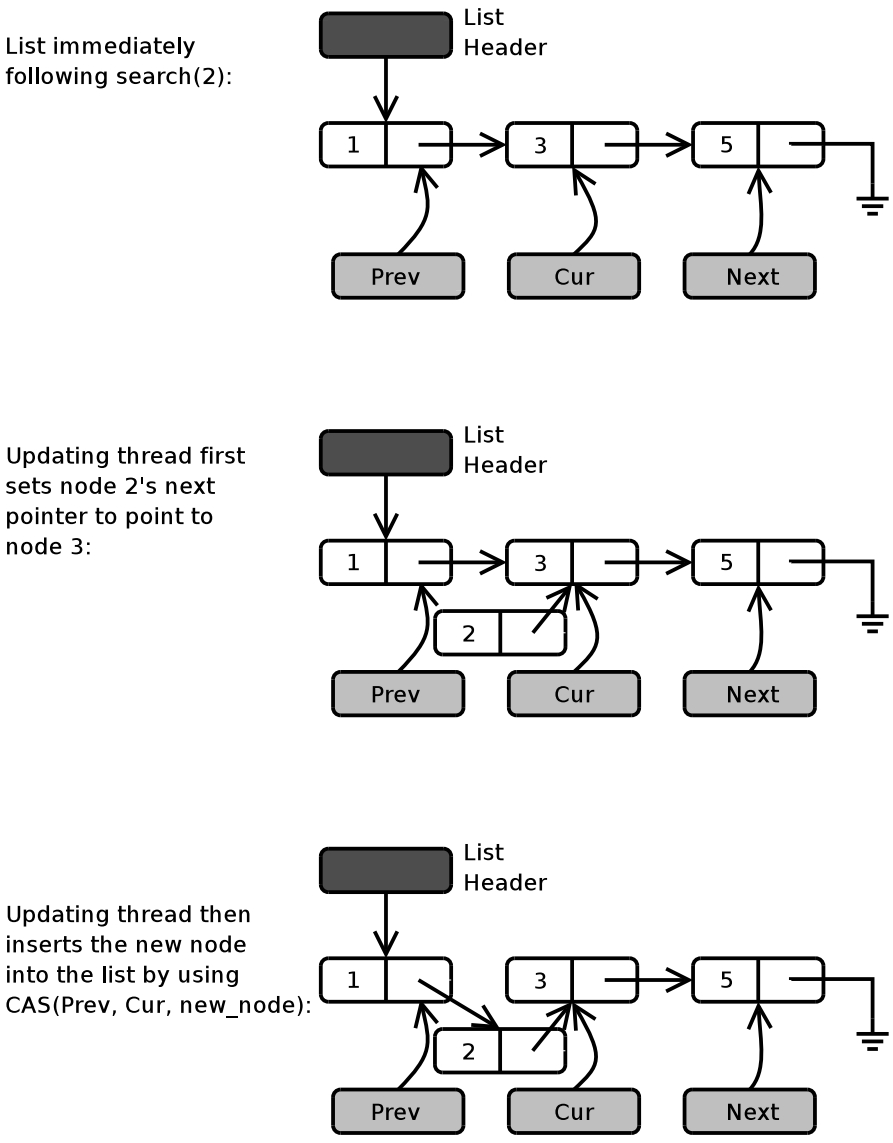

Figure 3.5: Example operation of insert function for lock-free linked list.

An insertion of a node with key 2 is shown in Figure 3.5. The insertion begins by searching for a node with key 2, as shown in Figure 3.3. This search ensures that we do not insert a duplicate key; furthermore, if no node with key 2 is found, it gives us the position in which to insert the new node with key 2 in order to keep the list sorted. We then initialize the new node's *next* pointer to point to the node with key 3 in Figure 3.3, and update the *next* pointer of the node with key 1 using CAS, thereby linking the new node into the list. If the CAS operation fails, then we must restart the insertion from the beginning. Note that the steps of the insert operation *must* be performed in this order, or else readers may follow the *next* pointer of the node with key 2 before it has been initialized, leading to indeterminate behavior.

Figure 3.7 shows a deletion of the node with key 3 from the linked list. Deletion is slightly more complex than insertion, since we must prevent the possibility of concurrent insertions and deletions corrupting the list, as shown in Figure 3.6. To do so, we first search for the node with the key we wish to delete, and mark the low-order bit of this node's *next* pointer using CAS. This is possible on current architectures because words lie on 4-byte boundaries; hence, the low-order two bits of any pointer are always zero. Marking the low-order bit using CAS will either fail or cause concurrent insertions which would otherwise corrupt the list, such as that shown in Figure 3.6, to fail. After marking the low-order bit, we then use CAS again to unlink the node from the list, thus completing the logical deletion.

Full details on the algorithm, including proofs of correctness, are available in [22], [43], and [47].

Lock-free Queues

Our lock-free queue is that presented in [50] and [47]; our code is structured according to the pseudocode given in the latter. A queue is represented by a singly-linked list with *head* and *tail* pointers. The implementation of the queue uses a dummy node which is

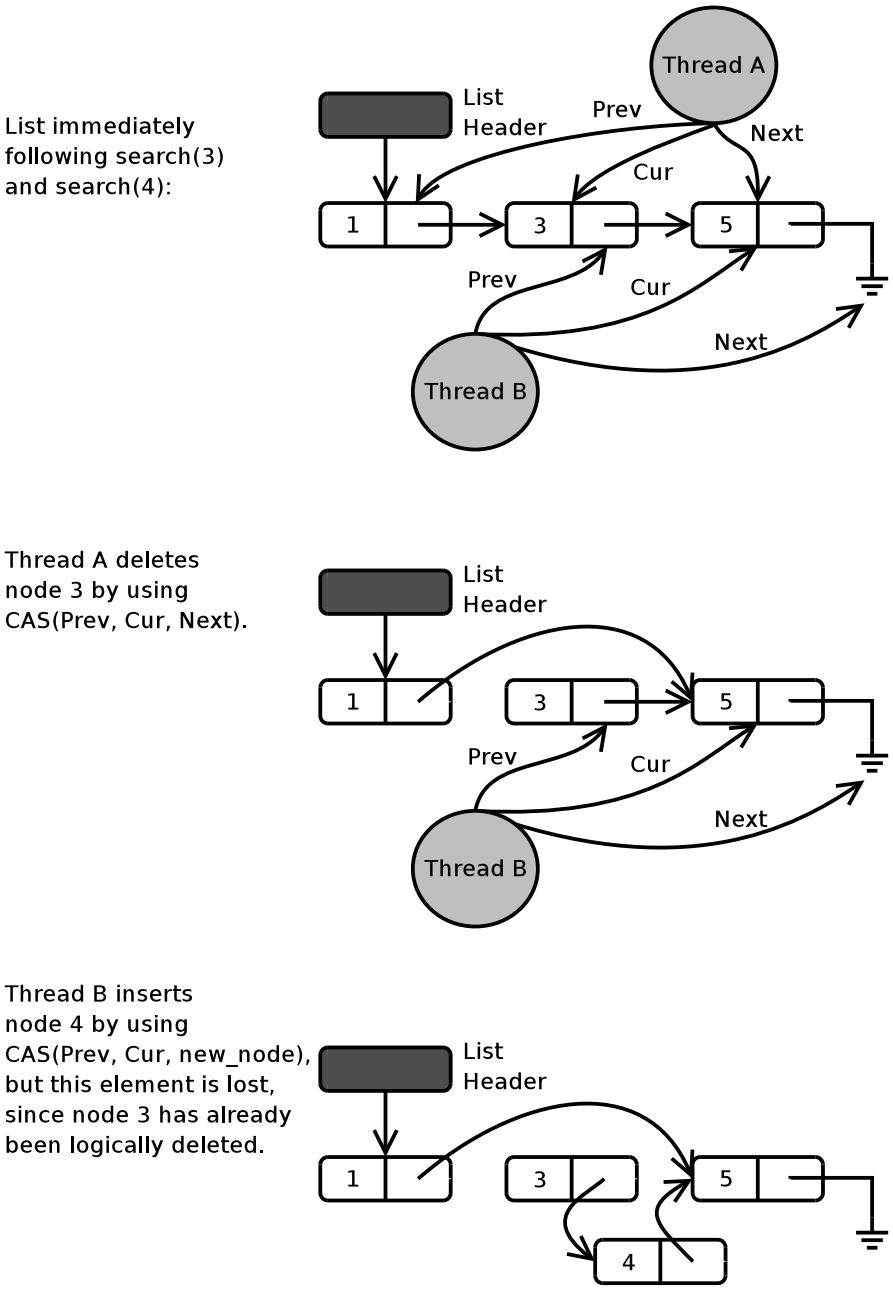

Figure 3.6: Error which can occur in a naïve lock-free linked list implementation when insertions and deletions are interleaved. To prevent such errors, the lock-free linked list must mark a node's *next* pointer before deleting the node (Figure 3.7).

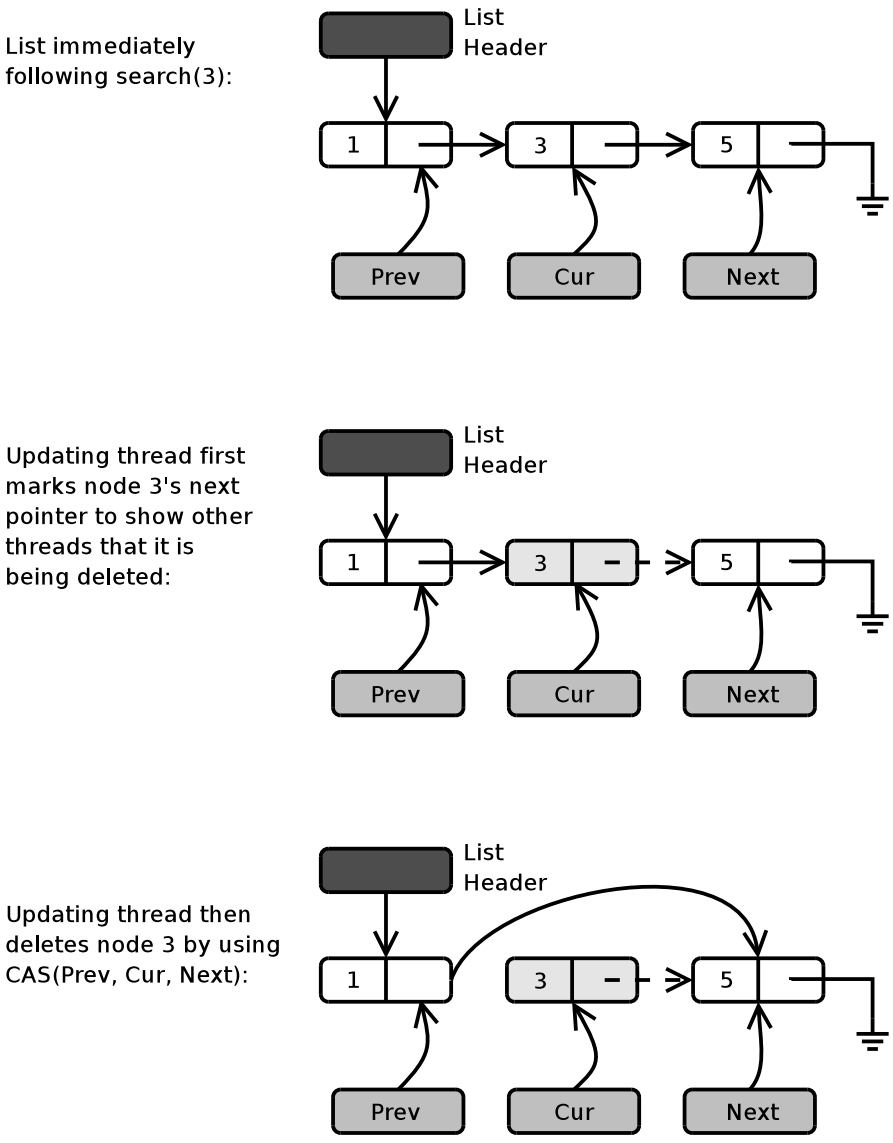

Figure 3.7: Example operation of delete function for lock-free linked list.

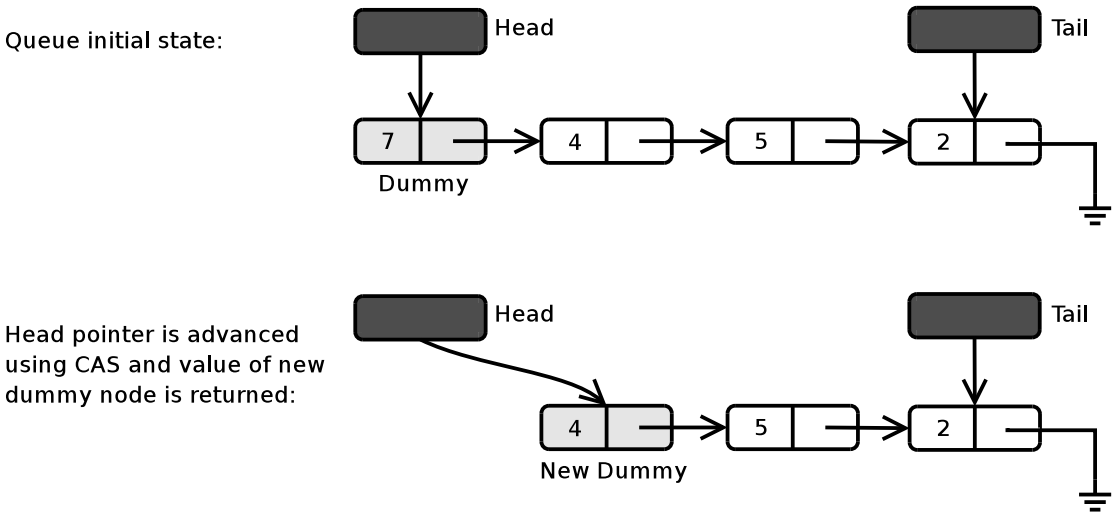

Figure 3.8: Dequeue from non-empty lock-free queue.

not logically part of the queue; hence, the *head* and *tail* pointers always have something to which to point. The dummy node is always either a node created for this purpose when the queue is initialized, or the most recently dequeued node. The *head* pointer must always point to the dummy node, and the *tail* pointer must always point to either the last or second-last node in the queue. Enqueue operations append nodes to the *tail* of the list, and dequeue operations remove nodes from the *head* of the list.

Figure 3.8 illustrates a dequeue operation, and Figure 3.9 shows the associated pseudocode. The dequeue operation begins by taking a consistent snapshot of pointers to the *head* (dummy) node and its successor, and to the *tail* node (lines 7-15 of Figure 3.9). If the only node in the queue is the dummy node, then the queue is empty; otherwise, if the *tail* pointer points to the dummy node, the dequeuing thread must attempt to advance the *tail* pointer using CAS, and then retry (lines 17-25). The thread then advances the *head* pointer to point to the dummy node's successor (lines 30-33); this node then becomes the new dummy node, and the old dummy node can be logically deleted. The new dummy node's key is then returned to the calling function.

An enqueue operation is shown in Figure 3.10, with the associated pseudocode shown

```
long dequeue(struct queue *Q)
{
  node *h, *t, *next;
  long data;

  while (1) {
    /* Get the old head and tail nodes. */
    h = HEAD(Q);
    t = TAIL(Q);

    /* Get the head node's successor. */
    next = h->next;
    memory_barrier();
    if (HEAD(Q) != h)
      continue;

    /* If the head (dummy) node is the only one, return EMPTY. */
    if (next == NULL)
      return EMPTY_SENTINEL;

    /* There are multiple nodes. Help update tail if needed. */
    if (h == t) {
      CAS(&TAIL(Q), t, next);
      continue;
    }

    /* Save the data of the head's successor. It will become the new dummy node. */
    data = next->key;

    /* Attempt to update the head pointer so that it points to the new dummy node. */
    if (CAS(&HEAD(Q), h, next))
      break;
  }

  /* The old dummy node has been unlinked, so reclaim it. */
  schedule_for_deletion(h);

  return data;
}
```

Figure 3.9: Pseudocode for dequeue function for lock-free queue, stripped of memory reclamation code.

