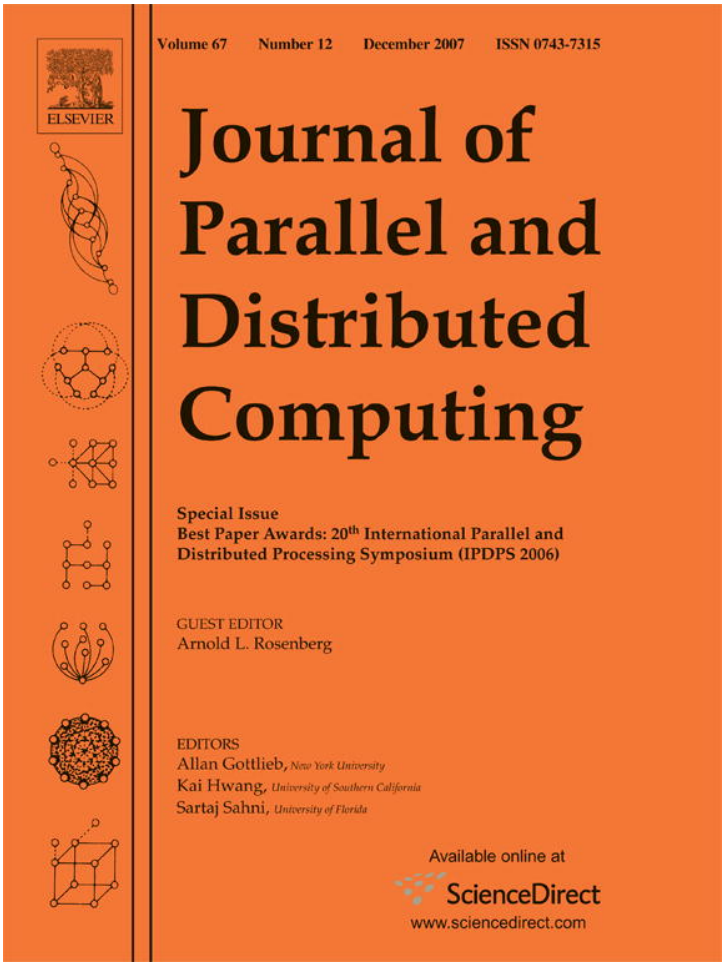

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
J. Parallel Distrib. Comput. 67 (2007) 1270–1285
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Performance of memory reclamation for lockless synchronization☆

Thomas E. Hart[a,*,1], Paul E. McKenney[b], Angela Demke Brown[a], Jonathan Walpole[c]

[a]*Department of Computer Science, University of Toronto, Toronto, Ont., Canada M5S 2E4*
[b]*IBM Linux Technology Center, IBM Beaverton, Beaverton, OR 97006, USA*
[c]*Department of Computer Science, Portland State University, Portland, OR 97207-0751, USA*

Received 1 July 2006; received in revised form 11 April 2007; accepted 25 April 2007
Available online 3 May 2007

Abstract

Achieving high performance for concurrent applications on modern multiprocessors remains challenging. Many programmers avoid locking to improve performance, while others replace locks with non-blocking synchronization to protect against deadlock, priority inversion, and convoying. In both cases, dynamic data structures that avoid locking require a *memory reclamation scheme* that reclaims elements once they are no longer in use.

The performance of existing memory reclamation schemes has not been thoroughly evaluated. We conduct the first fair and comprehensive comparison of three recent schemes—*quiescent-state-based reclamation*, *epoch-based reclamation*, and *hazard-pointer-based reclamation*—using a flexible microbenchmark. Our results show that there is no globally optimal scheme. When evaluating lockless synchronization, programmers and algorithm designers should thus carefully consider the data structure, the workload, and the execution environment, each of which can dramatically affect the memory reclamation performance.

We discuss the consequences of our results for programmers and algorithm designers. Finally, we describe the use of one scheme, quiescent-state-based reclamation, in the context of an OS kernel—an execution environment which is well suited to this scheme.

Keywords: Lockless; Non-blocking; Memory reclamation; Hazard pointers; Read-copy update; Synchronization; Concurrency; Performance

1. Introduction

As multiprocessors become mainstream, multithreaded applications will become more common, increasing the need for efficient coordination of concurrent accesses to shared data structures. Traditional locking requires expensive atomic operations, such as compare-and-swap (CAS), even when locks are uncontended. For example, acquiring and releasing an uncontended spinlock requires over 400 cycles on an IBM® POWER™ CPU. Therefore, many researchers recommend avoiding locking [3,10,28]. Some systems, such as the Linux™ kernel, use *concurrently readable* synchronization, which uses locks for updates but not for reads. Locking is also susceptible to priority inversion, convoying, deadlock, and blocking due to thread failure [5,13], leading researchers to pursue *non-blocking* (or *lock-free*) synchronization [9,15–17,19,39]. In some cases, lock-free approaches can bring performance benefits [31]. For clarity, we describe as *lockless* all synchronization strategies which permit access to shared data without using locks.

