Efficient Memory Reclamation is Necessary for Fast Lock-free Data Structures*

Thomas E. Hart
Department of Computer Science
University of Toronto

Paul E. McKenney
Linux Technology Center and Storage Software Architecture
IBM Beaverton

Angela Demke Brown
Department of Computer Science
University of Toronto

March 7, 2005

Technical Report CSRG-515
Department of Computer Science, University of Toronto

Abstract

Despite their advantages, lock-free algorithms are often not adopted in practice, partly due to a common belief 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 silver bullet for memory reclamation for lock-free algorithms.

*A version of this paper was submitted to PODC 2005.

Keywords: Lock-free, non-blocking, concurrently-readable, linked list, hash table, memory reclamation, hazard pointers, read-copy update, quiescent-state-based reclamation, safe memory reclamation, epoch-based reclamation, limbo lists.

1 Introduction

Concurrent applications require a means to coordinate accesses to shared data structures. Traditional 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 [16, 8, 20, 21]. Some recent lock-free algorithms provide an attractive alternative, as they have been shown to have lower overhead than their lock-based counterparts, even when contention for objects and the CPUs is low [18]. However, we demonstrate that using an inefficient memory reclamation scheme can squander this performance advantage.

Memory reclamation is required for all dynamic lock-free 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 algorithm. Reference counting [23, 6], for example, has high overhead in the base case, and scales poorly in structures for which long chains of elements must be traversed [18]. Little work has been done on comparing different memory reclamation strategies; we address this deficiency in this paper, 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 algorithms in choosing an appropriate scheme for their specific applications.

Current memory reclamation strategies suggested for lock-free data structures [17, 18, 9, 7, 5] 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) [16, 14, 8, 2, 13, 15], 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. We show by example that this scheme 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 remainder of this paper is organized as follows. Sections 2 and 3 provide background. In Section 4, we make an analytic comparison of the memory reclamation methods. Section 5 describes the experiment we used to validate our analysis, and Section 6 describes our experiment's results. We discuss related work in Section 7, and conclude with Section 8.

2 Memory Consistency Models

Current literature on lock-free algorithms generally assumes a sequentially-consistent [11] 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 [7].

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[1]:

- 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 $A <_m B$.
- 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

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

reclamation strategies considered depend almost entirely on the number of fences they require.

3 Memory Reclamation Schemes

This section reviews 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, and explain why we did not consider this method. Since all these memory reclamation methods have been published elsewhere [16, 2, 17, 18, 7], we discuss them in only enough detail for the reader to understand our work.

3.1 Reference Counting

Implementations of lock-free reference counting have been proposed by Valois [23] and by Detlefs et al. [5, 6]. 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. 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 previously shown by Michael [18] 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.

3.2 Epoch-Based Reclamation

Epoch-based reclamation (EBR) was introduced by Fraser [7], but builds on earlier ideas. 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 is illustrated in Figure 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 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.

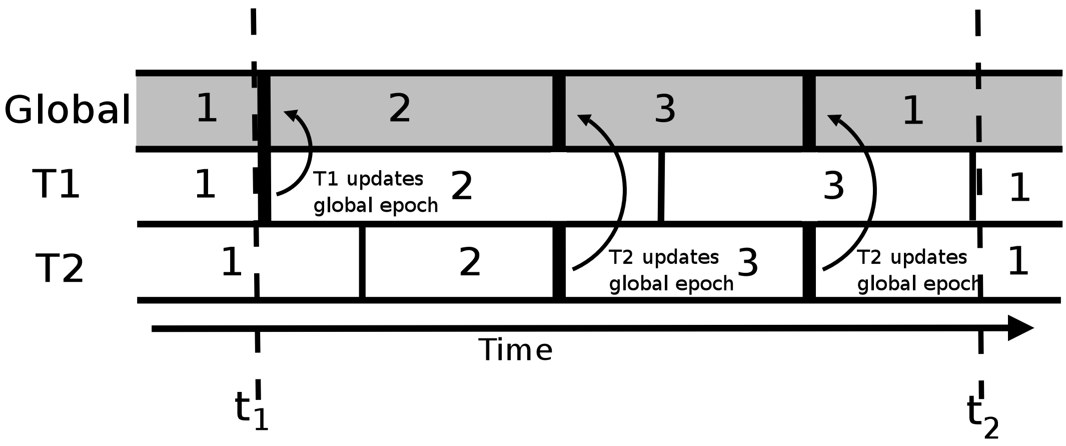

Figure 1: Illustration of EBR. Threads follow the global epoch. If a thread observes that all other threads have seen the current epoch, then it may update the global epoch. Hence, if the global epoch is e, threads in critical sections can be in either epoch $e+1$ or e, but not $e-1$ (all mod 3). The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

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 be equal to 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 last time 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.