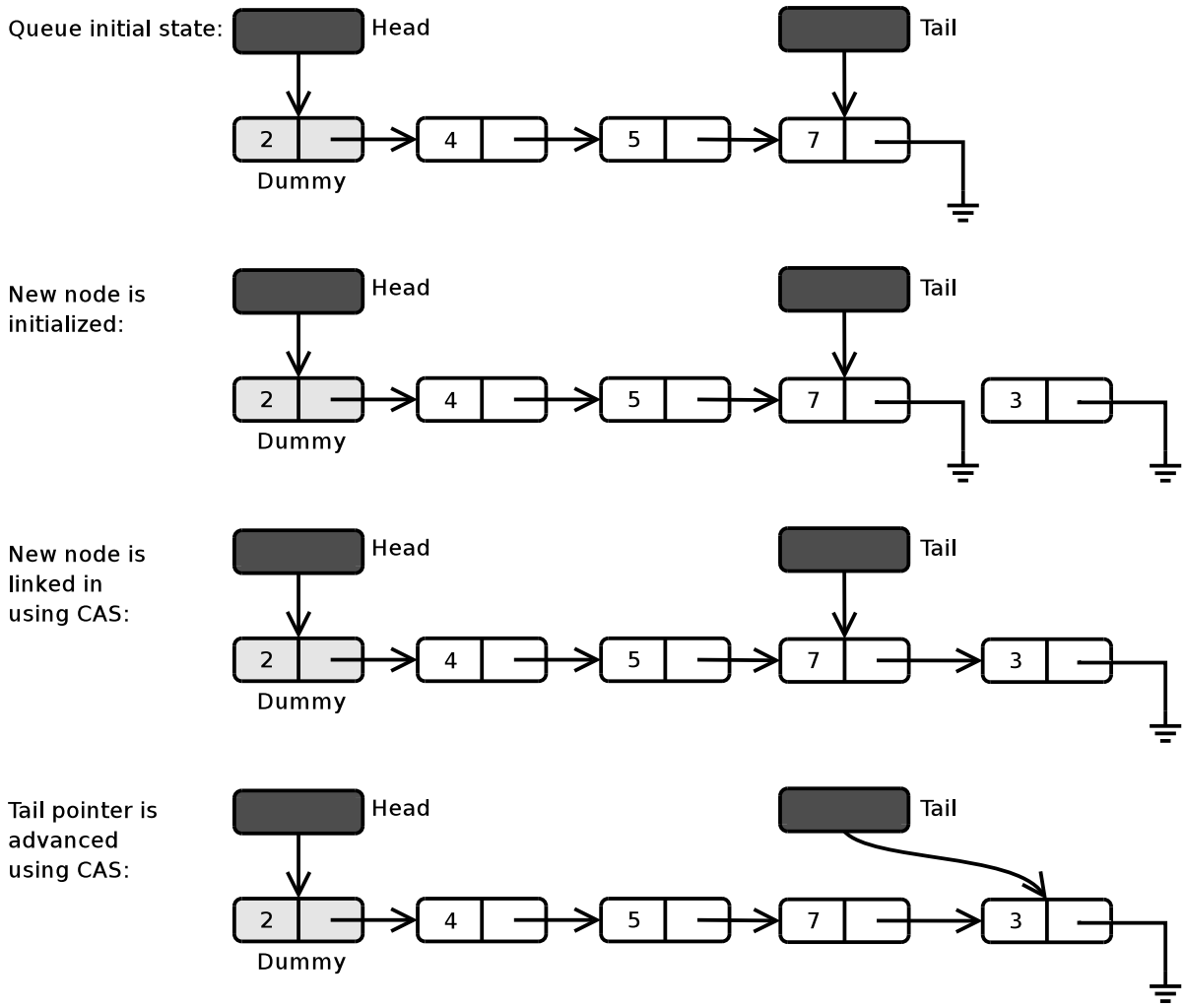

Figure 3.10: Enqueue to non-empty lock-free queue.

```
void enqueue(long data, struct queue *Q)
{
  node *newnode = allocate_new_node();
  node *t, *next;

  /* Initialize the new node. */
  newnode->key = data;
  newnode->next = NULL;

  /* Ensure that newnode->next = NULL before inserting it. */
  write_barrier();

 while(1) {
    /* Snapshot the old tail pointer and its successor. */
    t = TAIL(Q);
    next = t->next;
    if (TAIL(Q) != t)
      continue;

    /* Help update the tail pointer if needed. */
    if (next != NULL) {
      CAS(&TAIL(Q), t, next);
      continue;
    }

    /* Attempt to link in the new node. */
    if (CAS(&t->next, NULL, &newnode))
      break;
  }

  /* Swing the tail to the new node. */
  CAS(&TAIL(Q), t, &newnode);
}
```

Figure 3.11: Pseudocode for enqueue function for lock-free queue, stripped of memory reclamation code.

in Figure 3.9. First, the enqueuing thread allocates a new node, stores the key to be enqueued in it, initializes its *next* pointer to NULL, and executes a write fence (lines 3-11 of Figure 3.9). The thread then takes a snapshot of pointers to the *tail* node and its successor (lines 14-18). If the *tail* node's successor is not $NULL$, then the *tail* pointer must be pointing to the second-last node in the queue, so the thread attempts to advance the *tail* pointer, and then retries (lines 20-24). Next, the thread uses CAS to insert the new node at the end of the list; if this CAS fails, the thread must retry (lines 26-28). Once the CAS succeeds, a second CAS operation attempts to advance the queue's *tail* pointer to point to the new node (line 32); no failure condition is needed on this latter CAS, since it fails only if another thread has already succeeded in updating the *tail* pointer.

As with the lock-free list, full details, including a proof of correctness, are available in [51] and [47].

We note that both of these lock-free algorithms are quite complex implementations of relatively simple data structures. Much of this complexity is due to the fact that lock-free algorithms must coordinate multiple concurrent updates. Algorithms accommodating multiple readers, but only a single writer, can be significantly simpler.

3.3 Concurrently-Readable Algorithms

The most well-known use of concurrently-readable algorithms is in implementing read-copy update [38, 42, 39, 40, 41, 6]. These algorithms focus on concurrently-readable linked lists and chaining hash tables, although other applications of read-copy update exist [38]. We note that we examine concurrently-readable chaining hash tables in our experiments, and that these hash tables are simply arrays of concurrently-readable linked lists; therefore, we present an outline of these lists.

Figure 3.12 shows an example from [42] of an update to a node of a linked list which

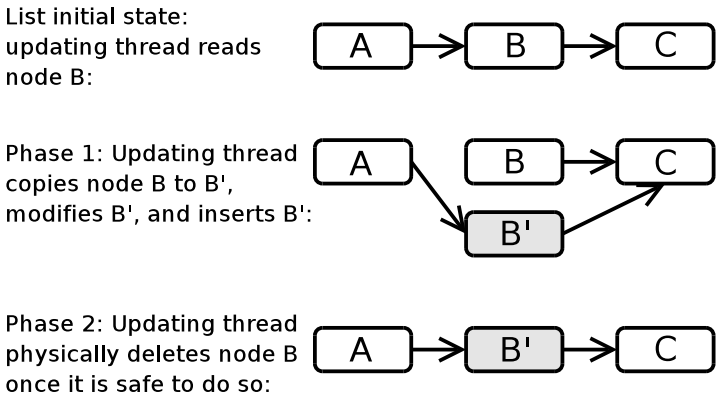

Figure 3.12: Concurrently-readable node modification example from which *read-copy update* derives its name.

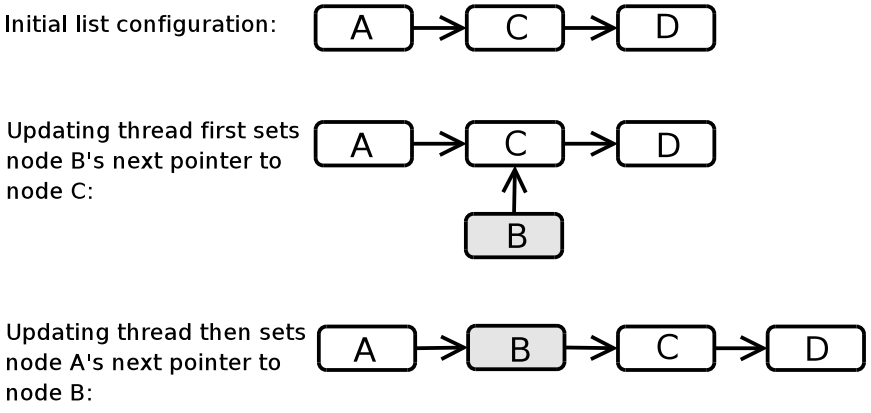

Figure 3.13: Concurrently-readable insertion.

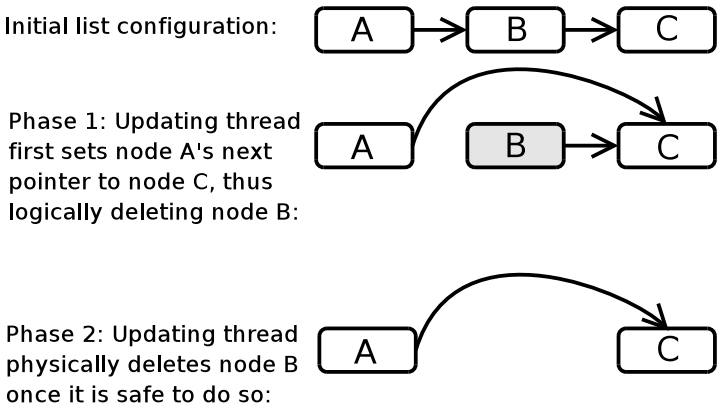

Figure 3.14: Concurrently-readable deletion.

can run concurrently with lockless reads; insertions and deletions are slightly simpler, and are shown in Figures 3.13 and 3.14. We focus on the example of the update shown in Figure 3.12. Readers merely traverse the list as they would if the program were single-threaded. A writer acquires a per-list spinlock, so writers need not deal with concurrent writes. Writes then proceed in two phases. First, the updating thread makes a copy of the node to be updated, and performs all needed modifications to the copy. The copy's *next* pointer is then made to point to the original node's successor, then, the *next* pointer of the original's predecessor must be updated to point to the modified node. As with the lock-free linked list algorithm, the updates *must* be performed in this order, or else lockless reads may follow the updated node's *next* pointer before it has been initialized. In the second phase of the update, the writer physically deletes the original node once it is safe to do so.

In a theoretical model with infinite memory, the second phase would be unnecessary and the algorithm would be trivial. The algorithm becomes more interesting when examined in a practical setting and combined with a high-performance memory reclamation scheme [42].

Chapter 4

Memory Reclamation Schemes

In this chapter, we present our three memory reclamation schemes — EBR, QSBR, and SMR, along with other schemes which we did not consider. For the schemes which we do consider, we show how they can be applied, and compare them analytically. The next chapter provides experimental validation of this analysis.

4.1 Descriptions of Schemes

This section details the three memory reclamation schemes under consideration: quiescent-state-based reclamation (QSBR), safe memory reclamation (SMR), and epoch-based reclamation (EBR). We also discuss reference counting, Greenwald's type-stable memory, and Pass the Buck, and explain why we did not consider these schemes. Since all these methods have been published elsewhere [42, 6, 44, 47, 17], we discuss them in only enough detail for the reader to understand our work.

4.1.1 Blocking Methods

We describe three blocking memory reclamation schemes: epoch-based reclamation, quiescent-state based reclamation, and Greenwald's type-stable memory. These methods

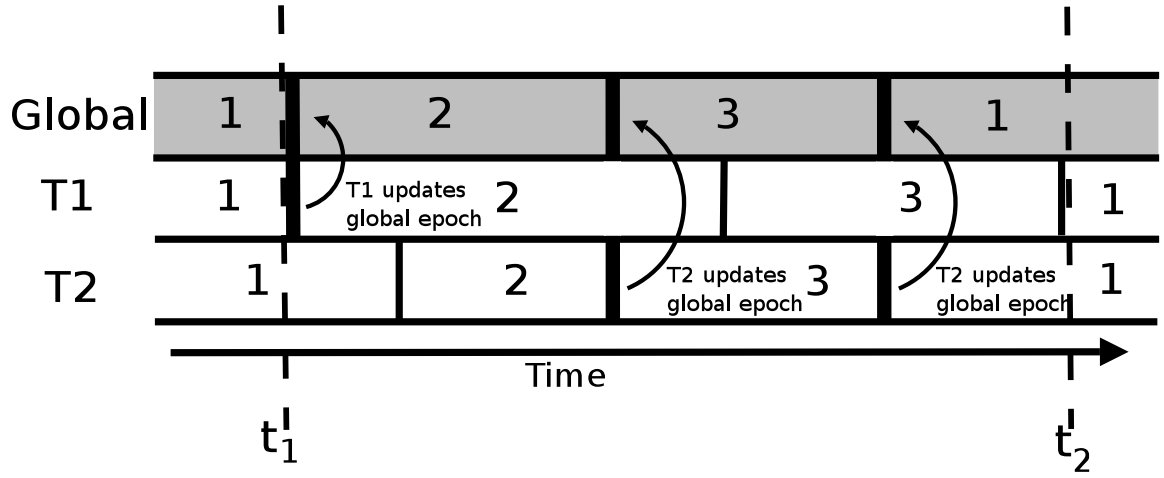

Figure 4.1: Illustration of EBR. Threads follow the global epoch. If a thread observes that all other threads have seen the current epoch, then it may update the global epoch. Hence, if the global epoch is e, threads in critical sections can be in either epoch $e+1$ or e, but not $e-1$ (all mod 3). The time period $[t_1, t_2]$ is thus a grace period for thread $T1$.

are all blocking, because they force threads to wait for some condition, which could be delayed arbitrarily, to become true, and therefore place no upper bound on the amount of unfreed memory at any given time. Since the amount of unfreed memory is unbounded, the system may run out of memory, thus causing threads to block on memory allocation and therefore fail to make progress. Figure 4.7 of section 4.3, below, shows how blocked threads can obstruct memory reclamation.

Epoch-Based Reclamation

Epoch-based reclamation (EBR) was introduced by Fraser [17], but builds on earlier ideas [31, 36, 53]. At any point in time, each thread is executing in one of three logical epochs. For each of the three epochs, the thread has an associated *limbo list* which holds logically deleted nodes awaiting physical deletion. When a thread T is in epoch e, it places all nodes that it logically deletes in limbo list e. T may physically delete these

nodes once a *grace period* has passed. A grace period $[a, b]$ is an interval of program execution time such that, after point b, all nodes logically deleted before point a can be physically deleted safely. EBR uses epochs to detect grace periods, as explained below.

EBR is illustrated in Figure 4.1. At the start of any lock-free operation, a thread enters a *critical section* with respect to memory reclamation (note that this use of the term *critical section* has nothing to do with mutual exclusion). Upon entering a critical section, the thread updates its local epoch to match the global epoch if the two epochs differ, as indicated by the thinner lines in Figure 4.1. It also sets a per-thread flag indicating to other threads that it is in a critical section. Upon exit of a critical section, a thread clears its flag. No thread is allowed to access an EBR-protected object outside of a critical section.

Upon entering a programmer-determined number of critical sections since seeing the global epoch change, a thread may attempt to update the global epoch. If any thread which is in a critical section has not updated its local epoch to match the global epoch, then this attempt to update the global epoch must fail. EBR therefore guarantees that at any time t, if the global epoch is e, the local epoch of each thread in a critical section is either e or $e+1$, but not $e-1$ (all mod 3). As a result, whenever a thread sets its local epoch to e, it can physically delete all nodes logically deleted the last time that it was in epoch e, since all operations which could have held a reference to the logically deleted node have completed.

EBR is completely encapsulated within a library, and is invisible to the application programmer. Further, threads that are not in critical sections can not obstruct the progress of EBR. These factors make EBR very generally applicable, and easy for a programmer to use.

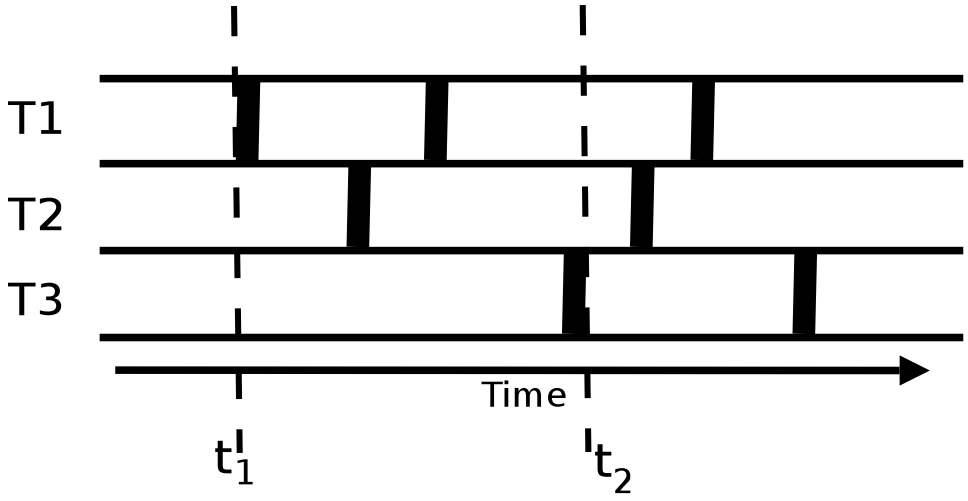

Figure 4.2: Illustration of QSBR. Thick lines represent quiescent states. The time interval $[t_1, t_2]$ is a grace period: at time t_2, each thread has passed through a quiescent state since t_1, so all nodes logically removed before time t_1 can be physically deleted.

Quiescent-State-Based Reclamation

Instead of dividing time into epochs, QSBR has the programmer identify quiescent states in the application code. A *quiescent state* for thread T is a point in T's program code at which T can hold no reference to any shared node; hence, from T's point-of-view, all nodes logically deleted by other threads can safely be physically deleted. A *grace period* for QSBR is an interval of execution time during which each thread passes through at least one quiescent state. QSBR is illustrated in Figure 4.2.

QSBR must enable threads to detect grace periods so that they can physically delete logically deleted nodes. However, no QSBR implementation is required to detect the smallest grace periods possible. Furthermore, unlike with EBR, the definition of a quiescent state is application-dependent. Natural and convenient quiescent states exist for many operating system kernels — the domain in which quiescent-state-based reclamation is used to implement read-copy update [42, 40, 18].

The fact that QSBR is application-dependent is the fundamental difference between QSBR and EBR. EBR, by definition, detects grace periods at the library level. QSBR,

by contrast, requires that the application report quiescent states to the QSBR library. As we show in Section 5.2, this gives QSBR a significant performance advantage over EBR.

Type-Stable Memory Management

EBR and QSBR both guarantee that a node is never physically deleted unless no thread can hold a reference to it. Type-stable memory (TSM) [20, 19] makes a weaker guarantee: a node's memory *cannot be re-used for an object of another type* until no thread can hold a reference to it.

Greenwald [19] outlines both kernel-level and user-level implementations of TSM. The kernel-level implementation relies on "safe points" which are equivalent to quiescent states. The user-level version uses per-type reference counters.

Like EBR and QSBR, Greenwald's TSM implementations are blocking, and hence suffer from the same drawbacks. Furthermore, TSM places additional burdens on the programmer, such as having to check after finding a node that the node has not been reallocated and inserted into another data structure. Such checks would add programming complexity and performance overhead to lockless linked list searches. Due to this disadvantage, and Greenwald's TSM's lack of any apparent advantages relative to EBR and QSBR, we do not consider it in our experiments.