A major challenge for lockless synchronization is handling the *read/reclaim races* that arise in dynamic data structures. Fig. 1 illustrates this problem. Threads $T1$ and $T2$ both hold references to element a of a linked list, and $T1$'s removal of element a is concurrent with $T2$'s read of a's *next* field. The memory occupied by removed elements must be reclaimed to allow reuse, or memory exhaustion will eventually block all threads; however, reclaiming a is unsafe while $T2$ continues referencing it, since after reclamation, a's contents would no longer be defined, and a's *next* field might not be a valid pointer. Thread $T2$ could therefore crash or corrupt the contents of

☆ Portions of this paper appeared in the Proceedings of the 2006 International Parallel and Distributed Processing Symposium (IPDPS 2006).

* Corresponding author.
E-mail addresses: tomhart@cs.toronto.edu (T.E. Hart), paulmck@us.ibm.com (P.E. McKenney), demke@cs.toronto.edu (A.D. Brown), walpole@cs.pdx.edu (J. Walpole).

[1] Supported by an NSERC Canada Graduate Scholarship.

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.04.010

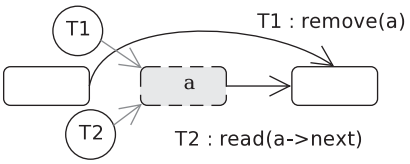

Fig. 1. Concurrent reading and writing causes read/reclaim races.

memory. The program or the system must somehow determine when *a* can safely be reclaimed.

Reclamation is subsumed into automatic garbage collectors in environments that provide them, such as Java™. Provided the garbage collector is thread-safe, programmers using garbage-collected languages therefore need not worry about read/reclaim races. However, for languages like C, where memory must be explicitly reclaimed (e.g. via `free()`), programmers must combine a *memory reclamation scheme* with their lockless data structures to resolve these read/reclaim races. Several such reclamation schemes have been proposed.

Programmers need to understand the semantics and the performance implications of each scheme, since the overhead of inefficient reclamation can be worse than that of locking. For example, *reference counting* [8,39] has high overhead in the base case and scales poorly with data-structure size. This is unacceptable when performance is the motivation for lockless synchronization. Unfortunately, there is no single optimal scheme, and existing work is relatively silent on factors affecting reclamation performance.

We address this deficit by comparing three recent reclamation schemes, showing the respective strengths and weaknesses of each. In Sections 2 and 3, we review these schemes and describe factors affecting their performance. Section 4 explains our experimental setup. Our analysis, in Section 5, reveals substantial performance differences between these schemes, the greatest source of which is per-operation atomic instructions. In Section 6, we discuss the relevance of our work to designers and implementers. We show that lockless algorithms and reclamation schemes are mostly independent, by combining a blocking reclamation scheme and a non-blocking algorithm, then comparing this combination to a fully non-blocking equivalent. We also present a new reclamation scheme that combines aspects of two other schemes to give good performance and ease of use. Section 7 describes the use of one of these memory reclamation schemes (QSBR) in the Linux kernel. We close with a discussion of related work in Section 8 and summarize our conclusions in Section 9.

2. Memory reclamation schemes

We examine four memory reclamation schemes: quiescent-state-based reclamation (QSBR) [3,28], epoch-based reclamation (EBR) [9], hazard-pointer-based reclamation (HPBR) [29,30], and lock-free reference counting (LFRC) [39,32]. In this section, we provide an overview of each scheme to help the reader understand our work.

Most of the functionality of each scheme is provided by a library into which clients call. However, each scheme places slightly different constraints on the calling code, leading to different interfaces to the respective libraries; hence, a developer cannot simply recompile her application with a new reclamation scheme, but must customize her code for each library's interface. These interfaces have different levels of complexity, leading to trade-offs between reclamation performance and coding difficulty. We elaborate on this point while discussing each scheme.

2.1. Quiescent-state-based reclamation

QSBR and EBR reclaim memory once a *grace period* has passed. A *grace period* is a time interval $[a, b]$ such that, after time b, all elements removed before time a can safely be reclaimed.

QSBR uses quiescent states to detect grace periods. A *quiescent state* for thread T is a state in which T holds no references to shared elements—in particular, T holds no references to any shared elements which have been removed from a lockless data structure. Any interval of time in which each thread passes through at least one quiescent state is thus a grace period for QSBR. Fig. 2 illustrates this relationship. Thread $T1$ goes through quiescent states at times t_1 and t_5, $T2$ at times t_2 and t_4, and $T3$ at time t_3. Hence, a grace period is any time interval containing either $[t_1, t_3]$ or $[t_3, t_5]$.

Note that there is no requirement that QSBR implementations find the shortest grace periods possible. In Fig. 2, for example, any interval containing $[t_1, t_3]$ or $[t_3, t_5]$ is a quiescent state; implementations which check for grace periods only when threads enter quiescent states would detect $[t_1, t_5]$, since $T1$'s two quiescent states form the only pair of quiescent states from a single thread which enclose a grace period.