3.3 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

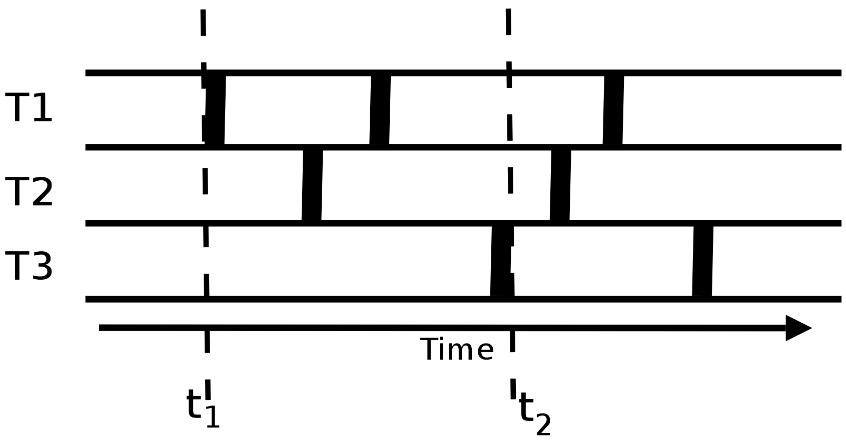

Figure 2: Illustration of QSBR. Thick lines represent quiescent states. The time interval $[t_1, t_2]$ is a grace period for thread $T1$: at time t_2, threads $T2$ and $T3$ have each passed through a quiescent state since t_1, so all nodes logically removed before time t_1 can be physically deleted.

least one quiescent state. QSBR is illustrated in Figure 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 — a domain in which quiescent-state-based reclamation is used to implement *read-copy update* (RCU) [16, 15, 8].

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 6, this gives QSBR a significant performance advantage over EBR.

3.4 Safe Memory Reclamation

Safe memory reclamation was introduced by Michael [17]. It provides a simple and intuitive existence locking mechanism for dynamic nodes. Each thread which accesses a lock-free 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 data-structure-dependent. K 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.

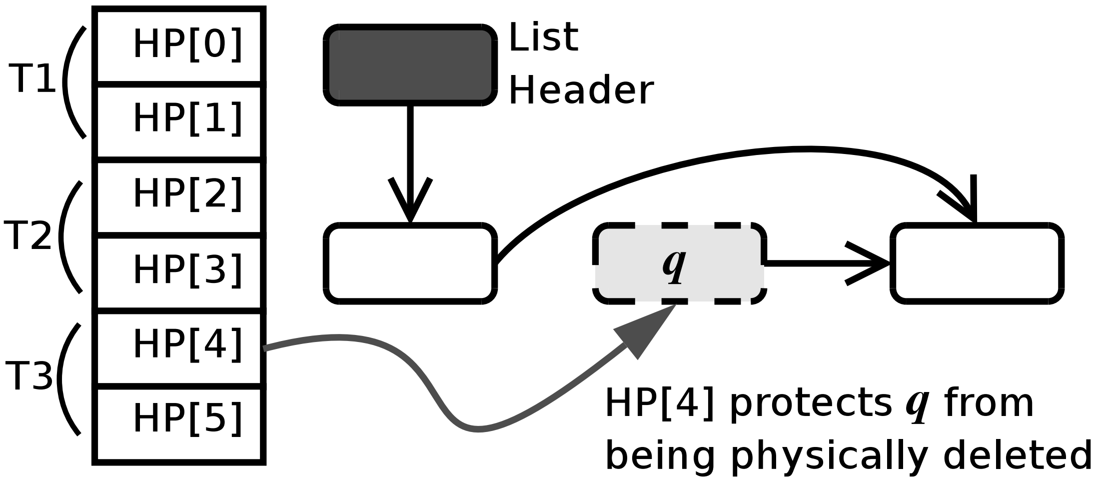

Figure 3: Illustration of SMR. q is logically removed from the linked list on the right of the diagram, but cannot be physically deleted while $T3$'s hazard pointer $HP[4]$ is still pointing to it.

When a thread T removes a node from a dynamic data structure, it places 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 $R = H + \Omega(H)$. [2]

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) [18] — in all lock-free operations. Before using any such reference, a hazard pointer must be made to point to the target object. After setting the hazard pointer, the algorithm must perform an algorithm-specific check to ensure that the object has not been deleted; the rules of the SMR deletion routine outlined above guarantee that the object will never be deleted

[2]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 making R a positive integer multiple of H, plus some optional constant, suffices.

so long as the hazard pointer continues to point to it. Figure 3 illustrates the use of SMR.