4.1.2 Lock-free Methods

Here, we present the three lock-free memory reclamation schemes of which we are aware: reference counting, safe memory reclamation, and Pass the Buck.

Reference Counting

Implementations of lock-free reference counting have been proposed by Valois [61] (corrected by Michael and Scott [50]), Sundell [59], and Detlefs et al. [11, 12]. Valois'

scheme uses compare-and-swap (CAS) and fetch-and-add instructions to manage reference counts, and requires that nodes retain their type after deletion. Sundell's scheme is based on Valois', but is wait-free. The method of Detlefs et al. allows the memory used by nodes to be re-used for structures of other types, but requires the double-compare-and-swap operation, which no current architecture supports in hardware.

Reference counting has been shown by Michael [47] to introduce performance overhead which makes lock-free algorithms perform worse than their lock-based counterparts in most situations. We thus omit reference counting from our experiments.

Safe Memory Reclamation

Safe memory reclamation was introduced by Michael [44]. It provides a simple and intuitive existence locking mechanism for dynamic nodes. Each thread which accesses a lock-free or concurrently-readable data structure has K hazard pointers which it uses to protect nodes from deletion by other threads. The required number of hazard pointers, K, is algorithm-dependent, and is typically very small: queues and linked lists need $K = 2$ hazard pointers, while stacks require only $K = 1$. If the total number of threads in the system that may access an SMR-protected data structure is N, then we need $H = NK$ hazard pointers in total.

When a thread T removes a node from a dynamic data structure, it places a reference to that node in a private list. When the list grows to size R, the thread attempts to physically delete all nodes in the list. R is a constant chosen by the programmer; a higher value of R will mean that memory remains unfreed for a longer period of time, but memory reclamation overhead will be amortized over a larger number of operations. However, to ensure that the expected amortized processing time per reclaimed node is kept constant, R must be chosen such that $R = H + \Omega(H)$. [1]

[1]The terminology $R = H + \Omega(H)$ is confusing to some. Roughly, this means that R should be parameterized by H, the total number of hazard pointers, and that R must always be greater than H by an amount which is at least linear in H. Hence choosing $R = aH + b$ where $a > 1$ and b is a constant

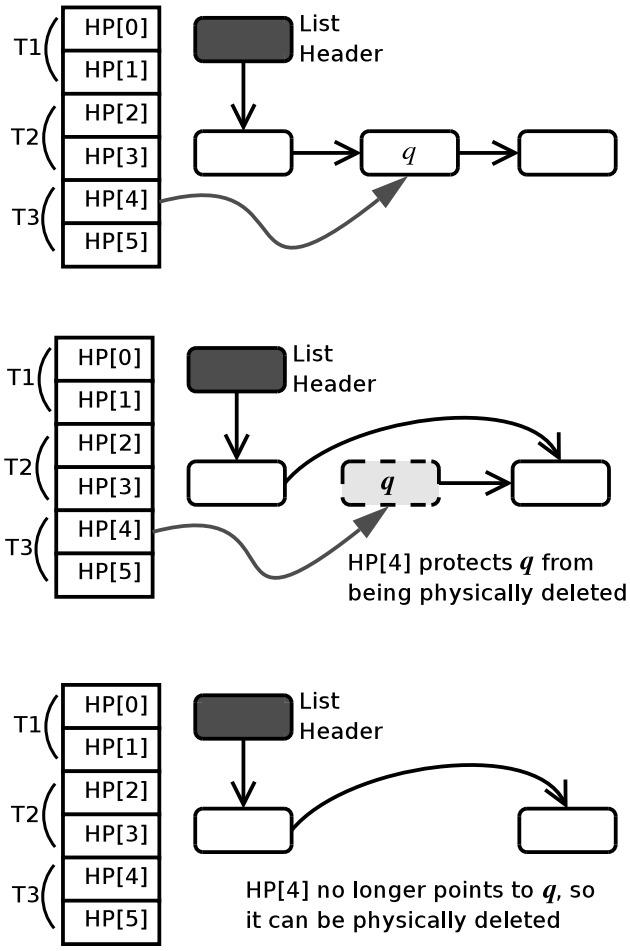

Figure 4.3: Illustration of SMR. q is logically removed from the linked list on the right of the diagram, but cannot be physically deleted while $T3$'s hazard pointer $HP[4]$ is still pointing to it.

To physically free nodes, T copies all non-NULL hazard pointers of all threads into a private array, which it then sorts. For each node n in T's private list of nodes to be reclaimed, T does a binary search for a pointer to n, and physically deletes n if no such pointer is found.

To use hazard pointers, an algorithm must identify all *hazardous references* — references to shared objects that may have been deleted by other threads (or are vulnerable to the ABA problem if hazard pointers are being used for ABA-protection) [47] — in all lockless operations. Before using any such reference, a hazard pointer must be made to point to the target object. After setting the hazard pointer, an algorithm-specific check must be made in order to ensure that the protected object has not been deleted; the rules of the SMR deletion routine outlined above guarantee that the object will never be deleted so long as the hazard pointer continues to point to it. Figure 4.3 illustrates the use of SMR.

One cited advantage of SMR is that it requires only atomic reads and writes, and is therefore usable on hardware platforms which do not support CAS or LL/SC [47]. Since most current hardware platforms support one of these strong synchronization primitives, the advantage of not requiring these operations lies mostly in avoiding their significant performance cost.

Pass the Buck

Herlihy et. al. [28, 27] present Pass the Buck, a solution to the *Repeat Offender Problem*, which is an attempt to formalize the problem of lock-free memory reclamation. Pass the Buck is similar to SMR, but uses expensive CAS operations while physically deleting nodes, and lacks SMR's amortized bound on the memory reclamation overhead per physically deleted node. On the upside, Pass the Buck has a property called *value progress*, which guarantees that logically deleted nodes will eventually be freed, even if there are

suffices.

thread failures.

We do not consider value progress attractive enough to justify Pass the Buck's higher overhead relative to SMR, so we do not include Pass the Buck in our experiments; however, due to Pass the Buck's similarity to SMR, we believe that our experimental analysis of SMR relative to QSBR and EBR would be applicable to Pass the Buck as well.

4.2 Applying the Schemes

As explained in Chapter 3, we examined three algorithms requiring deferred memory reclamation: a concurrently-readable chaining hash table, a lock-free chaining hash table, and a lock-free queue. We found that all three of these algorithms were compatible with each of our three memory reclamation schemes.

We illustrate this compatibility by way of the lock-free queue's `dequeue()` method. We chose this method because it is the simplest method which demonstrates the use of these schemes. Figures 4.4, 4.5, and 4.6 demonstrate the use of SMR, QSBR, and EBR, respectively. Since this is actual code and not pseudocode, some conventions must be explained. Our nodes are of type `struct el`. Lists are implemented using the doubly-linked list interface of the Linux kernel [35]: each node contains an instance of `struct list_head`, which contains two pointers: `struct list_head *prev` and `struct list_head *next`. The function `list_entry()` maps an instance of `struct list_head*` to a pointer to the `struct el` which contains it. Hazard pointers are implemented using a cacheline-aligned structure, `struct hazard_pointer`, which has one member: `struct el *p`.

SMR, QSBR, and EBR all register *callbacks* for logically deleted nodes. A callback is simply a record of the logically deleted node, and the function to be used to physically delete it once it is safe to do so; in our experiments, this function is always `kfree()`. Our

SMR implementation hides our callback interface within the body of the `retire_node()` function; however, the interface is exposed in our QSBR and EBR implementations, as shown in line 39 of Figure 4.5 and line 40 of Figure 4.6, respectively. We ask the reader to forgive this minor inconsistency in our interfaces.

The code in Figure 4.4, which illustrates the use of SMR, is the most complex of the three versions of the `dequeue()` function. For convenience, it uses two pointers to hazard pointers, `hp0` and `hp1`, which are first set to point to the two hazard pointers owned by the calling thread (lines 7-9 of Figure 4.4). After making a copy of the value of the queue's head pointer, we protect the head from deletion using a hazard pointer (lines 12-14). We then execute a fence instruction, after which we ensure that the head has not changed - and hence possibly been deleted (lines 15-17). Once we are sure that this has not happened, we know that the head will not be physically deleted until our hazard pointer is unset. A similar step must be taken after acquiring a pointer to the head node's successor (lines 22-27). If, after setting either of these hazard pointers, we find that the head of the queue has changed, we must retry our dequeuing attempt (lines 17 and 27).

Once we have successfully logically deleted the dequeued node, we can retire it using the `retire_node()` function (line 50), and unset our hazard pointers so that the nodes they pointed to can be reclaimed if necessary (line 51).

The code illustrating QSBR in Figure 4.5 is much simpler. Since reclamation works by keeping track of quiescent states at the application level, the code for the `dequeue()` method is not burdened by any memory reclamation code. The code in Figure 4.6, which illustrates the use of EBR, is almost identical. The only differences are that the code for EBR places calls to `critical_enter()` and `critical_exit()` at the beginning and end of the method, respectively (lines 7 and 41 of Figure 4.6), and that it schedules logically deleted nodes for physical deletion using `call_epoch_kfree()` instead of `call_rcu_kfree()` (line 40 of Figure 4.6 and line 39 of Figure 4.5).

```
long dequeue(struct queue *Q)
{
  struct list_head *h, *t, *next;
  struct el *node;
  long data;

  /* Initialize our hazard pointer pointers. */
  hp0 = &(HP[smp_processor_id()*K]).p;
  hp1 = &(HP[smp_processor_id()*K+1]).p;

  while (1) {
    /* Protect the old head node. */
    h = HEAD(Q);
    *hp0 = list_entry(h, struct el, list);
    memory_barrier();
    if (HEAD(Q) != h)
      continue;

    /* Get a pointer to the old tail node. */
    t = TAIL(Q);

    /* Get and protect the head node's successor. */
    next = h->next;
    *hp1 = list_entry(next, struct el, list);
    memory_barrier();
    if (HEAD(Q) != h)
      continue;

    /* If the head (dummy) node is the only one, return EMPTY. */
    if (next == NULL)
      return -1; /* Empty. */

    /* There are multiple nodes. Help update tail if needed. */
    if (h == t) {
      CAS(&TAIL(Q), t, next);
      continue;
    }

    /* Save the data of the head's successor. It will become the
     * new dummy node. */
    data = list_entry(next, struct el, list)->key;

    /* Attempt to update the head pointer so that it points to the
     * new dummy node. */
    if (CAS(&HEAD(Q), h, next))
      break;
  }

  /* The old dummy node has been unlinked, so reclaim it. */
  retire_node(list_entry(h, struct el, list));
  *hp0 = *hp1 = NULL;
  return data;
}
```

Figure 4.4: Lock-free queue's `dequeue()` function, using SMR.

```
long dequeue(struct queue *Q)
{
  struct list_head *h, *t, *next;
  struct el *node;
  long data;

  while (1) {
    /* Get the old head and tail nodes. */
    h = HEAD(Q);
    t = TAIL(Q);

    /* Get the head node's successor. */
    next = h->next;
    memory_barrier();
    if (HEAD(Q) != h)
      continue;

    /* If the head (dummy) node is the only one, return EMPTY. */
    if (next == NULL)
      return -1; /* Empty. */

    /* There are multiple nodes. Help update tail if needed. */
    if (h == t) {
      CAS(&TAIL(Q), t, next);
      continue;
    }

    /* Save the data of the head's successor. It will become the
     * new dummy node. */
    data = list_entry(next, struct el, list)->key;

    /* Attempt to update the head pointer so that it points to the
     * new dummy node. */
    if (CAS(&HEAD(Q), h, next))
      break;
  }

  /* The old dummy node has been unlinked, so reclaim it. */
  call_rcu_kfree(new_callback(), list_entry(h, struct el, list));
  return data;
}
```

Figure 4.5: Lock-free queue's `dequeue()` function, using QSBR.

```
long dequeue(struct queue *Q)
{
  struct list_head *h, *t, *next;
  struct el *node;
  long data;

  critical_enter();

  while (1) {
    /* Get the old head and tail nodes. */
    h = HEAD(Q);
    t = TAIL(Q);

    /* Get the head node's successor. */
    next = h->next;
    memory_barrier();
    if (HEAD(Q) != h)
      continue;

    /* If the head (dummy) node is the only one, return EMPTY. */
    if (next == NULL)
      return -1; /* Empty. */

    /* There are multiple nodes. Help update tail if needed. */
    if (h == t) {
      CAS(&TAIL(Q), t, next);
      continue;
    }

    /* Save the data of the head's successor. It will become the
     * new dummy node. */
    data = list_entry(next, struct el, list)->key;

    /* Attempt to update the head pointer so that it points to the
     * new dummy node. */
    if (CAS(&HEAD(Q), h, next)) break;
  }

  /* The old dummy node has been unlinked, so reclaim it. */
  call_epoch_kfree(new_callback(), list_entry(h, struct el, list));
  critical_exit();
  return data;
}
```

Figure 4.6: Lock-free queue's `dequeue()` function, using EBR.

The fact that each of the three memory reclamation methods is compatible with each of the the three algorithms we considered is important. To date, the literature on QSBR has concentrated on using QSBR with locking methods, usually a variant of the concurrently-readable linked list described in section 3.3; furthermore, the literature concentrates on using QSBR to support lockless reads. We show, by using QSBR with lock-free queues, that it makes sense to use QSBR even with data structures which do not have read-only operations.

One caveat about using QSBR or EBR for memory reclamation concerns the ABA problem, which was explained in section 2.1.3. SMR can be used to make an algorithm ABA-safe [46]. To make an algorithm ABA-safe using QSBR or EBR, we must ensure that we do not re-insert a node which has been removed from a data structure until the node has been physically deleted and reallocated. Since most node implementations consist of only a pointer to the next node and a pointer to the node's data, allocating new nodes to hold data which must be re-inserted into a data structure should be relatively-inexpensive; hence, this constraint should not cause programmers undue difficulties.

We note that not all algorithms are compatible with all memory management techniques. Some, such as the deques of [58] and Harris' original version of our lock-free linked list [22], must reference logically-deleted nodes. For such algorithms, neither SMR, EBR, nor QSBR is usable; these algorithms typically use reference counting. Other algorithms, such as the deques of [10], are also SMR-incompatible and assume automatic garbage collection. These incompatibilities are detailed in [44]. The existence of such incompatibilities prevents us from claiming that the choice of memory reclamation scheme is completely independent of the target algorithm; instead, based on our successful implementations, we claim only that the two are *mostly* independent.

4.3 Analytic Comparison of Methods

Here we lay out an analytic comparison of the memory reclamation strategies under consideration, which we validate with performance data from our experiments in Section 5.2. We identify the following factors which could affect the performance of our memory reclamation schemes:

- **Object contention:** Many threads may attempt to access or modify the same object concurrently; in the case of updates, these operations may conflict with one another, thus forcing one or more threads to retry. Having more threads performing operations on the same number of objects will increase object contention.

- **CPU contention:** If there are more threads than there are physical CPUs, some threads will be descheduled for periods during which other threads may perform large numbers of operations. A descheduled thread may delay the progress of other threads under blocking memory reclamation schemes.

- **Workload:** Objects typically support several operations such as search, insert, delete, enqueue, and dequeue. The *workload*, in this paper, is the proportion of each type of operation in a given experiment. If an object has read-only operations, we use the term *update fraction* to refer to the proportion of the operations invoked on the object which are not read-only.

- **Traversal length:** In the case of structures such as linked lists and trees, a process performing a search operation will have to traverse several nodes; we refer to the number of nodes accessed as the *traversal length*.

- **Execution environment:** External factors such as the memory allocator and OS scheduler.

When object contention, CPU contention, and traversal length are low, and the workload is read-mostly, QSBR should have considerably less per-operation runtime overhead

than either SMR or EBR. This is not obvious until we recall that we are working in a weakly-consistent memory model in which fences are necessary. As we show in Section 5.2, these fence instructions make SMR and EBR more expensive than QSBR in most situations. In the case of SMR, a fence is necessary between the setting and the validation of any hazard pointer, since no hazard pointer can be validated until it has been set and is visible to all threads. EBR requires a flag to be set upon entry into any critical section, and cleared upon exit of the critical section. A fence is necessary after setting and before clearing the flag. QSBR has no per-operation code to manage quiescent states, and hence no fences are required. As a result, the per-operation overhead of using QSBR, in the best case, is very low.

We note that we could modify EBR so that critical sections are entered and exited at the application level instead of the library level, thus amortizing the overhead of the fence instructions across more operations. Doing so, however, would make EBR application-dependent. The point of comparing the performance of EBR to that of QSBR is to evaluate the performance benefits of using an application-dependent method to detect grace periods.

Traversal length will be the primary factor influencing the performance of SMR. While traversing a linked list, a thread must, for each node, set a hazard pointer, execute a fence instruction, and validate the hazard pointer. In a contention-free case in which the thread never has to restart its traversal, the number of fences needed will be $O(n)$, where n is the traversal length. If there is object contention on the linked list so that the thread may have to restart its traversal, the maximum number of fences required is unbounded. In contrast, EBR needs exactly two fences per operation, no matter how many times a thread may have to restart its traversal.