One convenient way to implement QSBR is with a *fuzzy barrier* [14]. A barrier protects access to some code which no thread should execute before all other threads finish some prior stage of computation. In standard (non-fuzzy) barrier synchronization, threads announce their entry into a barrier, and then block until all threads have entered the barrier. In a fuzzy barrier, instead of blocking, a thread which enters the barrier skips the protected code and continues executing if some other thread has not yet entered the barrier. The thread will again attempt to execute the protected code upon subsequent fuzzy barrier entries. For implementing QSBR, a thread can enter the barrier when passing through a quiescent state; the protected code performs the memory reclamation.

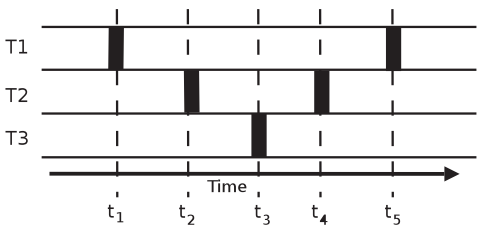

Fig. 2. Illustration of QSBR. Black boxes represent quiescent states.

Applications must somehow indicate to the QSBR library when quiescent states occur; however, the choice of quiescent states is application-dependent. In general, a thread may declare a quiescent state at any time when it has no references to any shared data. In the trivial case, a thread could declare a quiescent state after every lockless operation, as shown in Listing 1; however, it is often advantageous to declare quiescent states less frequently. In our experiments, we declare quiescent states by calling `quiescent_state()` at the end of our main test loop, as shown in Listing 4 in Section 4. A thread which calls `quiescent_state()` enters a fuzzy barrier. Calling `quiescent_state()` at the end of the loop allows us to amortize the cost of entering the fuzzy barrier across several operations.

Listing 1 : Trivial example flagging of quiescent states with QSBR.

```
1  void foo (struct list *l, long key)
2  {
3     remove (l, key);
4     quiescent_state ();
5  }
```

Many operating system kernels contain natural quiescent states. Linux uses QSBR to implement the *read-copy update* (RCU) API [10,26,28]. Several choices of quiescent state have been proposed in different QSBR implementations [24, Section 4.3] in Linux. A classical example is voluntary context switch. As another example, in parts of the Linux kernel, writers flag quiescent states immediately after writes, using the pattern shown in Listing 1, and then intentionally block until a grace period has elapsed, in order to increase maintainability and to reduce memory usage.

The standard model for reasoning about non-blocking synchronization is that any thread may experience a *fail-stop* failure at any time, and other threads cannot distinguish these failures from long stalls. In this model, QSBR is blocking, since failed threads will not go through quiescent states. Fig. 3 illustrates this problem. Thread $T2$ is a failed thread; since it has failed, it never goes through a quiescent state. By the definition of *grace period*, there are thus no grace periods in this system. Since there are no grace periods, threads $T1$ and $T3$ will never be able to reclaim memory. As a result, the system will eventually run out of memory, forcing $T1$ and $T3$ to block forever on memory allocation. In principle, systems that can detect thread failure might designate thread failure as an extended quiescent state; however, in practice, we are not aware of any such implementation, and in theory, failure detectors [7] are not a part of standard shared memory models.

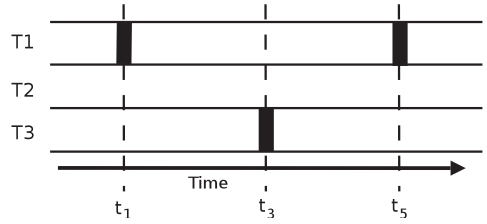

Fig. 3. QSBR is inherently blocking.

2.2. Epoch-based reclamation

Fraser's EBR [9], like QSBR, uses grace periods. EBR differs from QSBR in that QSBR relies on the programmer to annotate the program with quiescent states, but EBR hides this bookkeeping within the implementation of lockless operations. The body of a lockless operation is termed a *critical region*. Each thread sets a per-thread flag upon entry into a critical region, indicating that the thread intends to locklessly access shared data. The thread clears this flag at the end of the lockless operation. No thread is allowed to access an EBR-protected object outside of a critical region. After some pre-determined number of critical region entries, a thread attempts to enter a fuzzy barrier and reclaim memory.

Listing 2 shows an example of the use of EBR in a search of a linked list which allows lockless reads but uses locks for updates. QSBR omits lines 5, 10, and 14, which handle EBR's epoch bookkeeping, but is otherwise identical; QSBR's quiescent states are flagged explicitly by clients of the QSBR library, as shown by the pseudocode in Listings 1 and 4.

Listing 2: EBR concurrently readable search.

```
1  int search (struct list *l, long key)
2  {
3     element_t *cur;
4     int cur_key;
5     critical_enter();
6     for (cur = l->list_head;
7        cur != NULL; cur = cur->next) {
8        cur_key = cur->key;
9        if (cur->_key >= key) {
10          critical_exit();
11          return (cur_key == key);
12       }
13    }
14    critical_exit();
15    return (0);
16 }
```

EBR gets its name from the epochs it uses to implement a fuzzy barrier. Each thread executes in one of the three logical epochs and may lag at most one epoch behind the *global epoch*. Each epoch has an associated limbo list for elements awaiting reclamation. Whenever a thread enters a new epoch, it accesses the code protected by the fuzzy barrier and can safely reclaim memory. Three epochs are needed because, as Fig. 4 illustrates, a thread can execute in two epochs during a single global epoch, hence populating two limbo lists.