4 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 6. 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. 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 some 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 6, 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

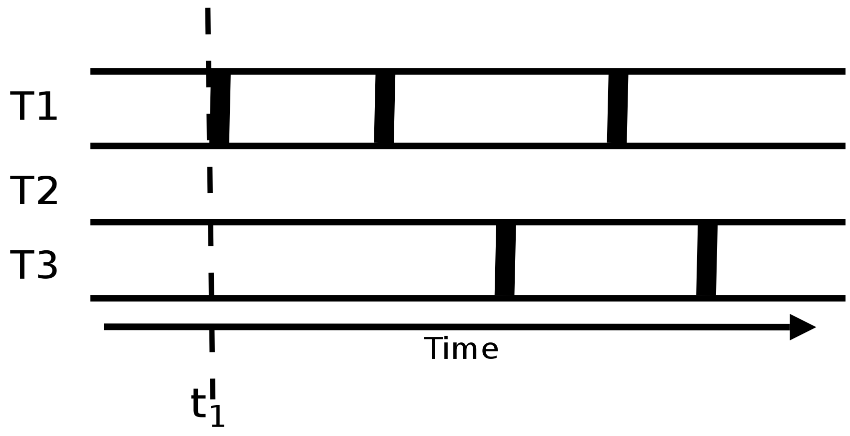

Figure 4: Illustration of why QSBR is inherently blocking. Here, thread $T2$ does not go through a quiescent state for a long period of execution time; hence, threads $T1$ and $T3$ must wait to reclaim memory. A similar argument holds for EBR.

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 element, 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 affect QSBR and EBR. Descheduled threads can delay other threads' memory reclamation, as shown in Figure 4. In extreme cases, this could lead to out-of-memory errors, which could cause threads to block on 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 [19], 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 [17]; we thus expect it should be less sensitive to the memory allocator and other factors in the external environment than QSBR and EBR.

5 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. This section provides details on these aspects of our experiments.