We can expect CPU contention to adversely affect QSBR and EBR. Descheduled threads can delay other threads' memory reclamation, as shown in Figure 4.7. In extreme cases, this could lead to out-of-memory errors, which could cause threads to block on

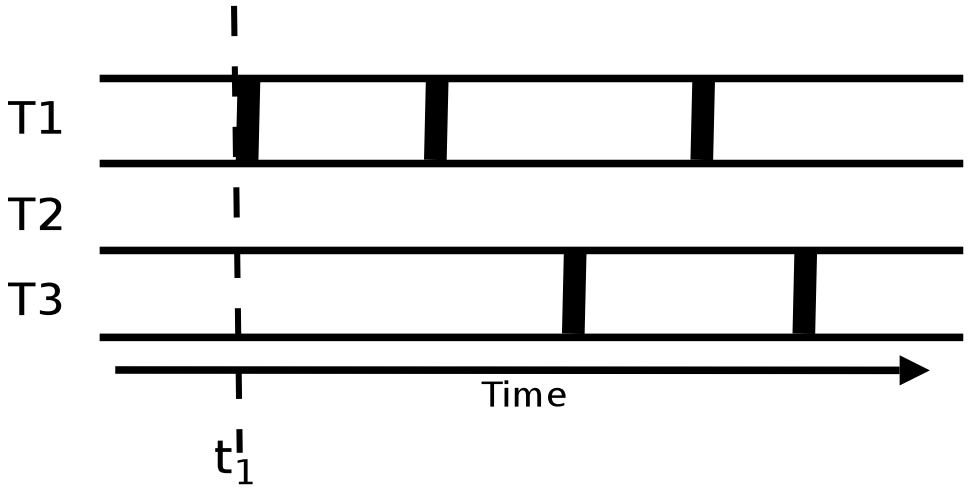

Figure 4.7: Illustration of why QSBR is inherently blocking. Here, thread $T2$ does not go through a quiescent state for a long period of execution time; hence, threads $T1$ and $T3$ must wait to reclaim memory. A similar argument holds for EBR.

memory allocation, severely degrading performance. Furthermore, if there are locks in the memory allocator, these more memory-hungry methods may increase the contention on these locks. If a thread is descheduled while holding a lock on a global freelist, other threads will block on memory allocation. These effects will be most noticeable when the workload has a high percentage of operations which must allocate memory.

The choice of memory allocator and OS scheduler could also significantly impact QSBR and EBR. Although lock-free memory allocation is possible [49], most memory allocators use locking. The unbounded memory use of QSBR and EBR could cause an algorithm using these methods to need to allocate memory from a lock-protected global pool much more frequently than the same algorithm using SMR, therefore increasing the contention on the pool's lock. Furthermore, since thread delays can delay memory reclamation, the policy of the OS scheduler could play a huge part in how long memory is left unreclaimed. The only strategy that provides a provable bound on the amount of unfreed memory at any point in time is SMR [44]; we thus expect it should be less sensitive to the memory allocator and other factors in the external environment than QSBR and EBR.

Our analysis allows us to gain some intuition into the workings of these three memory reclamation schemes; however, we cannot analytically quantify the extent to which the factors outlined above impact each method. We must therefore evaluate these schemes experimentally.

Chapter 5

Experimental Evaluation

In this chapter, we describe our experiments. We first describe the setup we used, and then detail our results.

5.1 Experimental Setup

We evaluated our memory reclamation strategies on two data structures — a queue and a hash table — using systems based on PowerPC and IA-32 processors, while independently varying each of the factors outlined in Section 4.3. This section provides details on these aspects of our experiments.

5.1.1 Algorithms Compared

The hash table used in our experiments is an array of buckets, where each bucket has a linked list of keys. Duplicate keys are not allowed. The lock-free hash table implementation is that given in [47] and described in section 3.2.2, which, we reiterate, stores the keys in each hash chain in sorted order. We therefore stored the keys in the lists for the lock-based algorithms in sorted order as well, so that we could fairly compare the performance of these alternatives as we increased the load factor of the hash table.

We compared the lock-free hash tables to a version using per-bucket spinlocks, and the concurrently-readable version described in section 3.3. The lock-free queue is the version of Michael and Scott's implementation given in [47] and described in section 3.2.2. We compare it to a simple spinlock-based queue.

The algorithms for lock-free queues, lock-free hash tables, and concurrently-readable hash tables were paired with each of the three memory reclamation schemes, for a total of nine combinations. We concentrate on the six combinations involving lock-free algorithms.

Spinlocks were implemented using the CAS operation and fence instructions. CAS is provided in hardware on IA-32, and implemented using LL/SC on PowerPC. No exponential back-off was used, since our primary interest is in cases of low CPU contention — that is, when the number of threads does not exceed the number of processors — in which back-off is unlikely to be useful.

5.1.2 Test Program

In our experiment, a parent thread creates n child threads, starts a timer, and stops the threads once that timer expires. Child threads keep track of the number of operations they perform, and report this value to the parent. The parent then calculates the average execution time per operation by dividing the total number of operations performed by all children by the duration of the test. Our tests performed seven trials, and reported the average of the median five.

Each thread runs repeatedly through a test loop. Once the loop completes, a quiescent state is identified if we are using QSBR, and the loop is begun again if the parent thread's timer has not yet expired, as shown in Figure 5.1. For hash tables, on each iteration of the loop, the thread does either a search, an insertion, or a deletion. The probabilities of performing an insertion or a deletion are always equal, to keep the average load factor constant throughout the trial. For queues, the thread does either an enqueue or a dequeue

```
while (parent's timer has not expired) {
    for i from 1 to 100 do {
        key = random key;
        op = random operation;
        if (testing queues) {
            q = random queue;
            op(q,key);
        } else { /* Testing a hash table */
            op(key);
        }
    }
    QUIESCENT_STATE();
}
```

Figure 5.1: High-level pseudocode for the test loop of our program. Each thread executes this loop. The call to QUIESCENT_STATE() is ignored unless we are using QSBR.

on each operation, again with equal probability.

The tests allow us to vary the number of queues or hash buckets, the number of threads, and the total number of nodes we begin with. In the case of hash tables, we are also able to vary the load factor and the update fraction.

As shown in Figure 5.1, each thread performs 100 operations per quiescent state; hence, the overhead of announcing a quiescent state is amortized over 100 operations. For EBR, each *op* in Figure 5.1 is a critical section; a thread attempts to update the global epoch whenever it has entered a critical section 100 times without seeing the global epoch updated. For SMR, we chose $R = 2H + 100$.

5.1.3 Operating Environment

We performed our experiments on two machines: one with two 2.0 GHz PowerPC G5 processors, and another with two 2.8 GHz Intel Xeon processors with no hyperthreading. The PowerPC machine ran Mac OS X Server version 10.3.3 with Darwin kernel version 7.4.0, while the Xeon machine ran Red Hat Enterprise Linux version 3 with Linux kernel

Table 5.1: Characteristics of Machines

	Machine 1	**Machine 2**
Processor	PowerPC	IA-32
# CPUs	2	2
GHz	2.0	2.8
Kernel	Darwin	Linux
Memory Model	weakly-consistent	weakly-consistent, writes ordered
Full Fence	86 ns	73.4 ns
Write Fence	13 ns	0 ns
CAS	33.9 ns	78.6 ns
Lock + Unlock	166 ns	141 ns

version 2.4.21-20.ELsmp. The Xeon machine has a slightly stronger memory model in which writes are always performed in program order; write fences therefore do not add to the costs of algorithms on this machine. Table 5.1 summarizes the properties of these machines; the last line refers to the combined cost of locking and unlocking a spinlock. The difference in CAS costs on the two architectures is partially due to the fact that on IA-32, a CAS implies a full fence, while on PowerPC it does not. For consistency, we report the results from experiments performed on the PowerPC machine unless otherwise noted — in almost all cases, the choice of architecture made no significant performance difference.

Threads in our experiment are Unix processes. Our memory allocator provides per-thread freelists. Each thread can have two freelists of 100 nodes each at any given time. Threads which exhaust their freelists can acquire more memory from a spinlock-protected global pool. This design is similar to that of the slab allocator with magazines [9]. All nodes are pre-allocated by the parent before the test begins.

5.1.4 Limitations of Experiment

Although we believe that these experiments provide significant insight into the behavior of different memory reclamation schemes, we do not know how accurately our microbenchmark reflects real applications. Some applications may not have natural quiescent states. Furthermore, detecting quiescent states in other applications may be more expensive than it is in our program. Our QSBR implementation, for example, has less overhead than that used in the Linux kernel, which must deal with issues such as CPU hotplugging and the need to support interrupt handlers and real-time workloads.

Our experiment is also limited by the fact that the threads do nothing but invoke operations on a small number of objects repeatedly, and that our memory allocator uses locking. Performing a macrobenchmark on an existing application and testing the performance of these schemes under a lock-free memory allocator are two avenues for future work.

Each iteration of our test loop calls the `random()` function at least once. This adds a constant amount of overhead to our measurements. Since we are primarily interested in overall trends, this overhead is not a major concern in our analysis.

Finally, we were limited by the hardware we had available. Although testing on commodity dual-CPU hardware evaluates these memory reclamation strategies under realistic conditions, we were unable to evaluate how these schemes scale to large numbers of CPUs or on other platforms such as NUMA machines or clusters.

Despite these limitations, we feel that our analysis highlights trends that show when each memory reclamation scheme is and is not efficient.

5.2 Performance Analysis

Our experiments show how varying object contention, CPU contention, workload, and traversal length significantly affect the performance of each memory reclamation scheme.

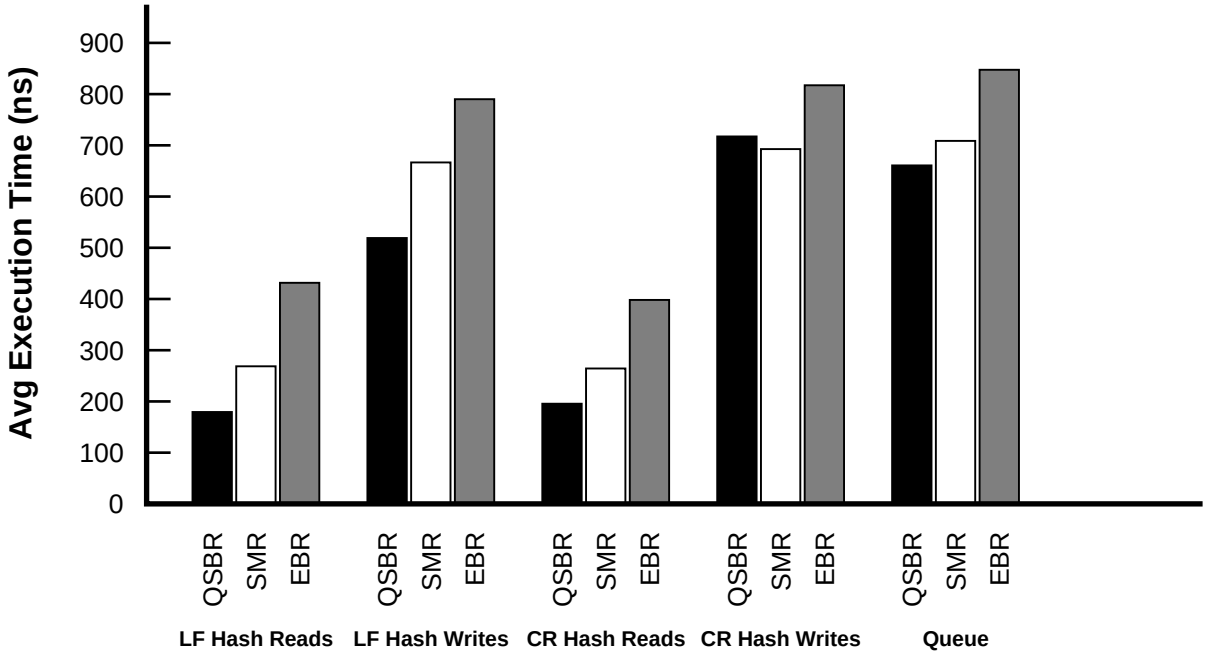

Figure 5.2: Single-threaded memory reclamation costs on PowerPC. Hash table statistics are for a 32-bucket hash table with a load factor of 1. Queue statistics are for a single non-empty queue.

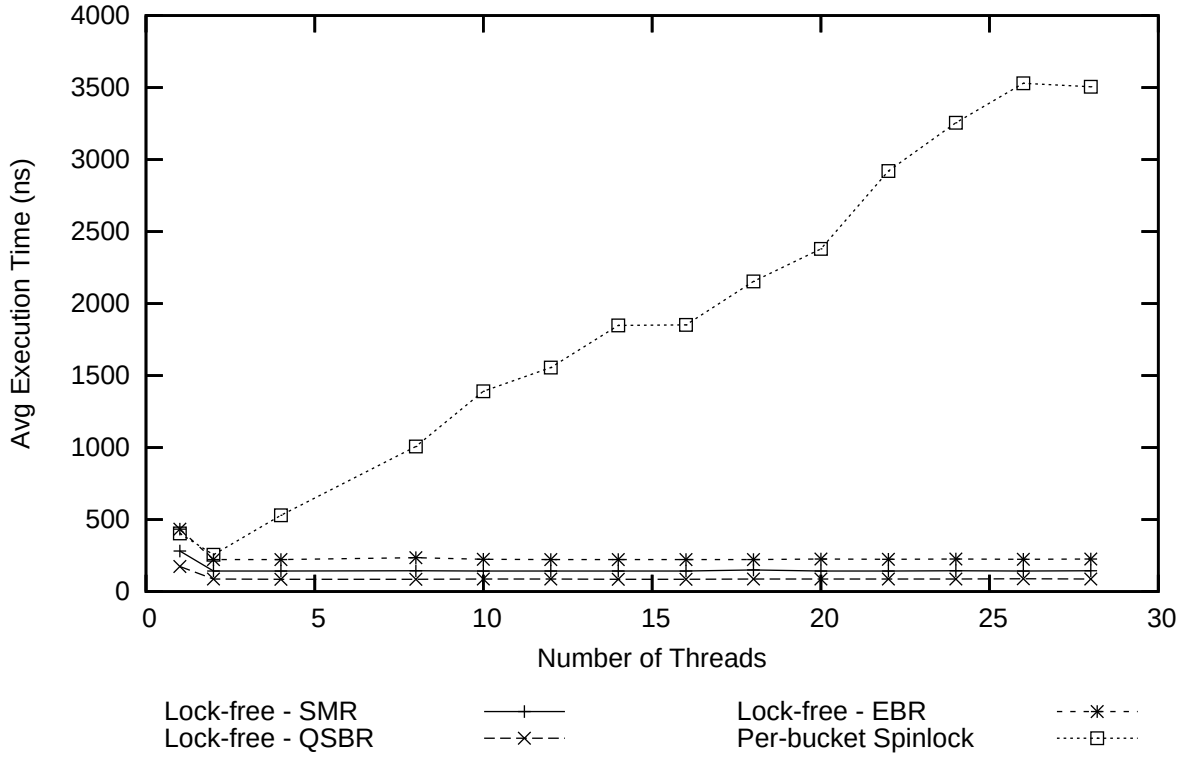

Figure 5.3: Hash table, 32 buckets, load factor 1, read-only workload. Spinlocks scale poorly as the number of threads increases.

Figure 5.2 shows the base costs of the concurrently-readable and lock-free algorithms under consideration when all these costs are low. The results are those for one thread, so that there is no contention of either type. The load factor of the hash tables is one. As predicted in Section 4.3, the fence instructions required make SMR and EBR more expensive than QSBR. The one exception is the write-only workload for the concurrently-readable hash table with SMR; this is because, in the write-only case, no hazard pointers need be used, so the overhead of using SMR is negligible.

We note that, in the base case, which memory reclamation scheme is most efficient seems to be determined solely by the per-operation fence instructions required.

We next show what is already well-established: contention for locks and the CPU seriously degrades the performance of lock-based algorithms. Figure 5.3 shows the per-

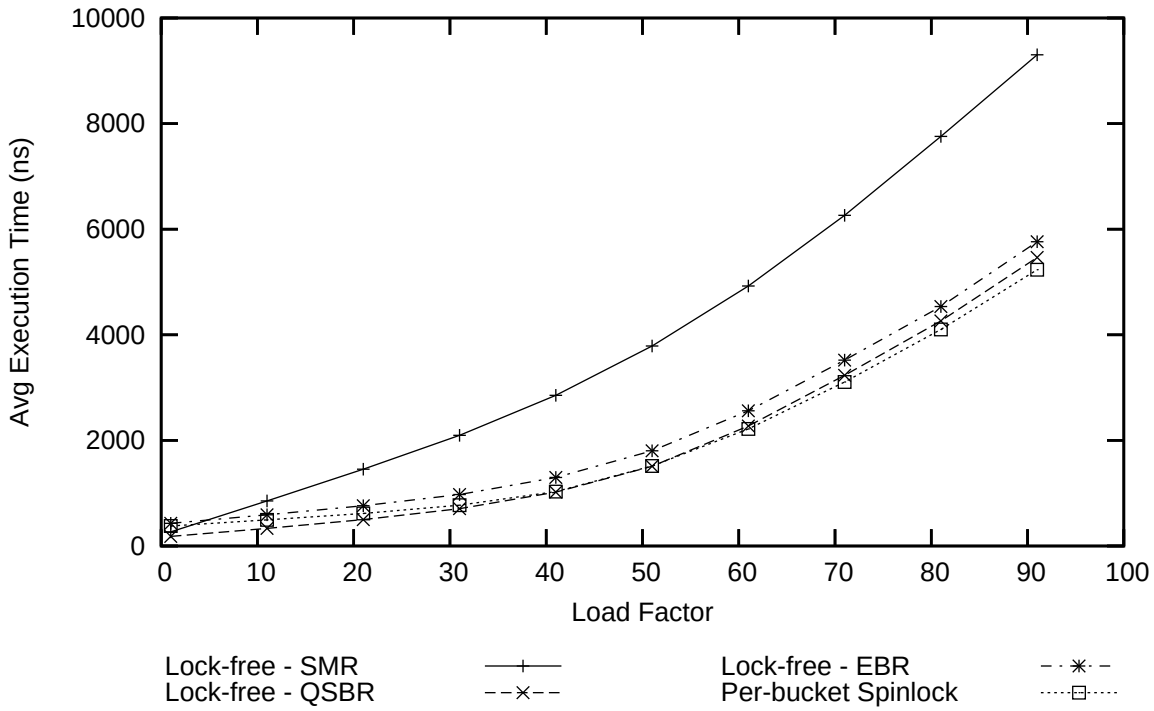

Figure 5.4: Hash table, 32 buckets, one thread, read-only workload, varying load factor.

formance degradation of spinlocks on the hash table as the number of concurrent threads increases, using a read-only workload and a load factor of 1. Since our tests were performed on a two-CPU machine, using more than two threads cannot increase our aggregate throughput; hence, horizontal lines on the graph indicate perfect scalability. Increasing the number of threads increases both the object contention, which makes threads more likely to spin while attempting to acquire a lock, and the CPU contention, which increases lock holder times. All memory reclamation schemes for the lock-free hash table scale equally well with these settings.