Fig. 4 shows how EBR tracks epochs, allowing memory to be reclaimed safely. When a thread enters a critical region, it updates its local epoch to match the global epoch. After some pre-determined number of critical region entries since changing its local epoch, a thread will attempt to increment the global epoch. This attempt will succeed only if the local epoch of each thread in a critical region is equal to the global epoch; hence, since threads update their local epochs only at the beginning of a critical region, if the global epoch is e, threads in critical

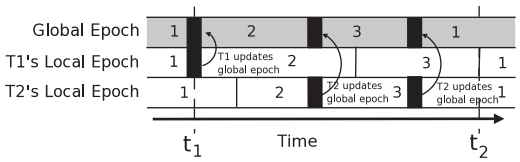

Fig. 4. Illustration of EBR. Thin solid lines show updates to a thread's local epoch, and thick solid lines show updates to both a local epoch and the global epoch.

regions can be in either epoch e or $e-1$, but not $e+1$ (all mod 3). Hence, when a thread T's local epoch changes to e, all lockless operations of other threads which were in progress the last time T was in epoch e have completed—a grace period has elapsed. The time period $[t_1, t_2]$ in Fig. 4 is thus a grace period.

As with QSBR, reclamation can be stalled by failed threads; however, unlike with QSBR, only threads that fail *within* critical regions can stall EBR. EBR's bookkeeping is invisible to the application programmer, making it simple for a programmer to use. Section 5 shows that this property imposes significant overhead on EBR.

2.3. Hazard-pointer-based reclamation

Michael's HPBR[29] scheme, sometimes called *safe memory reclamation* (SMR), provides an existence locking mechanism for dynamically allocated elements. Each thread performing lockless operations has K hazard pointers which it uses to protect elements from reclamation by other threads; hence, if there are N threads, we have $H = NK$ hazard pointers in total. K is data-structure-dependent, and often small. Queues and linked lists need $K = 2$ hazard pointers, while stacks require only $K = 1$; however, we know of no upper bound on K for general tree or graph traversal algorithms.

After removing an element, a thread places that element in a private list. When the list grows to a predefined size R, the thread reclaims each removed element lacking a corresponding hazard pointer. Increasing R amortizes reclamation overhead across more elements, but increases memory usage; if R is bigger than H by some amount proportional to H, written formally as $R = H + \Omega(H)$, the amortized per-element processing time is constant. Setting R to a positive integer multiple of H plus some constant suffices. Furthermore, since every removed element not protected by a hazard pointer can be reclaimed, at most H of the removed elements can be unreclaimable.

Since each thread has K hazard pointers and can hold R removed elements in its private list, a crashed thread can prevent only $K + R$ removed elements from being reclaimed. HPBR thus bounds the amount of memory which can be occupied by removed elements, even in the presence of thread failures. In this sense, HPBR is non-blocking—memory held by removed elements cannot grow arbitrarily and exhaust the system's memory, which would otherwise cause threads to stall. This guarantee, however, assumes a finite number of threads and hazard pointers.

An algorithm using HPBR must identify all *hazardous references*—references to shared elements that may have been removed by other threads or that are vulnerable to the ABA[2] problem [30]. Such references require hazard pointers. The algorithm sets a hazard pointer, then checks for element removal; if the element has not been removed, then its fields may safely be accessed. As long as the hazard pointer references the element, HPBR's reclamation routine refrains from reclaiming it. Fig. 5 illustrates the use of HPBR. Element a has been removed from the linked list, but cannot be reclaimed because $T2$'s hazard pointer $HP[2]$ references it.

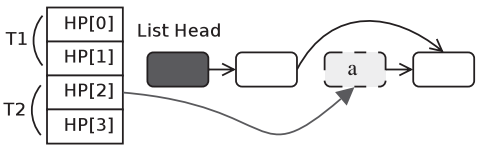

Fig. 5. Illustration of HPBR.

Listing 3, showing code adapted from Michael [30], demonstrates HPBR with a search algorithm corresponding to Listing 2. At most two elements must be protected: the current element and its predecessor ($K = 2$). The code removing elements, which are not shown here, use the low-order bit of the `next` pointer as a flag. This guarantees that the validation step on line 14 will fail and retry in case of concurrent removal. Full details are given by Michael [30].