5.1 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 [18], which 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 a version that uses locks only for updates, which we call *concurrently-readable* [12]. The lock-free queue is the version of Michael and Scott's implementation given in [18]. 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.

```
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: 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.

5.2 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 the loop is begun again if the parent thread's timer has not yet expired, as shown in Figure 5. 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 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 elements 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, 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 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 $R = 2H + 100$.

Table 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

5.3 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 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 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 1000 elements 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 [4]. All nodes are pre-allocated by the parent before the test begins.

5.4 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

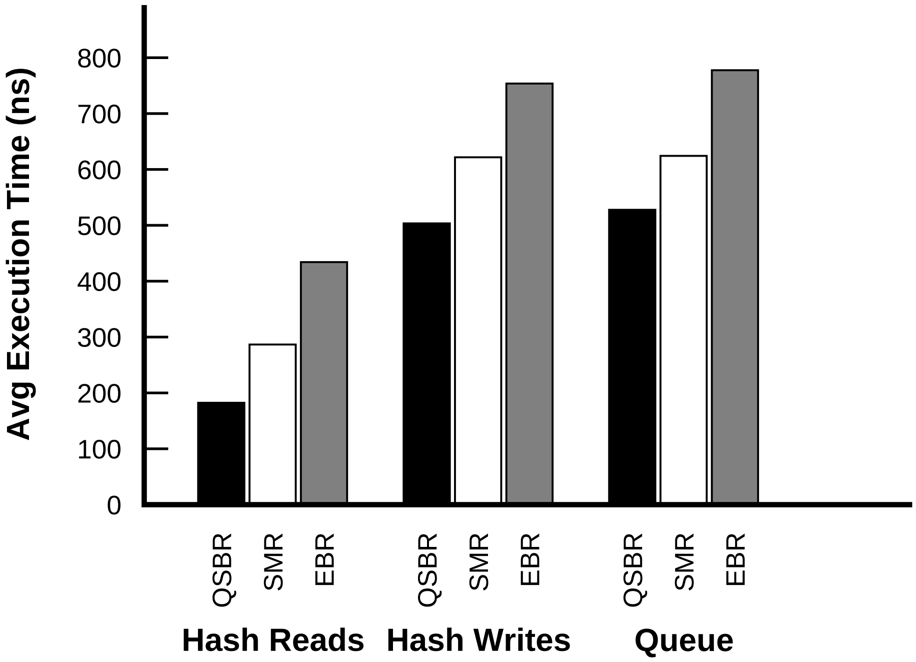

Figure 6: Single-threaded memory reclamation costs on PowerPC. Hash table statistics are for a 32-bucket hash table with a load factor of 1. Queue statistics are for a single non-empty queue.

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.

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.

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

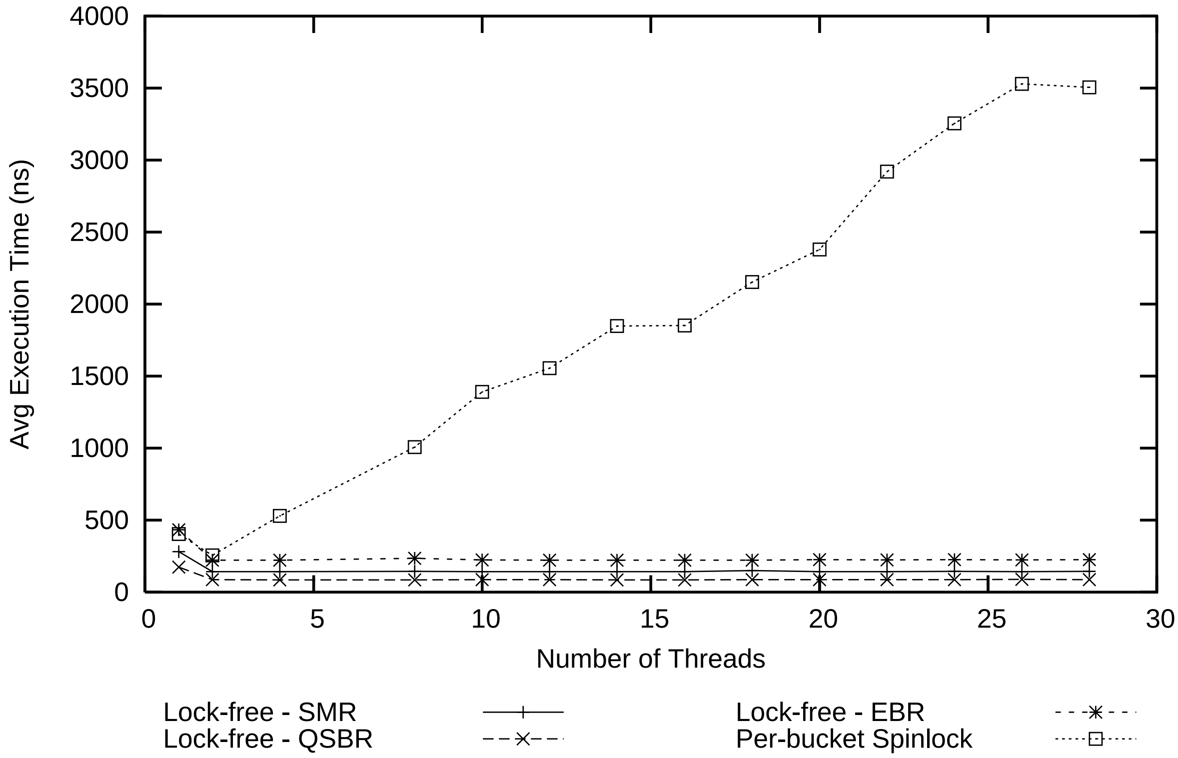

Figure 7: Hash table, 32 buckets, load factor 1, read-only workload. Spinlocks scale poorly as the number of threads increases.