5.2.1 Effects of Traversal Length

In Figures 5.4 and 5.5, we show the effect of increasing the load factor of the hash table, when a single thread executes either a read-only or write-only workload, respectively.

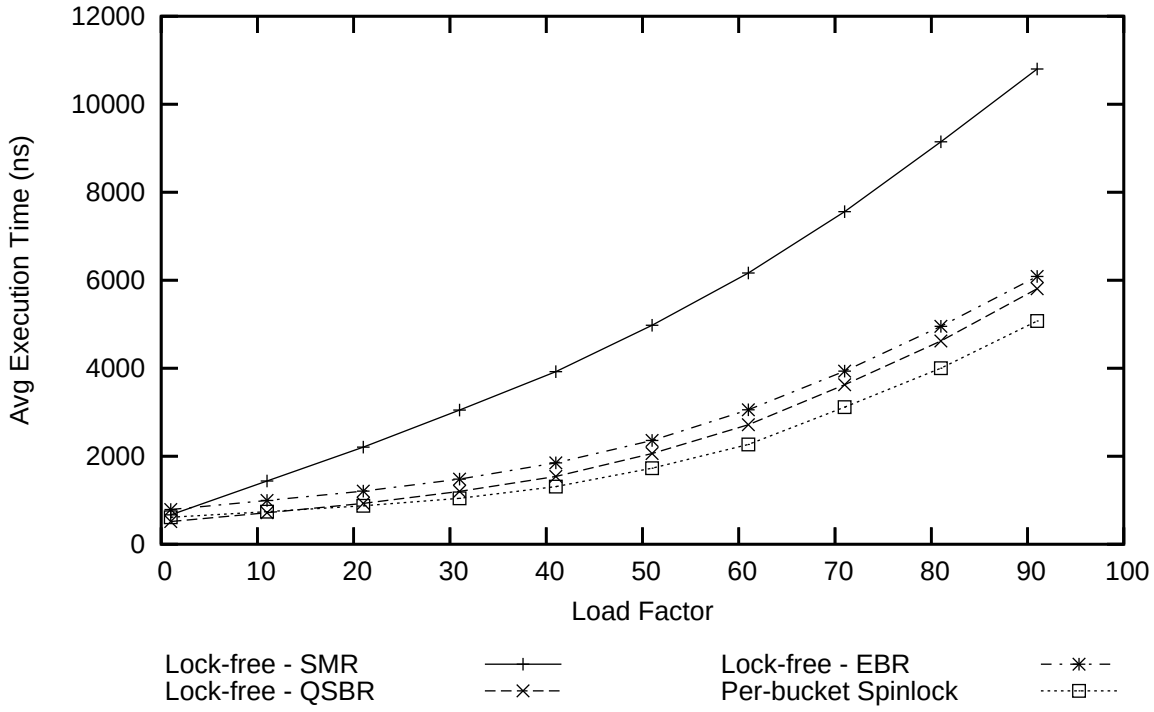

Figure 5.5: Hash table, 32 buckets, one thread, write-only workload, varying load factor.

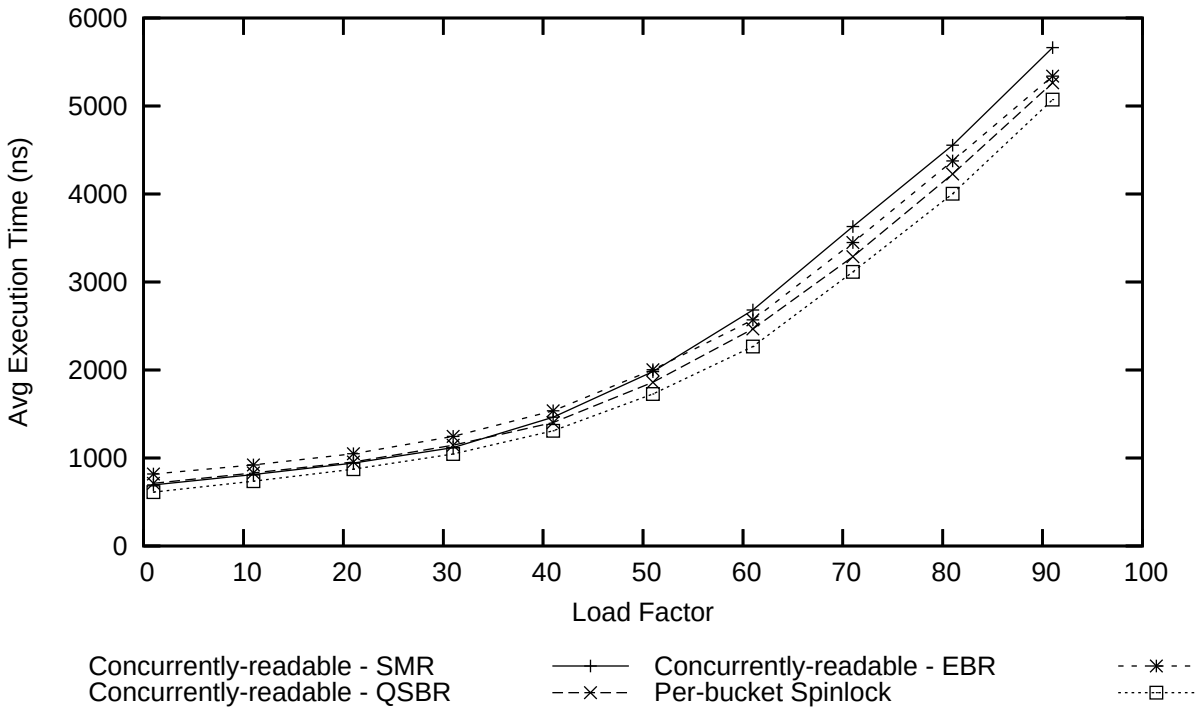

Figure 5.6: Hash table, 32 buckets, one thread, write-only workload, varying load factor.

Increasing the load factor effectively increases the traversal length for all operations, as even insertions first search for the key to be inserted in order to prevent duplicate entries. As predicted in Section 4.3, the per-node fence instructions ruin the performance of the lock-free hash table when using SMR: both the lock-based hash table and the lock-free hash table using QSBR or EBR severely out-perform it. Using SMR for memory reclamation for any structure requiring traversals of long chains of nodes, such as dynamic trees, is thus likely to be extremely expensive.

In read-mostly situations, the effect of increasing the load factor on the performance of the concurrently-readable algorithm using SMR is similar to its effect in the lock-free case. There is a difference, however, in write-mostly situations, as shown in Figure 5.6. The cause is the same as that of the anomaly in Figure 5.2: the concurrently-readable algorithm holds a lock for updates; hence, traversals done by updates do not suffer the

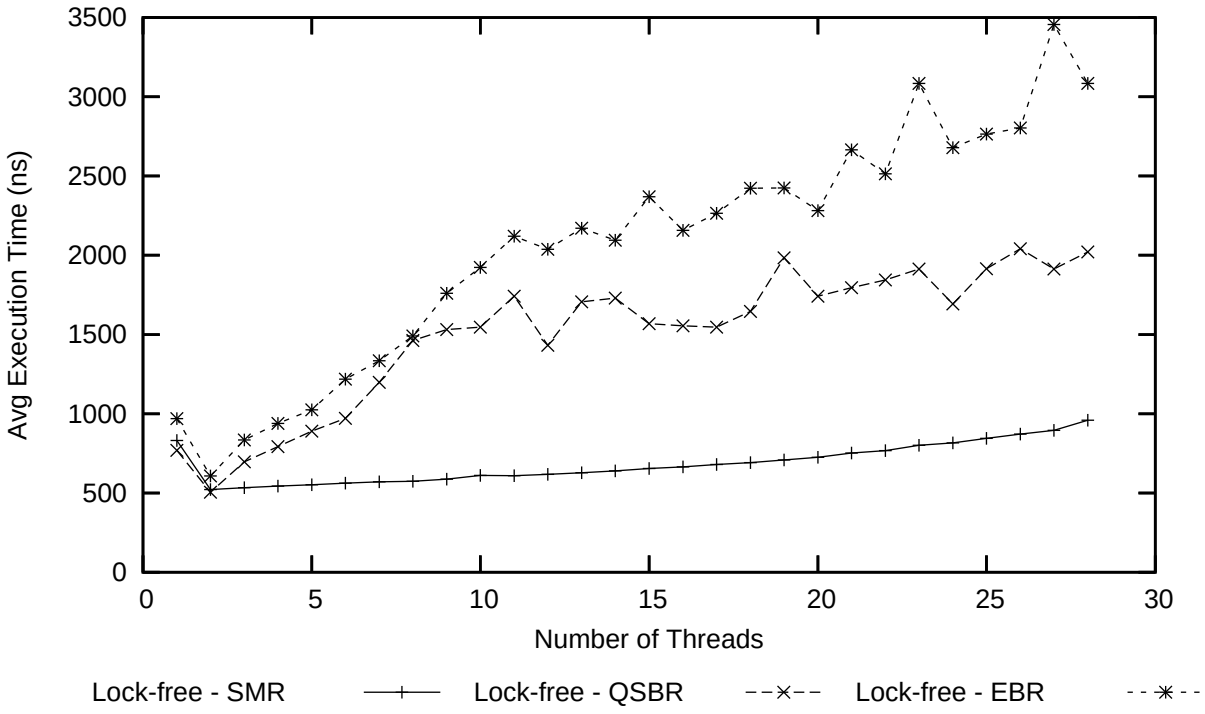

Figure 5.7: 100 queues, variable number of threads, Darwin/PPC.

per-node overhead of fence instructions, so the concurrently-readable algorithm's updates perform similarly regardless of reclamation method. This case is somewhat degenerate, however, since with a write-only workload, the concurrently-readable algorithm is equivalent to a naive per-bucket spinlock approach.

5.2.2 Effects of CPU Contention

QSBR and EBR scale well with load factor, and, as shown in Figure 5.3, they scale well with CPU contention under read-only workloads. However, Figure 5.7 demonstrates that they scale poorly with CPU contention when the workload involves a significant number of operations that must allocate memory. For this experiment, we focus on queues where every operation either allocates or deallocates memory. Although QSBR and EBR are hurt by CPU contention and allocation-heavy workloads on both systems, the magnitude of the effect depends on the execution environment. A similar, though less pronounced,

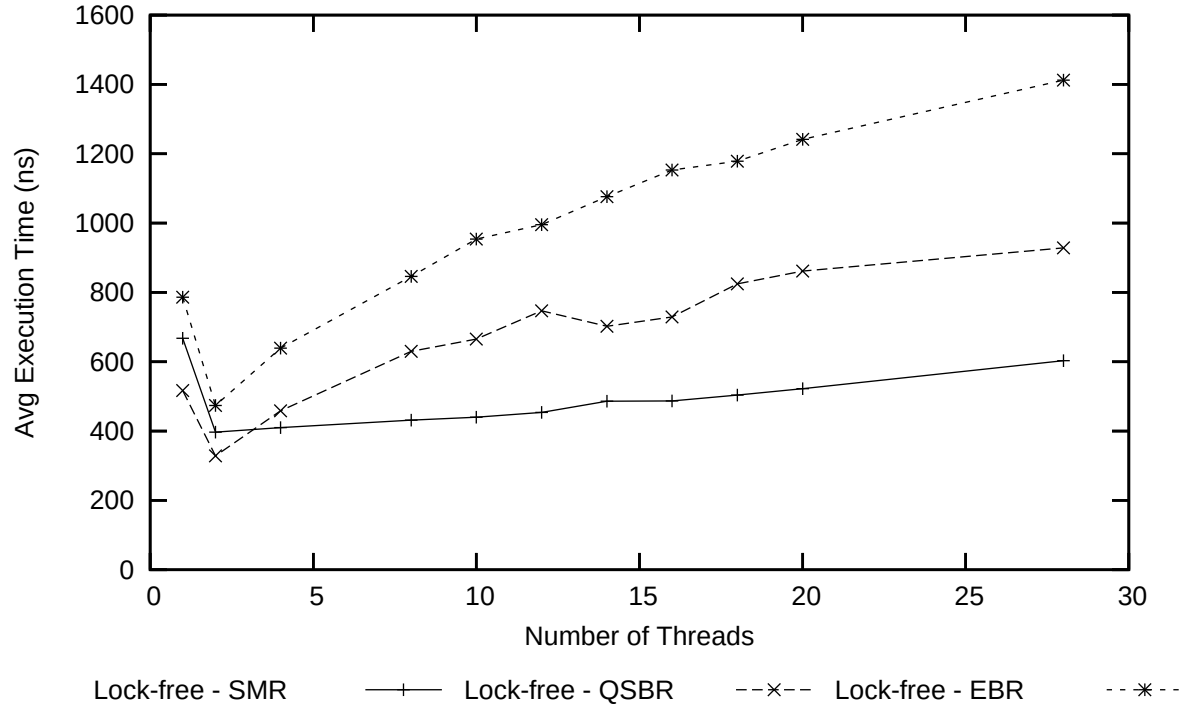

Figure 5.8: Hash table, 32 buckets, load factor 1, write-only workload, variable number of threads, Darwin/PPC.

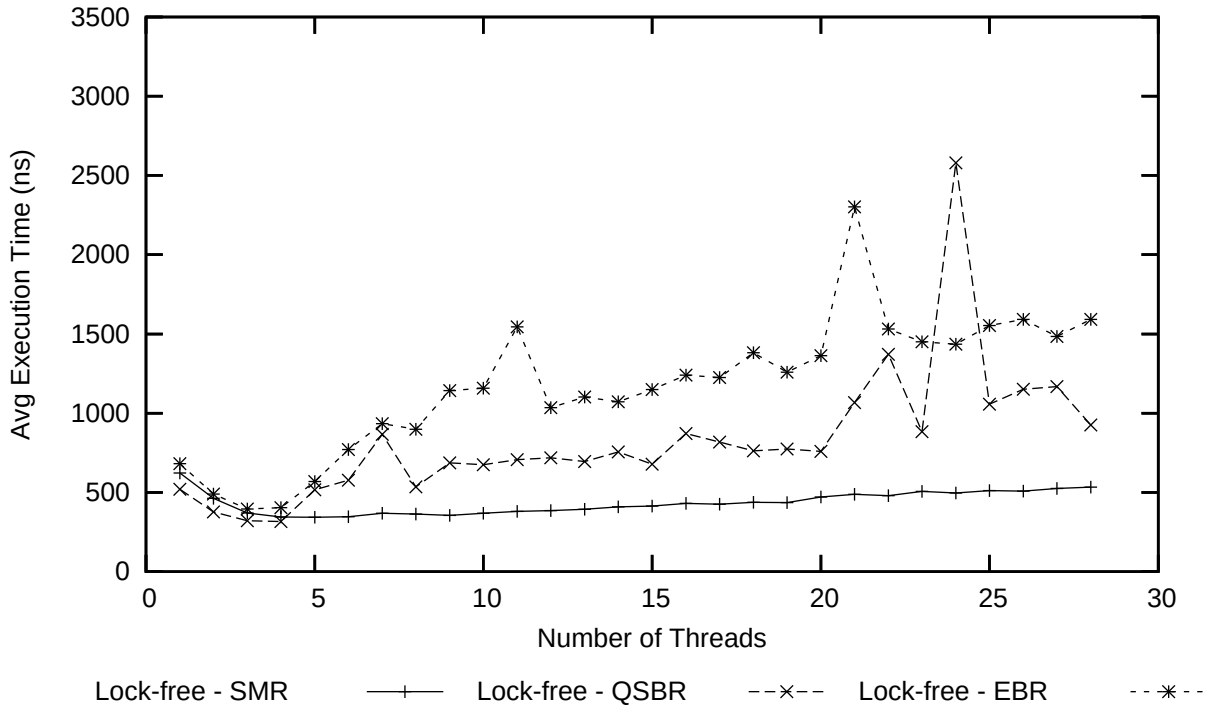

Figure 5.9: 100 queues, variable number of threads, Linux/IA-32.

effect can be observed for hash tables, as illustrated in Figure 5.8.

We predicted this scalability problem in Section 4.3. The extent to which this overhead seems to be scheduler-dependent and unpredictable is surprising, however. Figure 5.9 shows data from the same experiment as shown in Figure 5.7, but running on our Linux/IA-32 machine. Here, the shapes of the curves are quite different. Since this experiment involves significant multithreading, we believe that the differences are due to the different schedulers in the two kernels.

Although the increased contention for locks in our memory allocator and exhaustion of the memory pool play a part in QSBR and EBR's poor performance, we found while trying to mitigate these factors that we could not make these methods perform well when both CPU contention and memory allocation rates are high. Among other factors, we have found evidence that QSBR and EBR's inefficient use of memory interacts poorly with the OS's memory management strategies on the Darwin/PowerPC system. We are presently unable to analyze completely all the ways that QSBR and EBR interact with the external execution environment; however, it is clear from our results that delaying threads can have an profound impact on the performance of these two schemes since threads are prevented from physically deleting nodes. Further, as external factors come into play, it is extremely difficult for an application programmer to defend against these costs. SMR, in contrast, is largely immune to contention and bounds the memory usage, shielding the programmer from external concerns.

We note that, in our experiments, QSBR scales better than EBR with increasing numbers of threads. We view this as an artifact of our implementations. Our QSBR mechanism was designed specifically to minimize the per-grace period overhead of each thread, while Fraser's EBR scheme, which we adopted, was not.