Listing 3: HPBR concurrently readable search.

```
1   int search (struct list *l, long key)
2   {
3     element_t **prev, *cur, *next;
4     /* Index of our first hazard pointer. */
5     int base = getTID()* K;
6     /* Offset into our hazard pointer segment. */
7     int off = 0;
8   try_again:
9     prev = &l->list_head;
10    for (cur = *prev; cur != NULL; cur = next & ~ 1) {
11      /* Protect cur with a hazard pointer. */
12      HP[base+off] = cur;
13      memory_fence();
14      if (*prev != cur)
15        goto try_again;
16      next = cur->next;
17      if (cur->key >= key)
18        return (cur->key == key);
19      prev = &cur->next;
20      off = (off+1)
21    }
22   return (0);
23  }
```

[2] The ABA problem [20] occurs when we use CAS to update a data structure. Suppose that p is an element in a linked data structure. If some thread removes p and replaces it with p', which uses the *same memory* previously occupied by p, a concurrent CAS operation (for example, another thread trying to remove p) will not be able to tell the difference between p' and p. This CAS operation could then succeed when it should fail (for example, unintentionally removing p').

Provided that no lockless operation returns a reference to an element, the changes needed for a program to use HPBR are localized to the implementation of lockless operations, as Listing 3 shows. If, however, a function returns a reference to an element, a hazard pointer must protect this element during the reference's lifetime. Furthermore, applying HPBR to the implementation of a lockless operation is often more complicated than applying EBR, as can be seen by contrasting Listing 2 with 3.

Herlihy et al. [18] presented a very similar scheme called Pass the Buck. Since this scheme's per-operation costs are very similar to those of HPBR, we believe that our HPBR results apply to Pass the Buck as well.

2.4. Lock-free reference counting

LFRC is a well-known garbage-collection technique. Threads track the number of references to elements, reclaiming any element whose count is zero. Valois' LFRC scheme [39] (corrected by Michael and Scott [32]) uses CAS and fetch-and-add (FAA), and requires elements to retain their type after reclamation. Sundell's scheme [37], based on Valois', is wait-free. The scheme of Detlefs et al. [8] allows elements' types to change upon reclamation, but requires double compare-and-swap (DCAS), which no current CPU supports.

Although LFRC avoids locks, it does not bound the amount of memory consumed by removed nodes like HPBR does—Michael and Scott report easily running out of memory using Valois' version of LFRC [32]. Furthermore, Michael [30] showed that LFRC introduces overhead which often makes lockless algorithms perform worse than lock-based versions. We include some experiments with Valois' scheme to reproduce Michael's findings.

As with HPBR, only the implementations of lockless operations must be changed in order to support LFRC, provided these operations do not return references to elements. Implementing these changes is slightly more complex than for HPBR, since the programmer must ensure that each reference count which is incremented is later decremented; by contrast, if hazard pointers are reused on subsequent operations, it is not necessary to explicitly unset them.

2.5. Summary

We consider QSBR, EBR, HPBR, and LFRC. For the convenience of the reader, we list these schemes, their associated acronyms, and some basic characteristics of each in Table 1; this list also includes *new epoch-based reclamation* (NEBR), which we introduce in Section 6.2.

3. Reclamation performance factors

Several factors can affect the performance of memory reclamation schemes—memory consistency, workload, contention, thread preemption, scheduling, and memory constraints. This section explains these factors, which we vary experimentally in Section 5.

3.1. Memory consistency

Current literature on lock-free algorithms generally assumes a sequentially consistent [23] memory model, which prohibits instruction reordering and globally orders memory references. However, sequential consistency precludes many hardware and compiler performance optimizations which are possible when using a weaker memory consistency model [1]. Since most codes does not require sequential consistency, modern CPUs enforce sequential consistency only when needed by having programmers use special *fence* instructions (also called *memory barriers*). Although fences are often omitted from pseudocode, they are expensive on most modern CPUs and must be included in realistic performance analyses.

The schemes we consider require different numbers of fence instructions. HPBR, EBR, and LFRC require per-operation fences, while QSBR does not. HPBR, as shown in Listing 3, requires a fence between hazard-pointer setting and validation, thus one fence per visited element. LFRC also requires per-element fences, in addition to the atomic instructions needed to maintain reference counts. EBR requires two fences per operation: one when setting a flag when entering a critical region, and one when clearing it upon exit. QSBR has no per-operation code to manage quiescent states, so no per-operation fences are required. As we show in Section 5, this lack of per-operation fences enables QSBR to have very low per-operation overhead in many cases.

3.2. Workload, contention, and scheduling

Data structures differ in both the operations they provide and in their common workloads. Queues are update-only, but linked lists and hash tables are often read mostly [24]. Schemes which do not bound memory usage may perform poorly with update-heavy structures, since the risk of memory exhaustion is higher. Conversely, schemes which require per-element fences

Table 1
Summary of memory reclamation schemes

Acronym	Full name	Characteristics
QSBR	Quiescent-state-based reclamation	Detects grace periods using application-dependent quiescent states
EBR	Epoch-based reclamation	Detects grace periods using application-independent epochs
HPBR	Hazard-pointer-based reclamation	Uses per-thread hazard pointers for existence locking
LFRC	Lock-free reference-counting	Uses per-element reference counts for existence locking
NEBR	New epoch-based reclamation	Introduced in Section 6.2

may perform poorly with operations which must visit many elements, such as list or tree traversal.