6 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 6 shows the base costs of the lock-based 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, the fence instructions required make SMR and EBR more expensive than QSBR.

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 7 shows the performance 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.

6.1 Effects of Traversal Length

In Figure 8, we show the effect of increasing the load factor on the hash table, when a single thread executes either a read-only (left graph) or a write-only (right graph) workload. Increasing the load factor effectively increases the traversal length for all operations, as even insertions first search for the key. As predicted in Section 4, the per-element 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 of any structure requiring traversals of long chains of elements, such as dynamic trees, is thus likely to be extremely expensive.

6.2 Effects of CPU Contention

QSBR and EBR scale well with load factor, and, as shown in Figure 7, they scale well with CPU contention under read-only workloads. However, Figure 9 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 deletes memory, and show data from both the PowerPC and the IA-32 based systems. 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, effect can be observed for hash tables; the graphs are omitted due to space limitations.

We predicted this scalability problem in Section 4. The extent to which this overhead seems to be scheduler-dependent and unpredictable is surprising, however. 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 extreme 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

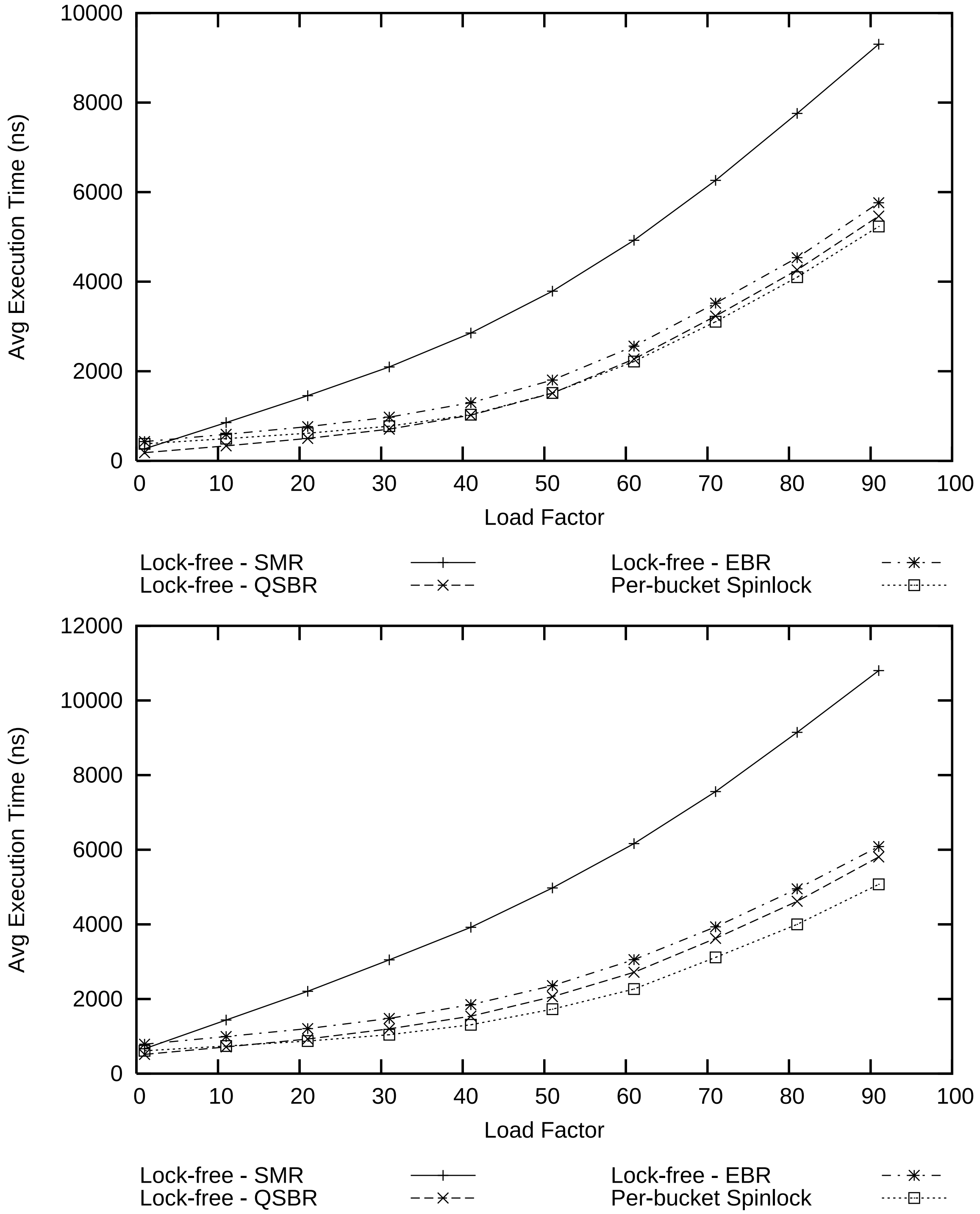

Figure 8: Hash table, 32 buckets, one thread, load factor 1, showing both read-only (left) and write-only (right) workloads. SMR scales poorly as the load factor increases.

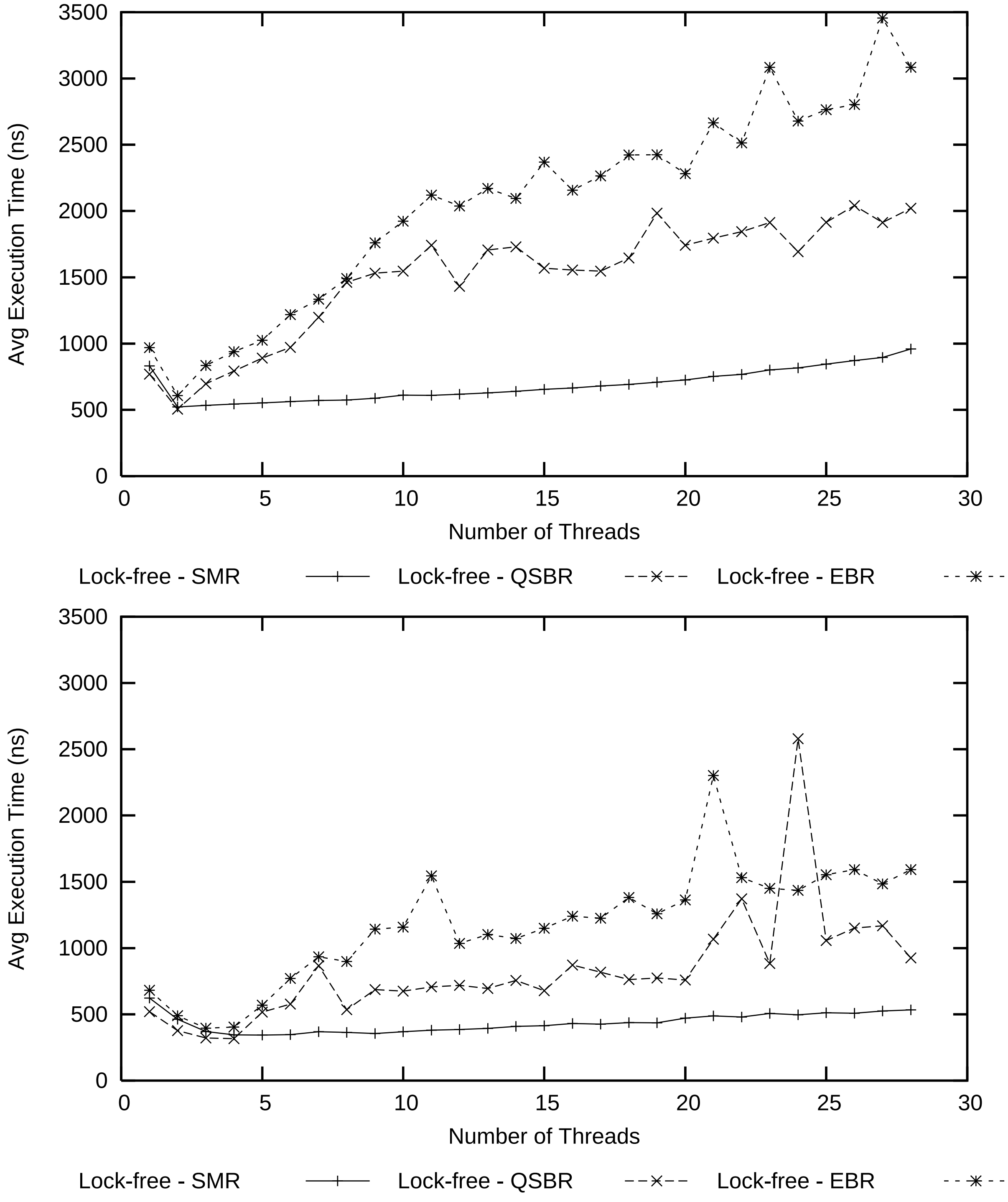