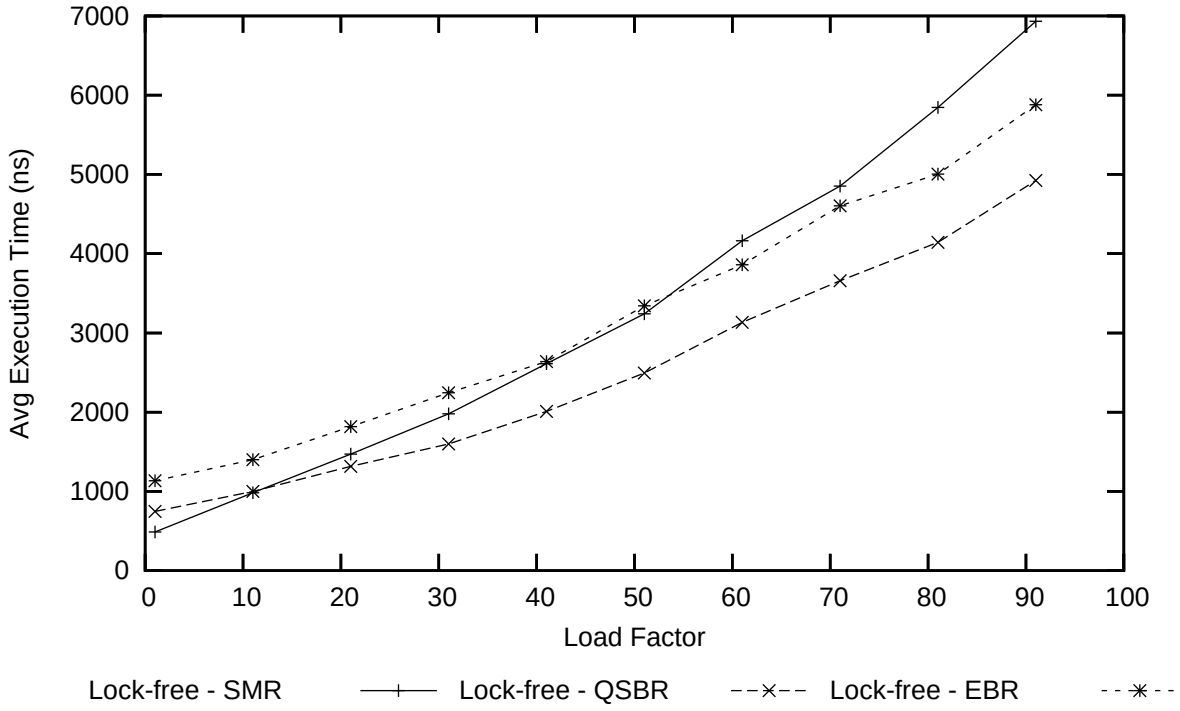

Figure 5.10: Hash table, 32 buckets, 16 threads, write-only workload, varying load factor.

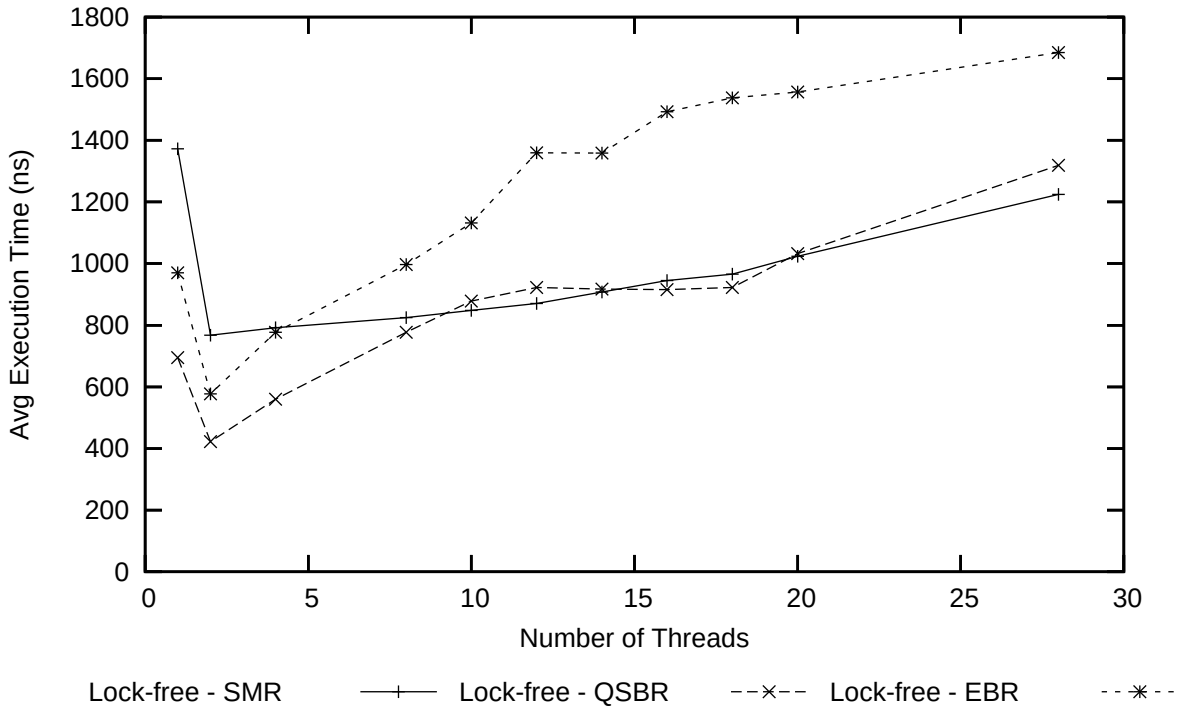

Figure 5.11: Hash table, 32 buckets, load factor 10, write-only workload, varying number of threads.

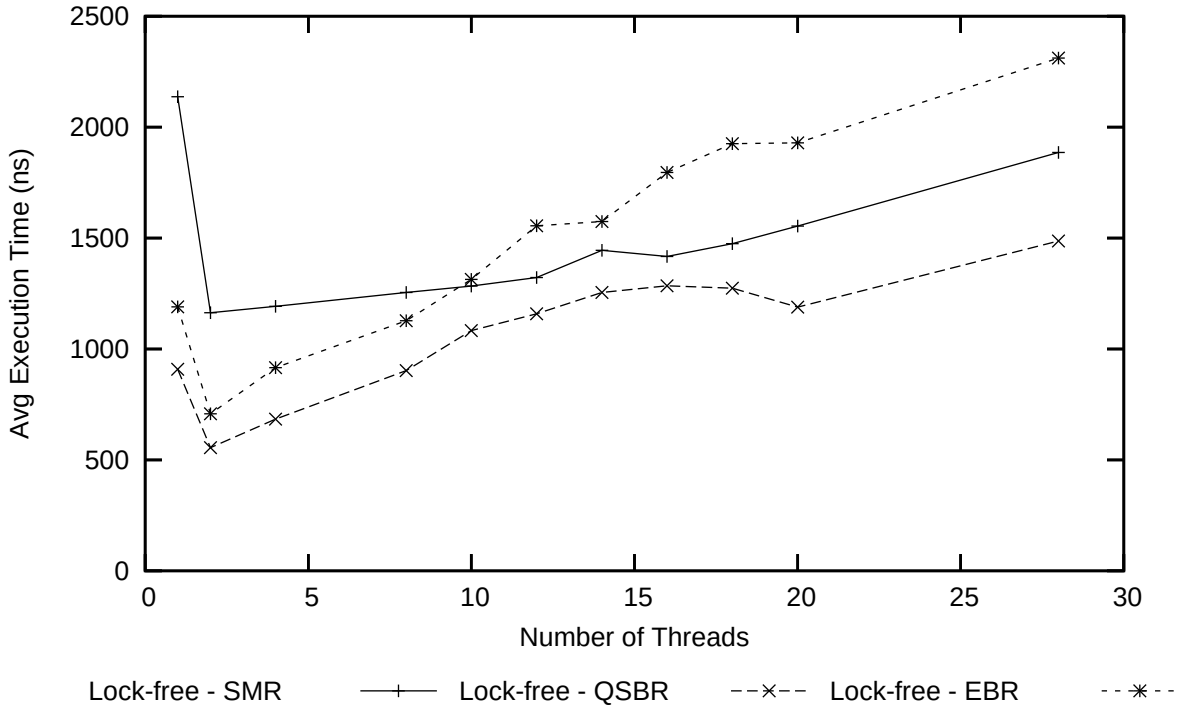

Figure 5.12: Hash table, 32 buckets, load factor 20, write-only workload, varying number of threads.

5.2.3 Relative Severity

We have seen that SMR's performance scales poorly with traversal length, while EBR and QSBR scale poorly for update-intensive workloads in the presence of CPU contention. In cases in which traversal length is high, the workload is update-heavy, and there is CPU contention, a programmer may wish to know which factor will influence the performance of memory reclamation schemes most severely.

Figures 5.10, 5.11, and 5.12 address this question. All three graphs show runs with a high load factor, CPU contention, and a write-only workload.

Figure 5.10 shows the effect of increasing load factor when we have 16 threads — eight threads per CPU. Even at this high level of CPU contention, QSBR begins to outperform SMR when the load factor exceeds 10, and EBR starts to beat SMR when the load factor exceeds 50.

Figure 5.11 shows the effect of increasing the number of threads when the load factor is 10. SMR begins to outperform EBR when there are more than four threads. SMR and QSBR are competitive when there are between 10 and 20 threads, and SMR begins to become superior when there are more than 20 threads. Figure 5.12 shows a similar plot, but with a load factor of 20. Here, SMR only begins to outperform EBR when we have more than 10 threads, and it does not outperform QSBR for any number of threads we tested.

Judging from Figures 5.10, 5.11, and 5.12, it appears that we need only a moderate load factor in order to make SMR perform poorly, but a very large amount of CPU contention with an update-intensive workload in order to make QSBR and EBR perform poorly. However, we urge caution in interpreting these results. First, the performance difference between our EBR implementation and our QSBR implementation show that the relative performance of memory reclamation schemes is very application-dependent. Second, in the case of QSBR, our experiments all had 100 operations per quiescent state; in real code, there may be many more operations per quiescent state, and therefore more

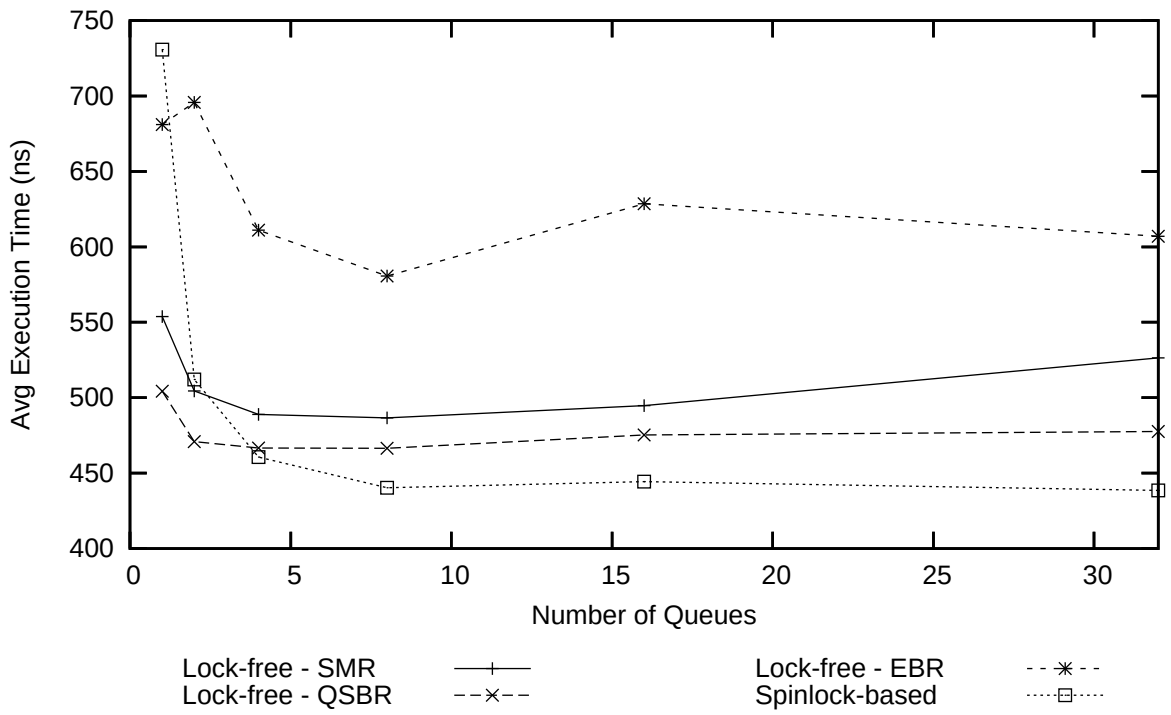

Figure 5.13: Queues, two threads, varying number of queues.

callbacks per grace period. Nevertheless, we hope that these results can give programmers some intuition as to the relative severity of factors affecting the performance of memory reclamation.

5.2.4 Low Overhead of QSBR

Based on our results so far, we find that QSBR is consistently the best-performing memory reclamation scheme, except when CPU contention combines with allocation-heavy workloads. This is further demonstrated in Figures 5.13 and 5.14, in which QSBR outperforms the other two memory reclamation strategies in all cases, often by a considerable margin. The left graph shows the effect of varying the number of queues, while the right graph shows the effect of varying the update fraction on the hash table.

The difference in performance is most pronounced when we consider hash tables. With two CPUs, one thread per CPU, and a load factor of five, the combination of the

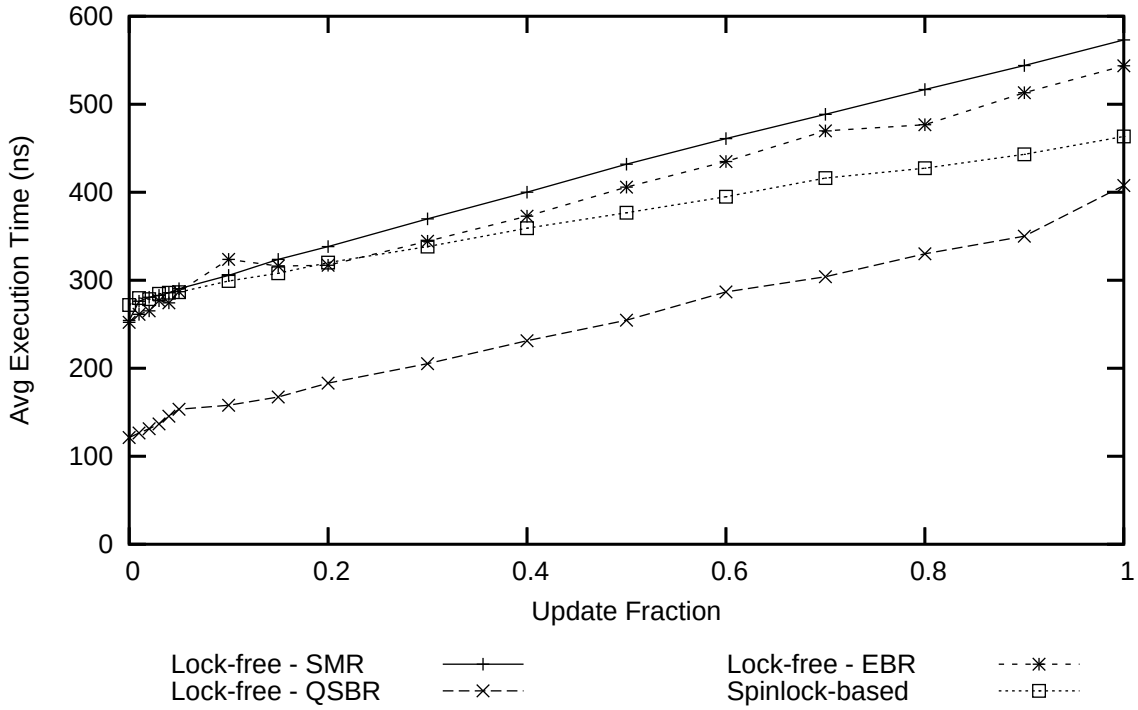

Figure 5.14: Hash table, 32 buckets, two threads, load factor 5, varying update fraction.