We expect contention due to concurrent threads to be a minor source of reclamation overhead; however, for HPBR and LFRC, it could be unbounded in degenerate cases. Readers using these schemes may have to restart their traversals due to interference from concurrent writes—for example, as shown in lines 14 and 15 of Listing 3. Readers forced to repeatedly restart their traversals must *repeatedly* execute fence instructions for every element. These degenerate cases are more likely when there are many threads and the workload is update-heavy.

When the number of threads exceeds the number of processors, threads will be preempted. Preemption can adversely affect schemes which rely on grace periods—descheduled threads will not go through quiescent states or update their local epochs, and can thus delay reclamation, potentially exhausting memory. This risk of memory exhaustion is greatest with update-heavy workloads. Longer scheduling quanta may increase the risk of this exhaustion.

3.3. Memory constraints

Typically, a multi-threaded memory allocator will give each thread a local pool of memory; threads that exhaust their local pools replenish them from a global pool [4,6]. Using local pools reduces contention on the global pool. Although lock-free memory allocators exist [31], many allocators protect the global pool using locking.

If an allocator uses locking, schemes which do not bound memory usage may see greater lock contention due to having to access the lock-protected global pool more frequently. Furthermore, if a thread is preempted while holding such a lock, other threads will block on memory allocation. The size of the global pool is finite and governs the likelihood of memory exhaustion. Only HPBR [29] provides a provable bound on the amount of unreclaimed memory; it should thus be less sensitive to these constraints.

4. Experimental setup

We evaluated the memory reclamation strategies with respect to the factors outlined in Section 3 using commodity SMP systems with IBM POWER CPUs. Table 2 shows the characteristics of the two machines we used; the last line of this table gives the combined costs of locking and then unlocking a spinlock. The code for our experiments is available at `http://www.cs.toronto.edu/~tomhart/perflab/ipdps06.tgz`.

In our tests, a parent thread creates N child threads, starts a timer, and stops the threads upon timer expiry. Child threads count the number of operations they perform, and the parent then calculates the average *execution time* per operation by dividing the duration of the test by the total number of operations. The *CPU time* is the execution time multiplied by the number of threads. Provided that the threads do not outnumber the CPUs, CPU time compensates for increasing numbers of CPUs, allowing us to focus on synchronization overhead.

Table 2
Characteristics of machines

	XServe	IBM Power
CPUs	2x 2.0 GHz PowerPC G5	8x 1.45 GHz Power4+
Kernel	Linux 2.6.8-1.ydl.7g5-smp	Linux 2.6.13 (kernel.org)
Fence	78 ns (156 cycles)	76 ns (110 cycles)
CAS	52 ns (104 cycles)	59 ns (86 cycles)
Lock	231 ns (462 cycles)	243 ns (352 cycles)

In our results, we report execution time when there are more threads than CPUs, and CPU time otherwise. We average our times over five trials.

In each trial, each thread runs repeatedly through the test loop shown in Listing 4 until the timer expires. QSBR tests place a quiescent state at the end of the loop. The probabilities of inserting and removing elements are equal, keeping data-structure size roughly constant throughout a given run.

Listing 4: Per-thread test pseudocode.

```
1   while (parent's timer has not expired) {
2     for i from 1 to OPS_PER_LOOP do {
3       key = random key;
4       op = random operation;
5       d = data structure;
6       op(d, key);
7     }
8   if (using QSBR)
9     quiescent_state();
10  }
```

We tested the reclamation schemes on linked lists and queues. We used Michael's ordered lock-free linked list, which forbids duplicate keys, and Michael and Scott's lock-free queue. These data structures are known to work with HPBR and have been previously evaluated with that technique [30]; choosing these data structures makes our work more easily comparable with this prior work. We coded our concurrently readable lists similarly to the lock-free lists. Linked lists permit arbitrary lengths and read-to-update ratios, so we used them heavily in our experiments. Queues allow evaluating QSBR on an update-only data structure, which no prior studies have done.

The tests allow us to vary the number of threads and the total number of elements with which the experiment begins. When using linked lists, we are also able to specify the ratio of reads to updates.

QSBR, EBR, and HPBR all have parameters which affect the frequency of reclamation; we attempted to choose these parameters such that the schemes performed well and reclaimed memory with comparable frequency. These factors include the number of operations per quiescent state, the frequency with which threads using EBR attempt to update the global epoch, and the frequency with which threads using HPBR attempt to reclaim memory. We chose these parameters so as not to bias our experiments against any scheme. As shown in Listing 4, each thread performs `OPS_PER_LOOP` operations per quiescent state; hence, grace-period-related overhead is amortized across `OPS_PER_LOOP` operations. We set the value of `OPS_PER_LOOP` to 100 in our experiments. For EBR, each `op`

in Listing 4 is a critical region; a thread attempts to update the global epoch whenever it has entered a critical region 100 times since the last update, again amortizing grace-period-related overhead across 100 operations. For HPBR, we amortized reclamation overhead over $R=2H+100$ element removals.