Figure 9: 100 queues, variable number of threads. Left shows Darwin/PPC, right shows Linux/IA-32. High CPU contention combined with a workload involving many memory allocations causes QSBR and EBR to perform poorly, but the exact interactions seem scheduler-dependent.

contrast, is largely immune to contention and bounds the memory usage, shielding the programmer from external concerns.

6.3 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 Figure 10, in which QSBR out-performs 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 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.

6.4 An Application to OS Kernels

Read-copy update (RCU), as currently used in several OS kernels including Linux, combines a concurrently-readable linked list or chaining hash table with QSBR. RCU has extremely high performance for read-mostly workloads. We refer to this combination as *Concurrently Readable* QSBR. We wanted to determine if, given efficient memory reclamation, our lock-free hash table could be suitable for use in OS kernels.

Figure 11 shows that the combination of the lock-free hash table with QSBR out-performs 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.

This result may have practical implications. The lock-free hash table with QSBR performs well relative to spinlocks for any update fraction. 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.

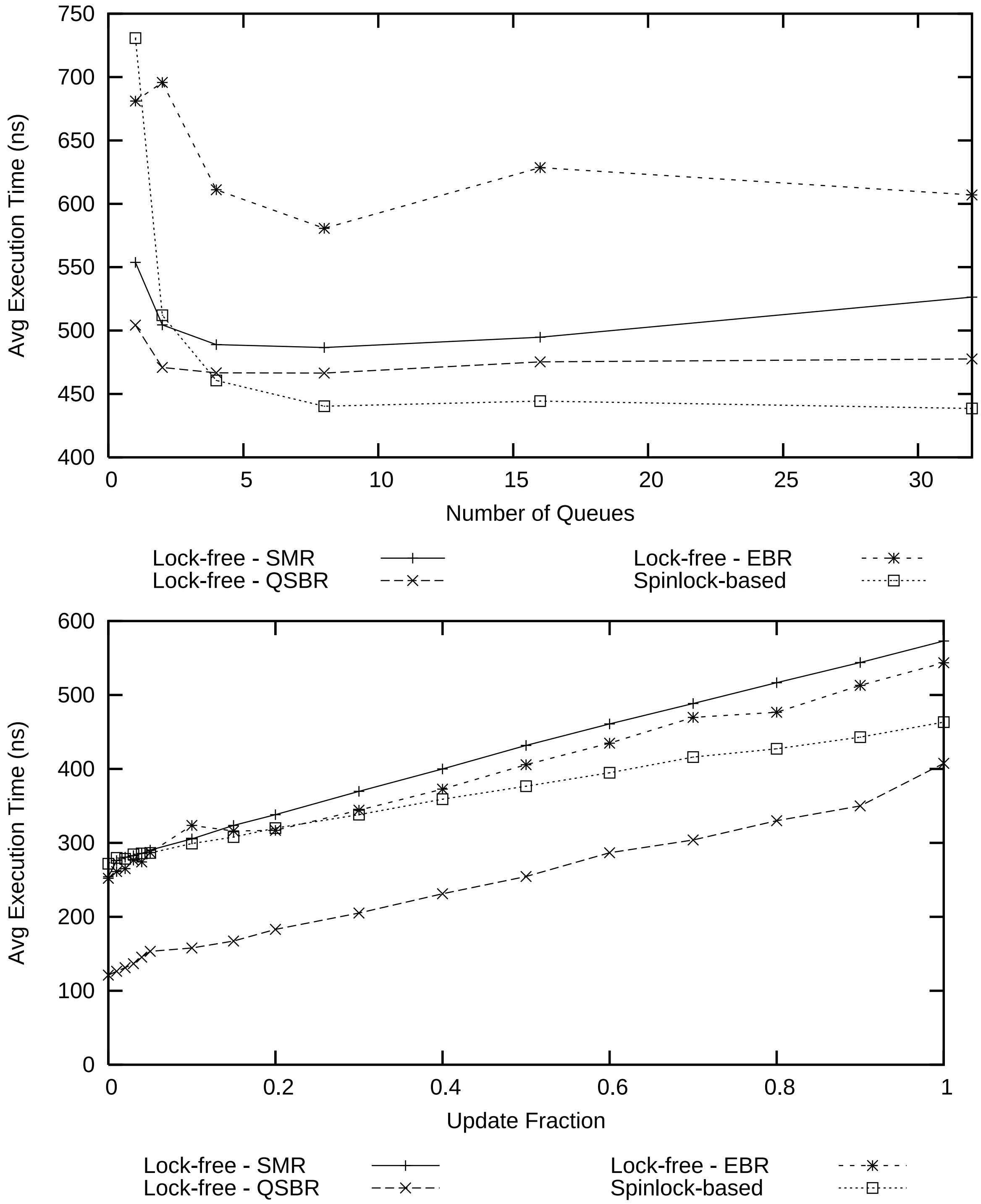

Figure 10: Left, queues, two threads, varying number of queues. Right, hash table, 32 buckets, two threads, load factor 5, varying update fraction. In the absence of CPU contention and many allocations, QSBR is the best-performing memory reclamation method.

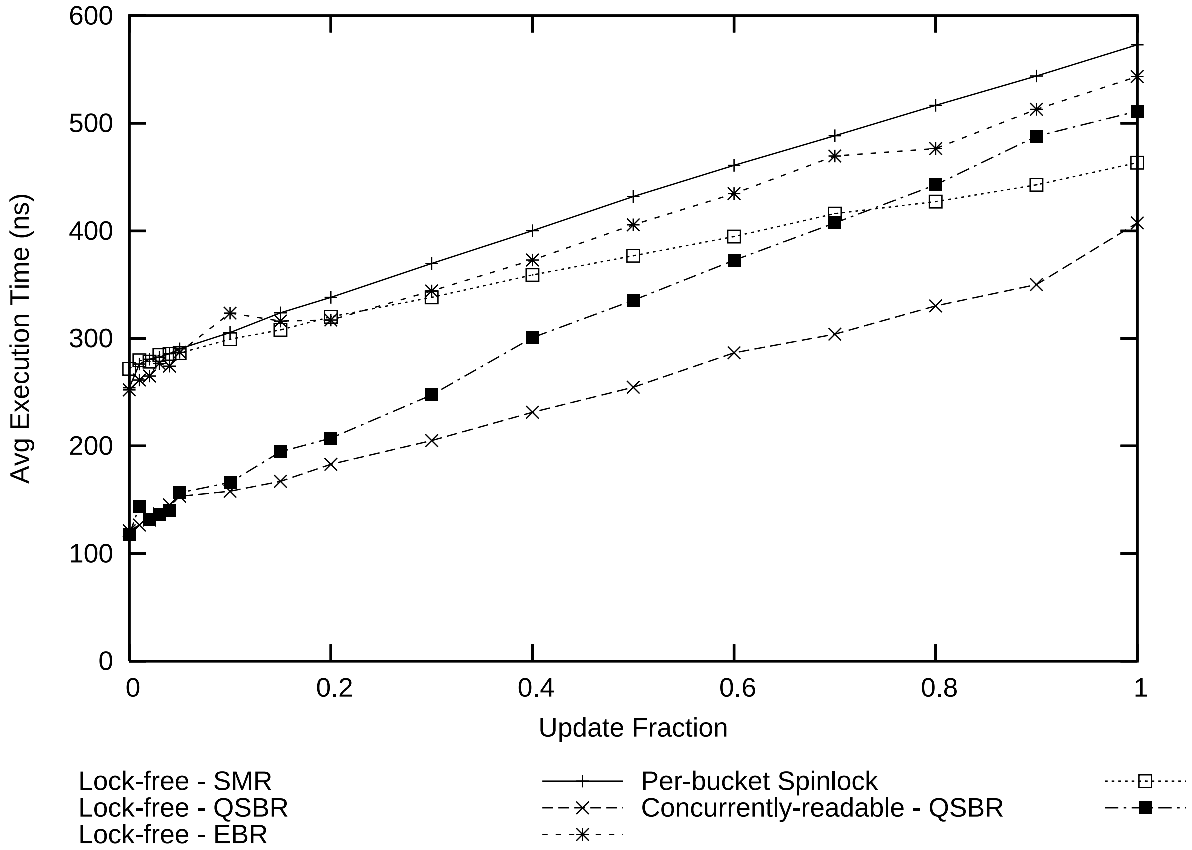

Figure 11: Hash table, 32 buckets, two threads, load factor 5, varying update fraction. QSBR allows the lock-free algorithm to out-perform RCU for almost any workload; neither SMR nor EBR achieve this.

7 Related Work

Our contribution lies not in introducing new memory reclamation strategies, but in providing a comprehensive analysis of their relative merits, and, in the case of QSBR, using it 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.