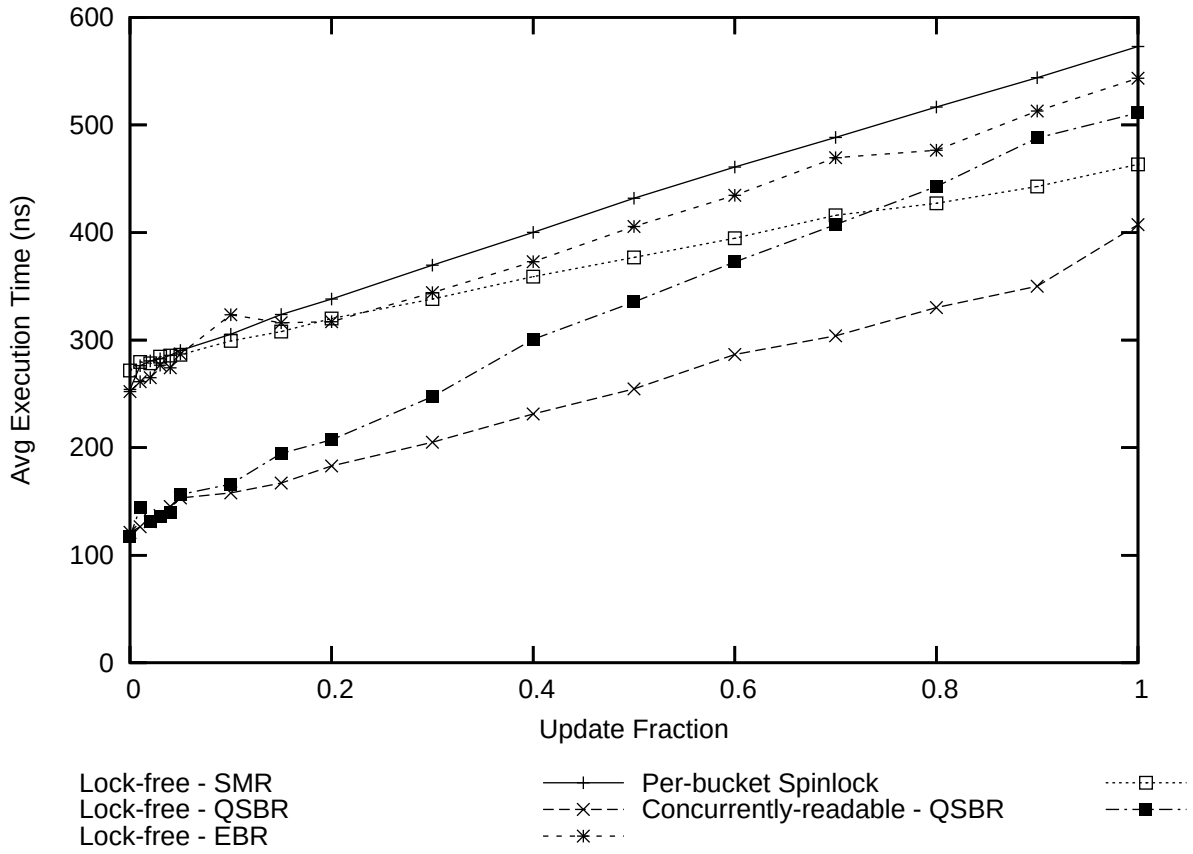

Figure 5.15: Hash table, 32 buckets, two threads, load factor 5, varying update fraction. QSBR allows the lock-free algorithm to out-perform RCU for almost any workload; neither SMR nor EBR achieve this.

lock-free algorithm with QSBR out-performs the lock-based alternative for any update fraction, while the lock-free algorithm with SMR or EBR fails to out-perform the lock-based version for almost all update fractions. Here, the choice of memory reclamation scheme clearly determines whether or not a lock-free algorithm can out-perform a lock-based one.

5.2.5 Lock-free Versus Concurrently-readable Linked List Algorithms

Using QSBR for both the concurrently-readable and lock-free linked list algorithms allows us to fairly compare their performance, and to see where each may be appropriate.

```
struct list_head *cur;

#define clean_pointer(p)       ((unsigned long)((p)) & (−2))

int search (struct list_head **head, long key)                                    5
{
        cur = *head;
        while (cur != NULL) {
                long ckey = (list_entry(cur, struct el, list))−>key;
                if (ckey >= key) {                                                10
                        return (ckey == key);
                }
                cur = clean_pointer(cur−>next);
        }
        return (0);                                                               15
}
```

Figure 5.16: Code for fast searches of lock-free list; compare to pseudocode of Figure 3.4.

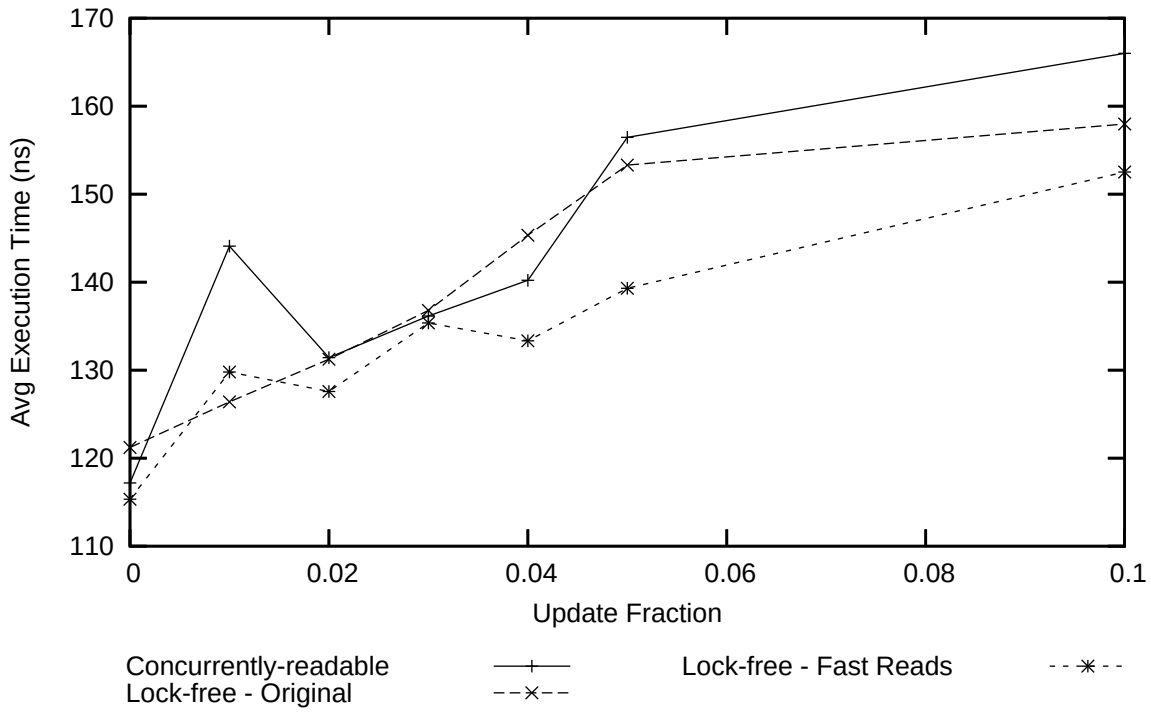

Figure 5.17: Hash table, 32 buckets, two threads, load factor 5, varying update fraction between 0 and 0.1.

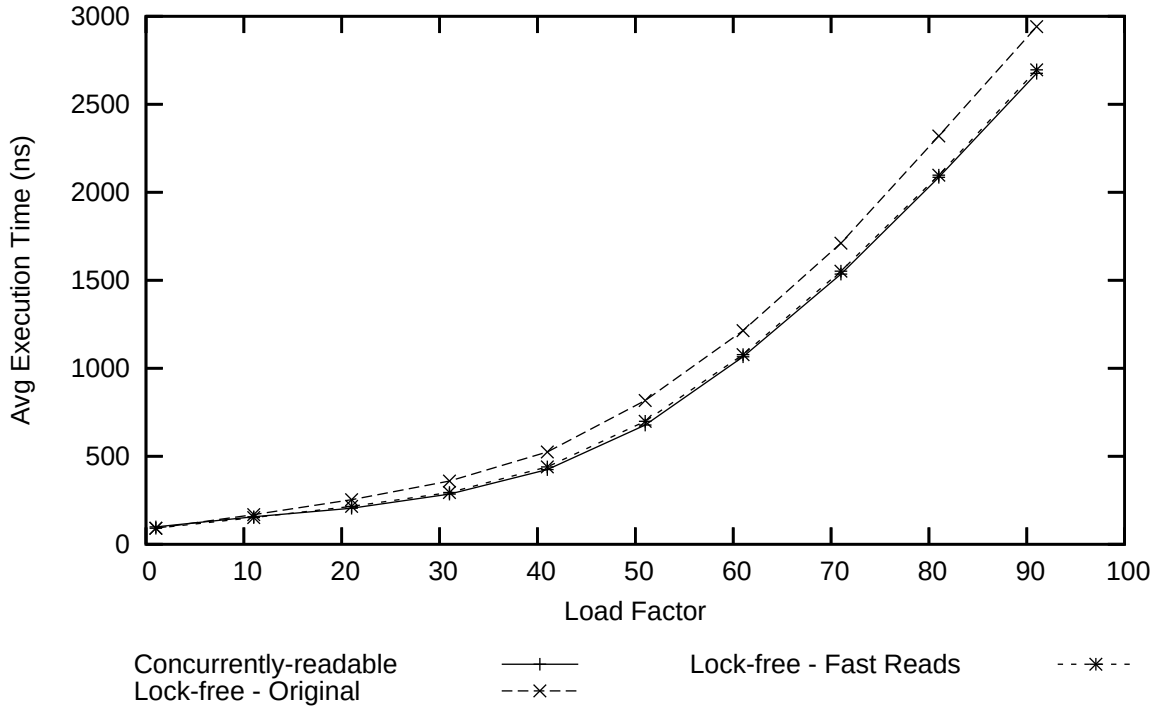

Figure 5.18: Hash table, 32 buckets, two threads, read-only workload, varying load factor.

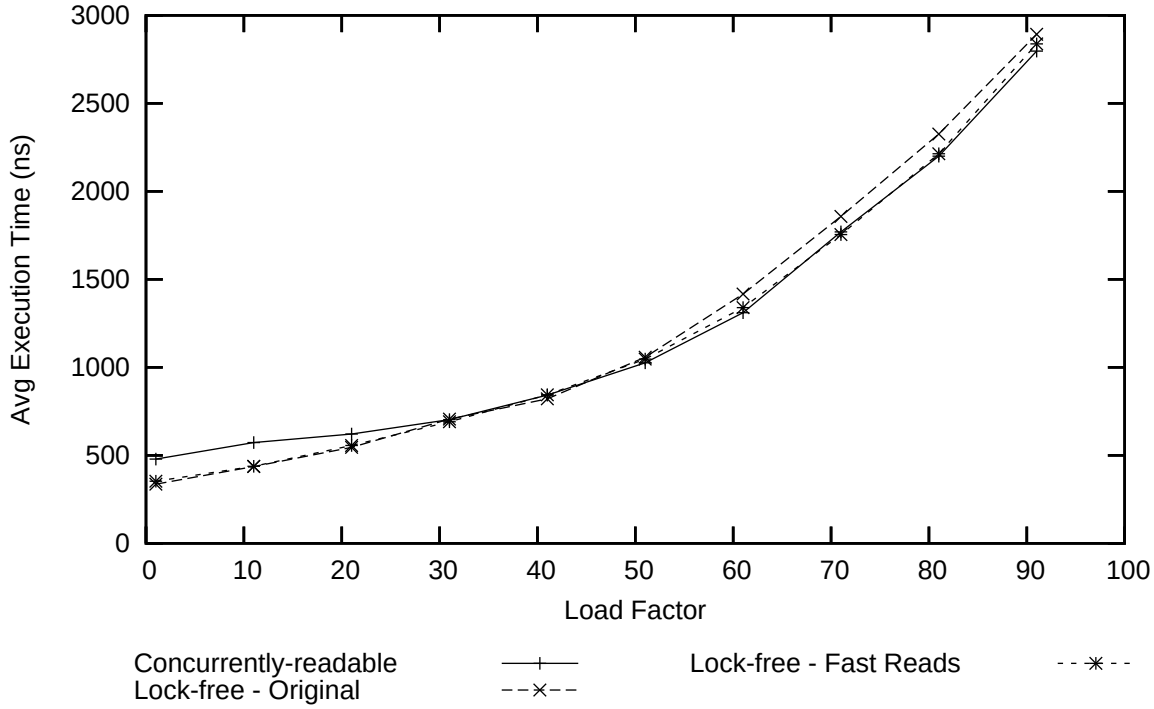

Figure 5.19: Hash table, 32 buckets, two threads, write-only workload, varying load factor.

The combination of the concurrently-readable linked list algorithm with QSBR is called read-copy update (RCU), which was mentioned in section 3.3 and is currently used in several OS kernels including Linux. RCU has extremely high performance for read-mostly workloads. We wanted to determine if, given efficient memory reclamation, our lock-free hash table could be suitable for use in OS kernels.

Figure 5.15 shows that the combination of the lock-free hash table with QSBR outperforms the concurrently-readable version for almost all update fractions. When the workload is nearly read-only, the concurrently-readable version performs slightly better than the lock-free one, since its concurrently-readable searches do not have the extra checks required of the lock-free searches. The lock-free algorithm has considerably cheaper updates, however, which allow it to scale much better as the update fraction increases. The lock-free algorithm's ability to perform competitively against the concurrently-readable one depends on our use of QSBR. If we use either SMR or EBR, the per-operation overhead of memory reclamation makes the lock-free algorithm inefficient.

Figure 5.17 zooms in on the range of update fractions between 0 and 0.1 of Figure 5.17. Here, we consider only the algorithms using QSBR; however, we add a new version of the lock-free algorithm which weakens the semantics of its reads (see Figure 5.16) to make it more competitive with the concurrently-readable algorithm. The design of these reads takes to heart the tenet of the RCU paradigm which seeks to minimize read-side synchronization [38]. These fast reads simply ignore nodes marked for deletion instead of helping to unlink them; hence, these reads may return nodes which have already been marked for deletion. These semantics are only slightly weaker than the original one, since in a concurrent programming environment, a read could find an undeleted node, and another thread could mark that node for deletion as it is returned to the application program.

These reads are used when the application program searches the hash table. The

deletion method uses this fast search on the first attempt to find the node to delete; if the deletion method must retry, it uses the original search code in order to ensure forward progress.

These faster reads have little impact on overall performance in Figure 5.17 — the performance of all three algorithms is very similar at low update fractions. This is because the load factor is low, so the per-node overhead of lock-free searches is very small, and is in the noise for the experiment.

The fast reads are more important at higher load factors. Figures 5.18 and 5.19 show the performance of these three algorithms on a read-only and write-only workload, respectively, as the load factor increases. The per-node overhead of the extra checks becomes significant for the original version of the lock-free algorithm. The modified version, however, remains competitive with the concurrently-readable version, even at very high load factors.

The result that, when using QSBR, the lock-free algorithm can outperform thee concurrently-readable one when the load factor is low and the update fraction is above 0.1 may have practical implications. The lock-free algorithm also outperforms per-bucket spinlocking for any update fraction, while the concurrently-readable algorithm does not. This lock-free algorithm may find a niche in kernels which use RCU, for use with update-heavy structures. Investigating this further is a topic for future work.

5.2.6 Summary of Recommendations

Given the results examined in the previous sections, we are able to provide some rules of thumb for choosing between competing algorithms and memory reclamation strategies.

We have seen that SMR performs poorly when long chains of nodes must be traversed, and that EBR and QSBR perform poorly when there is CPU contention and an allocation-heavy workload. Of the two factors, traversal length seems to have the greater effect. QSBR has the lowest base overhead, while EBR has the most. EBR's only advan-

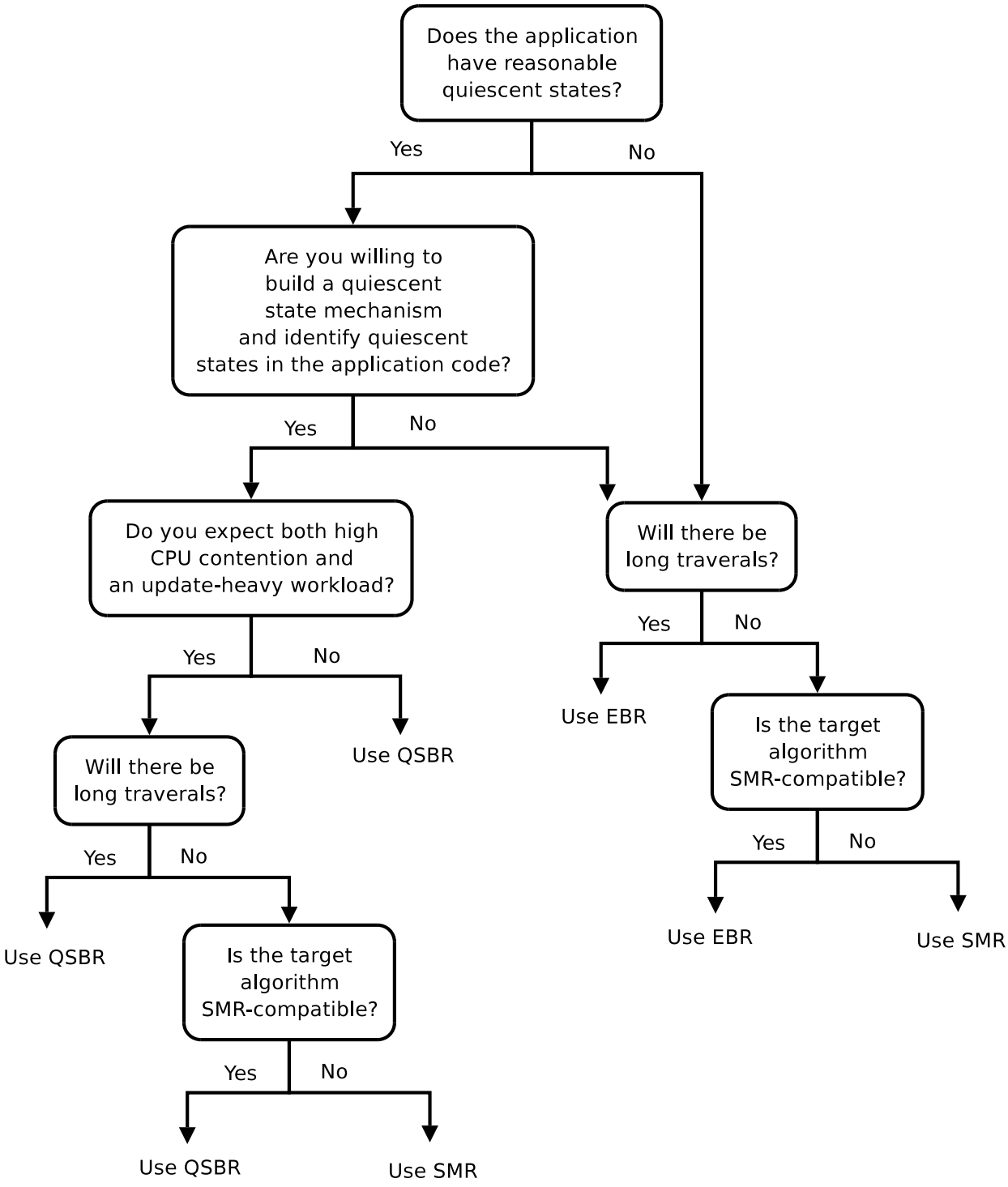

Figure 5.20: Decision tree for choosing a memory reclamation scheme.

tage over QSBR is that it is application-independent; in some cases, this advantage may make EBR preferable. Figure 5.20 presents a decision tree to help programmers choose a memory reclamation scheme.