Both QSBR and EBR require a fuzzy barrier algorithm; however, measuring the performance and scalability of different barrier algorithms is not our goal. We therefore chose to use EBR's fuzzy barrier algorithm for both our QSBR and EBR implementations. We experimented with other barrier algorithms and found that this one had low overhead and scaled well. In principle, we could use other fuzzy barrier algorithms for each scheme while maintaining the same programming interface.

Our memory allocator is similar to that of Bonwick [6]: each thread has two freelists of up to 100 elements each and can acquire more memory from a global non-blocking stack of freelists. This non-blocking allocator allowed us to study reclamation performance without considering pathological locking conditions discussed in Section 3.3.

The threads in our experiment were processes, created using `fork()`. We implemented CAS using POWER's `LL/SC` instructions (`larx` and `stcx`), and fences using the `eieio` instruction. Our spinlocks were implemented using CAS and fences. Our locks used exponential backoff [2], implemented using busy waiting, upon encountering conflicts, as did all our lockless algorithms.

4.1. Limitations of experiment

Microbenchmarks are never perfect [22]; however, they allow us to study reclamation performance by varying each of the factors outlined in Section 3 independently. Our results show that these factors significantly affect reclamation performance. In macrobenchmark experiments, it is more difficult to gain insight into the causes of performance differences, and to test the schemes comprehensively.

Some applications may not have natural quiescent states; furthermore, detecting quiescent states in other applications may be more expensive than it is in our experiments. Our QSBR implementation, for example, is faster than that used in the Linux kernel, due to the latter's need to support dynamic insertion and removal of CPUs, interrupt handlers, and real-time workloads.

Our HPBR experiments statically allocate hazard pointers. Although this is sufficient for our experiments, some algorithms, to the best of our knowledge, require unbounded numbers of hazard pointers.

We believe that, despite the above limitations, our experiments thoroughly evaluate these memory reclamation schemes and show when each scheme is and is not efficient. In Section 7, we describe experiments with one scheme, QSBR, in the context of the Linux kernel and show that these macrobenchmark results are consistent with the predictions of our microbenchmark.

5. Performance analysis

We first investigate the base costs for the reclamation schemes: single-threaded execution on small data structures. We then show how workload, list traversal length, number of threads, and preemption affect the performance.

5.1. Base costs

We first measure the costs of these schemes without the costs associated with contention, preemption, or traversing long lists; this represents the best-case performance of these schemes. Fig.6 shows the single-threaded base costs of these schemes on non-blocking queues and single-element linked lists with no preemption or contention. We note that in a well-designed system, contention should usually be low—good best-case performance is thus highly desirable.

For the purposes of comparison, we show how these lockless algorithms, with the support of the different reclamation schemes, compare with simple spinlock-based alternatives in the base case. These data are presented only to show that the lockless schemes are competitive: our goal is to compare different memory reclamation schemes, not to compare lock-based algorithms to lockless ones, or to compare different types of locks. We therefore limit our experiments to lockless synchronization for the evaluation in this section.

We ran LFRC only on read-only workloads; these were sufficient for us to corroborate Michael's [30] result that LFRC performs poorly.

In these base cases, the dominant influence on the performance is per-operation atomic instructions: CAS, FAA, and fences make LFRC much more expensive than the other schemes. Since EBR requires two fences per operation (when calling `critical_enter()` and `critical_exit()`, respectively), and HPBR requires one for most operations considered here, EBR is usually the next most expensive. QSBR, needing no per-operation atomic instructions, is the cheapest scheme in the base case.

Workload affects the performance of these schemes. Under an update-intensive workload, a significant number of operations will involve removing elements; for each attempt to reclaim a removed element, HPBR must search the array of hazard pointers. This overhead can become significant for update-intensive workloads, as can be seen in Fig. 6: HPBR performs best for the read-only linked lists and worst for the

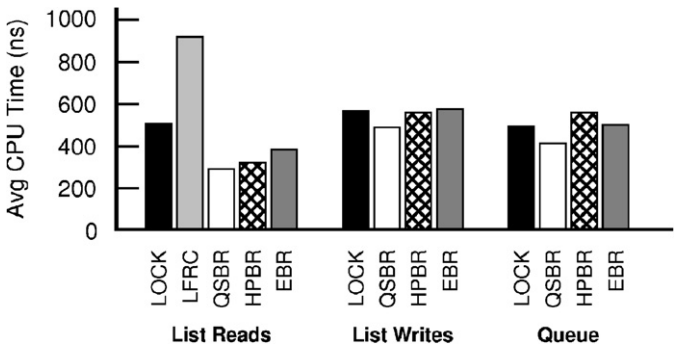

Fig. 6. Base costs—single-threaded data from 8-CPU machine. *Y*-axis shows *CPU time*, defined as the average per-operation execution time multiplied by the number of threads.