We are not the first to use QSBR with lock-free algorithms — Auslander implemented a lock-free hash table with QSBR [13] in the K42 operating system [1, 8]. 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 [13].

Michael [18] criticized QSBR for its unbounded memory use. No evaluation was given of the impact of this limitations on the performance of lock-free algorithms using QSBR. Sarma and McKenney [22], however, have shown that this leads to the possibility of denial-of-service attacks, and that preventing these attacks becomes an engineering problem.

Others have proposed that memory reclamation strategies that are not lock-free should sometimes be used in combination with lock-free algorithms. Fraser [7] 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 [10] and almost non-blocking data structures [3], designed to preserve the advantageous properties of lock-freedom while improving performance.

8 Conclusions

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 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 2.

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

Table 2: 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

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.

Finally, we note that SMR and reference counting — the two lock-free memory reclamation methods — both have overhead that grows with traversal length. In the case of SMR, this overhead comes from fence instructions, while reference counting's overhead also comes from expensive operations such as CAS. An interesting topic for future work would be to design a general-purpose lock-free memory reclamation scheme with significantly less per-element overhead which scales nicely with traversal length, or to prove that no such method can exist.

9 Acknowledgments

The authors would like to thank Maged Michael and Keir Fraser for their helpful comments on their respective work, and our colleagues at the University of Toronto for their advice on this manuscript.

References

[1] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, to appear 2005.

[2] A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proc. USENIX Annual Technical Conference (FREENIX Track)*, pages 297–310. USENIX Association, June 2003.

[3] H.-J. Boehm. An almost non-blocking stack. In *Proc. 23rd Annual ACM Symp. on Principles of Distributed Computing*, pages 40–49, 2004.

[4] J. Bonwick and J. 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.

[5] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, 2001.

[6] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

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

[8] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, pages 87–100, 1999.

[9] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proc. 16th Intl. Symp. on Distributed Computing*, October 2002.

[10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd Intl. Conference on Distributed Computing Systems*, page 522. IEEE Computer Society, 2003.

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

[12] D. 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`.

[13] P. 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.

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

[15] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001.

[16] P. E. McKenney and J. 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.

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

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

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

[20] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. 34th Intl. Symp. on Microarchitecture*, December 2001.

[21] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

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

[23] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th Annual ACM Symp. on Principles of Distributed Computing*, pages 214–222, August 1995.