Choosing between the lock-free and concurrently-readable algorithms is simpler. If there is CPU contention, we can expect the lock-free algorithm to perform much better. Otherwise, the lock-free algorithm will perform better if the update fraction is significant, and the traversal length is not so long that the per-node traversal overhead becomes significant. However, if operations other than insertions, deletions, and searches are required, it is likely to be easier to add them to the concurrently-readable list than the lock-free one. Figure 5.21 shows a decision tree for choosing between the two algorithms.

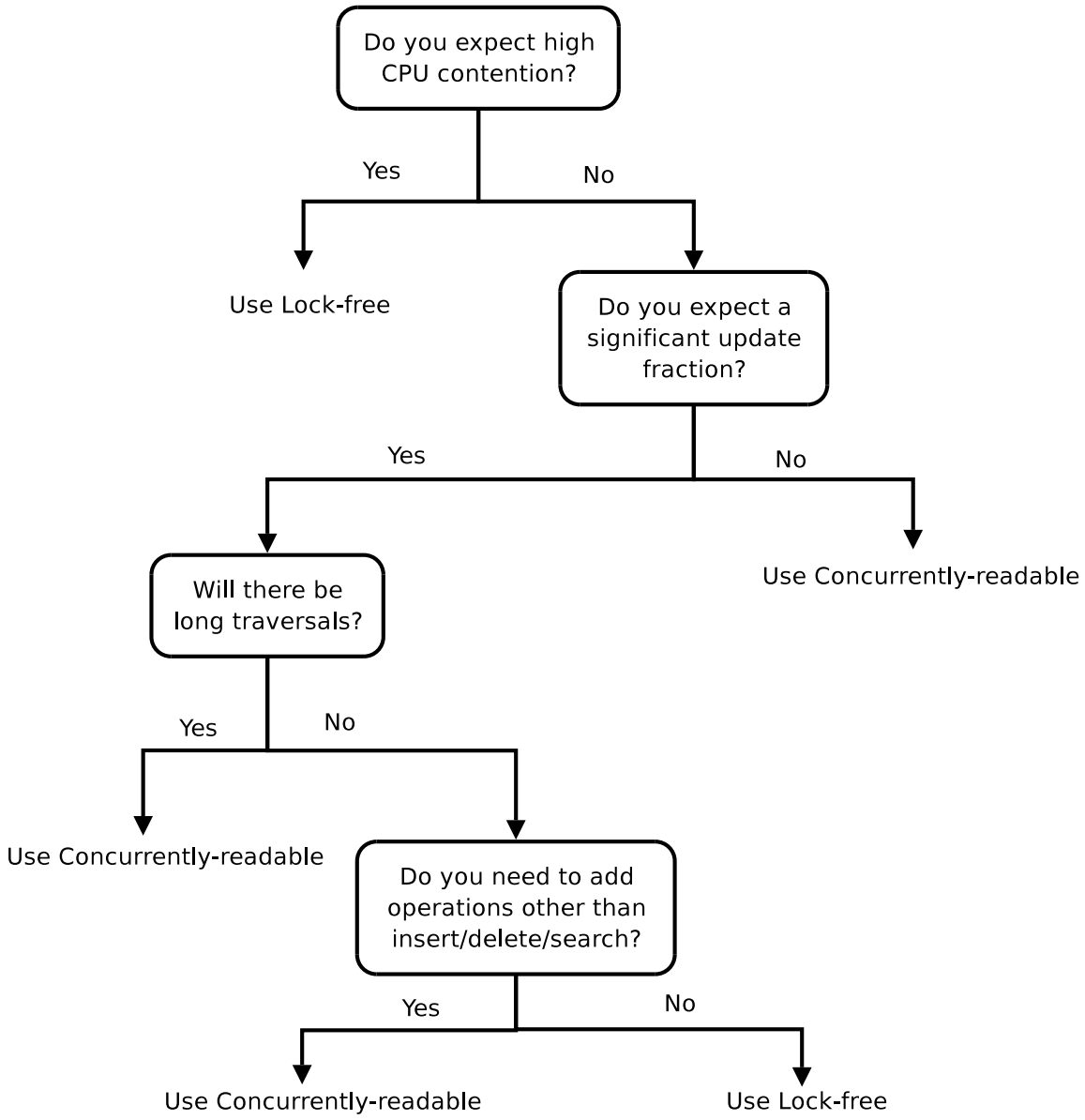

Figure 5.21: Decision tree for choosing whether to use the lock-free or concurrently-readable linked list algorithm.

Chapter 6

Related Work

Our contribution lies not in introducing new memory reclamation strategies, but in providing a comprehensive analysis of their relationship and their relative merits, and in using QSBR with lock-free queues. This work is, to our knowledge, the first to pair QSBR with an object that does not have read-only operations.

6.1 Blocking Memory Reclamation for Non-blocking Algorithms

Others have proposed that memory reclamation strategies that are not lock-free should sometimes be used in combination with lock-free algorithms. Fraser [17] noted, but did not thoroughly evaluate, the performance overhead of SMR due to the fence instructions it requires, and used EBR instead. Our work continues Fraser's in showing that EBR itself has high overhead — often higher than that of SMR — and that more efficient memory reclamation is often possible. We view our work and Fraser's as being part of a trend towards weakenings of lock-freedom, such as obstruction-freedom [29] and almost non-blocking data structures [8], designed to preserve the advantageous properties of lock-freedom while improving performance.

We are not the first to use QSBR with lock-free algorithms — Auslander implemented a lock-free hash table with QSBR [38] in the K42 operating system [5, 18]. However, pairing QSBR with hash tables does not fully separate it from the RCU paradigm, since hash tables have read-only operations. Thus, it does not evaluate QSBR for use with more general lock-free objects as we have done by pairing it with a lock-free queue. Furthermore, no performance evaluation, either between different memory reclamation methods or between concurrently-readable and lock-free hash tables, was provided in [38].

In response to Auslander's work, McKenney [38] posed several questions concerning the use of RCU with lock-free synchronization; among them were:

- Whether using QSBR will make non-blocking synchronization more broadly applicable.
- With which non-blocking algorithms it makes sense to use QSBR.
- What merits using QSBR with non-blocking synchronization has, relative to other techniques.
- How various combinations of QSBR and non-blocking synchronization compare to one another, both empirically and analytically.

Although it is difficult for experimental research to answer all aspects of McKenney's questions, our work addresses them significantly. First, our results show that while in many situations using QSBR can improve the performance of a lock-free algorithm, it is not a silver bullet. QSBR therefore makes non-blocking synchronization more feasible in many environments, but locking may still be preferable in many situations (see Figure 5.13).

Second, our good performance results from pairing QSBR with queues indicate that it makes sense to use QSBR with many lock-free algorithms. In particular, pairing QSBR

with queues shows that using QSBR makes sense even when the target data structure has no read-only operations.

Third, the results of our performance analysis show quite clearly the advantages and disadvantages of using QSBR relative to other memory reclamation schemes. QSBR has the lowest base time overhead of any memory reclamation scheme we are aware of, and its overhead does not grow when the traversal length increases. SMR, by contrast, has significant per-node overhead, so its cost grows linearly as the traversal length increases. The only disadvantage of QSBR with regard to performance is that it performs poorly for update-intensive workloads when CPU contention is high. In all other situations, QSBR is the clear winner in terms of performance.

Last, our results also provide a comprehensive comparison of two combinations of QSBR and non-blocking synchronization. Our analysis and experiments both show that, in the base case, the performance differences between different reclamation schemes come from the per-operation overhead due to expensive operations such as fences, and not due to the complexity of periodic reclamation routines. Also, as noted above, Auslander's implementation of a lock-free hash table using QSBR [38] left a need for an analysis of such combinations, which our work has addressed.

6.2 Vulnerabilities of Blocking Memory Reclamation Schemes

Michael [47] criticized QSBR for its unbounded memory use. No evaluation was given of the impact of this limitation on the performance of lock-free algorithms using QSBR. Sarma and McKenney [56], however, have shown that this leads to the possibility of denial-of-service attacks, and that preventing these attacks becomes an engineering problem.

The denial-of-service attacks noted by Sarma and McKenney occur in the IPV4 route

cache of the Linux 2.5.53 kernel when a large number of softirqs create a correspondingly large number of pending deletion callbacks, which cause the route cache to overflow. This vulnerability and the poor results we saw for QSBR when we combined an update-heavy workload with high CPU contention are instances of a more general problem: when we get too many callbacks, they then stress parts of the systems beyond the limits for which they were designed. Our results show that EBR is similarly vulnerable.

6.3 Performance Comparisons

Several comparisons of different QSBR implementations have been made [40, 41]. These comparisons, however, have not compared QSBR to other memory reclamation schemes. In addition, these implementations have all been made in the context of operating system kernels, and have not tested QSBR under conditions of CPU contention.

Michael [47] compared the performance of SMR to that of reference counting, and found that SMR's performance is much better; however, he did not compare SMR to any other memory reclamation schemes. Furthermore, Michael's experiments did not show the effect of increasing traversal length, which we show is an important weakness of SMR. Finally, Michael neither discussed nor evaluated the performance tradeoffs involved between blocking and lock-free memory reclamation schemes.

Fraser [17] criticized SMR for its overhead due to fence instructions, and cited this overhead as a reason for using his EBR scheme instead. We have shown that EBR itself has high overhead due to fences (Figure 5.2); in fact, in the base case for all algorithms, EBR has more overhead than SMR. Further, all Fraser's experiments were run with fewer threads than CPUs. This setup does not evaluate EBR's performance in the presence of CPU contention and an allocation-heavy workload, which we have shown is one of EBR's major weaknesses.

Although each of these three memory reclamation schemes has been evaluated by its

respective creator, in all cases, these evaluations either ignore these competing methods or criticize them with little or no experimental evaluation. Furthermore, these evaluations have been set up in such a way that they do not expose the weaknesses of the schemes. Our contribution lies not in introducing new memory reclamation schemes, but in providing, to our knowledge, the first fair comparison of these proposals.

Similarly, many publications in the area of lock-free algorithms compare the performance of these algorithms to lock-based alternatives [23, 47, 50, 17, 43]. Similarly, comparisons have been made between concurrently-readable algorithms and more traditional locking approaches [39, 42, 38]. Ours, however is, to our knowledge, the first attempt to compare the performance of a lock-free algorithm to a concurrently-readable alternative.

Chapter 7

Conclusions and Future Work

This thesis has made three main contributions:

- We have shown that memory reclamation schemes can be chosen mostly independently of the target algorithms.

- We have analyzed the strengths and weaknesses of three memory reclamation schemes.

- Using a common memory reclamation scheme, we have made a fair comparison of lock-free and concurrently-readable chaining hash tables.

Establishing that memory reclamation schemes are mostly independent of the algorithms which use them helps to clarify much of the current literature. In particular, much of the RCU literature presents QSBR and the concurrently-readable algorithms which use it in a tightly-coupled manner. Furthermore, QSBR is typically explained in terms of its implementation in operating system kernels. Our work establishes the independence of the QSBR from the algorithms it services and its kernel-level implementations, and shows that it is useful for a wider variety of algorithms than those presented in the RCU literature. Hopefully, this realization will help others to find new uses for QSBR.

Table 7.1: Properties of Memory Reclamation Schemes

	QSBR	**SMR**	**EBR**
Base Time Overhead	Negligible	Unbounded	Constant
Memory Use	Unbounded	Bounded	Unbounded
CPU Contention Scalability	Poor When Update-Heavy	Good	Poor When Update-Heavy
Traversal Length Scalability	Good	Poor	Good
Application-dependent?	Yes	No	No

We have demonstrated, by comparing the performance of EBR, SMR, and QSBR, that the choice of memory reclamation scheme has a huge effect on the performance of lock-free and concurrently-readable algorithms. Choosing the right scheme for the environment in which an implementation is expected to run is essential. No memory reclamation scheme provides a silver bullet — the trade-offs between the schemes are shown in Table 7.1.

SMR and EBR have a higher base cost than QSBR because of the fence instructions they require. For EBR, the overhead due to fences is constant, while for SMR it is unbounded.

Our results show that when there is no CPU contention, or the workload involves few updates, QSBR is the best-performing memory reclamation scheme available. QSBR and EBR scale poorly when there is CPU contention and many updates, since they wait for other threads to complete their operations, thus allowing descheduled threads to delay memory reclamation.

Our comparison of the lock-free and concurrently-readable linked lists shows that the lock-free list has substantially cheaper updates, but pays a small amount of per-node overhead. This per-node overhead becomes significant at high load factors. A modified version of the lock-free list reduces this overhead considerably. When the load factor is suitably low, the lock-free hash table seems to be the best performer for most update

fractions.

Given the extremely low overhead of QSBR, experimenting with user-level QSBR implementations in a realistic application would be an interesting topic for future work. In particular, it would be interesting to build a QSBR implementation and interface which works with Pthreads or a lock-free library such as NOBLE [1]. Another topic would be to experiment with applying QSBR to a wider range of data structures and with transactional-memory-based implementations such as those of Fraser [17], whose experimental setup is available under the GPL [16].

The good performance of the lock-free hash table with QSBR is encouraging. Presently, the Linux kernel provides kernel programmers with a concurrently-readable hash table with QSBR. Given that hash tables typically have very low load factors, the per-node overhead of the lock-free hash table is unlikely to be problematic. Furthermore, a similar lock-free hash table is already part of K42 [38]. Building a lock-free hash table in Linux, and, more importantly, determining which parts of the kernel would benefit from its cheaper writes or the advantages of lock-freedom, would be another avenue for future work.

Finally, we note that SMR, Pass the Buck, and reference counting — the three lock-free memory reclamation methods — all have overhead that grows with traversal length. In the case of SMR and Pass the Buck, this overhead comes from fence instructions, while reference counting's overhead also comes from expensive operations such as CAS. This opens the question of whether high per-node overhead is an inherent downside of lock-free memory reclamation in weakly consistent memory models; this question could be answered by designing a general-purpose lock-free memory reclamation scheme with significantly less per-node overhead which scales nicely with traversal length, or proving that no such method can exist.

Bibliography

[1] Noble - a library of non-blocking synchronization protocols. `http://www.noble-library.org`.

[2] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 125–134. ACM Press, 1992.

[3] James H. Anderson and Mark Moir. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 168–182. Springer-Verlag, 1995.

[4] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM Press, 1995.

[5] Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Experience with K42, an open-source, linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[6] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In

Proceedings of the USENIX Annual Technical Conference (FREENIX Track), pages 297–310. USENIX Association, June 2003.

[7] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th Annual ACM symposium on Parallel Algorithms and Architectures*, pages 261–270. ACM Press, 1993.

[8] Hans-J. Boehm. An almost non-blocking stack. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 40–49, 2004.

[9] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference, General Track 2001*, pages 15–33, 2001.

[10] David L. Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73. Springer-Verlag, 2000.

[11] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, 2001.

[12] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[13] 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 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 216–224. ACM Press, 2004.

[14] Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–39. ACM Press, 2004.

[15] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–59. ACM Press, 2004.

[16] Keir Fraser. Lock-free library. Source code release. Available: `http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free/src/lockfree-lib.tar.gz`.

[17] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[18] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[19] Michael Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999.

[20] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136. ACM Press, 1996.

[21] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle

Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314. Springer-Verlag, 2001.

[23] Timothy L. Harris, Keir Fraser, and Ian Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the IEEE Symposium on Distributed Computing*, October 2002.

[24] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 206–215. ACM Press, 2004.

[25] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[26] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[27] Maurice Herlihy, Victor Luchangco, and Mark Moir. Brief announcement: Dynamic-sized lock-free data structures. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, 2002.

[28] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Symposium on Distributed Computing*, October 2002.

[29] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522. IEEE Computer Society, 2003.

[30] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, July 2003.

[31] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, 1980.

[32] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[33] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[34] Doug Lea. Concurrency: where to draw the lines. Invitational Workshop on the Future of Virtual Execution Environments, September 2004. Available: `http://www.research.ibm.com/vee04/Lea.pdf`.

[35] Robert Love. *Linux Kernel Development*. Sam's Publishing, 2004.

[36] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 268–282. ACM Press, 1982.

[37] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, June 1991.

[38] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[39] Paul E. McKenney. RCU vs. locking performance on different CPUs. In *linux.conf.au*, Adelaide, Australia, January 2004.

[40] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001.

[41] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.

[42] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[43] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium Parallel Algorithms and Architectures*, pages 73–82, August 2002.

[44] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.

[45] Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *The 9th Euro-Par Conference on Parallel Processing*, volume 2790, pages 651–660, August 2003.

[46] Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM T. J. Watson Research Center, January 2004.

[47] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[48] Maged M. Michael. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. In *The 18th International Conference on Distributed Computing*, October 2004.

[49] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.

[50] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, Computer Science Department, University of Rochester, December 1995.

[51] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM Press, 1996.

[52] Mark Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319. Springer-Verlag, 1997.

[53] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, Department of Computer Science, University of Maryland, June 1990.

[54] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[55] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[56] Dipankar Sarma and Paul E. McKenney. Issues with selected scalability features of the 2.6 kernel. In *Ottawa Linux Symposium*, July 2004.

[57] Håkan Sundell. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, April 2003.

[58] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.

[59] Håkan Sundell. Wait-free reference counting and memory management. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, April 2005.

[60] R. Kent Treiber. Systems programming: Coping with parallelism. Research Report RJ 5118, IBM Almaden Research Center, April 1986.

[61] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.