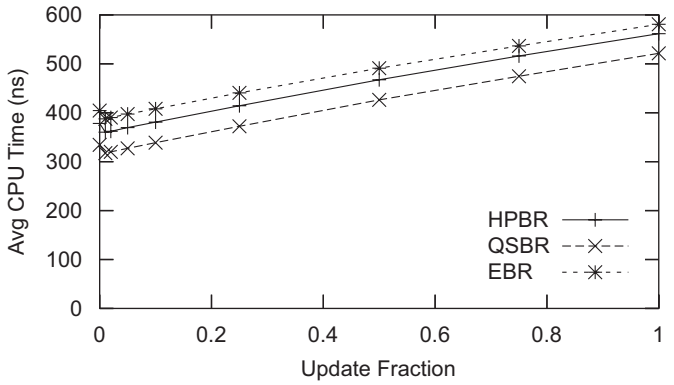

Fig. 7. Lock-free list, one thread, one element, 8-CPUs, varying workload.

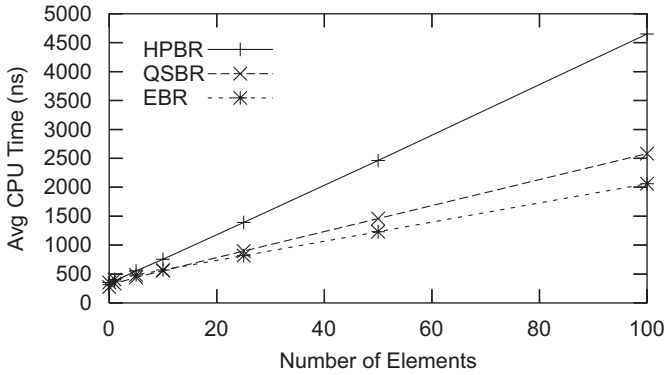

Fig. 8. Effect of traversal length—read-only lock-free list, one thread, 8 CPUs.

update-only queues. HPBR's performance for the update-only linked lists is intermediate—list removals fail when a matching key is not found in the list, and therefore do not result in an element needing reclamation, and therefore do not result in an element needing reclamation. Reclamation overhead is thus amortized across more operations compared to the update-only queue case, in which dequeue operations fail only when the queue is empty.

5.2. Scalability with workload

Fig. 6 shows us how the reclamation schemes perform with read-only and update-only workloads. Using linked lists allows us to see how the schemes perform with intermediate read-to-update ratios. Fig. 7 shows the schemes' performance as we gradually increase the read-to-update ratio from read-only to update-only on a single-threaded workload. QSBR and EBR exhibit almost the same slope. HPBR, however, experiences a slight increase in overhead as the update fraction increases. As noted above, this is due to the fact that HPBR has to perform additional processing each time it reclaims a removed element. Since this increase is small, we conclude that workload, in isolation, is a minor factor in reclamation performance.

5.3. Scalability with traversal length

Fig. 8 shows the effect of list length on a single-threaded read-only workload. We observed similar results in update-only workloads. As expected, per-element fence instructions degrade HPBR's performance on long chains of elements; QSBR and EBR do much better.

Fig. 9 shows the same scenario, but also includes LFRC. At best, LFRC takes more than twice as long as the next slowest scheme, and the performance gap rapidly increases with the list length due to the multiple per-element atomic instructions. Because LFRC is always the worst scheme in terms of performance, we do not consider it further.

5.4. Scalability with threads

Concurrent performance is an obvious concern for memory reclamation schemes. We study the effect of threads sharing

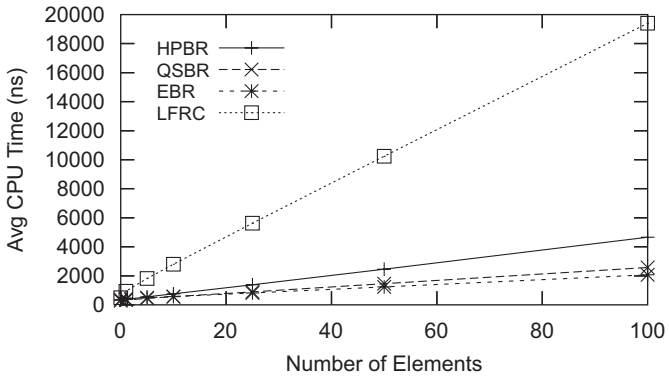

Fig. 9. Effect of traversal length, including LFRC—read-only lock-free list, one thread, 8 CPUs.

the data structure when there is no CPU contention, and when threads must also compete for the CPU.

5.4.1. No preemption

When we attempted to run eight non-real-time processes simultaneously on our 8-CPU machine, we experienced the effects of preemption, which we examine in the next subsection. To reduce the risk of preemption and the other effects of CPU contention, such as thread migration, we use a maximum of seven threads, ensuring that one CPU is available for other processes, following Fraser [9]. We could have instead prevented preemption by running our threads with real-time priority. We ran limited tests to confirm that doing so prevents preemption and its associated effects on performance; however, since the scheduler could have other effects on performance, we chose to simply keep a CPU free instead.

Figs. 10 and 11 show the performance of the reclamation schemes with a read-only workload on a linked list, and with an update-only workload on a queue, respectively. All three schemes scale almost linearly in the read-only case. In the update-only case shown in Fig. 11, we see increased overhead in all cases because multiple threads are trying to update a single queue; since hazard pointers have to be re-set when a thread restarts an operation, HPBR becomes slightly worse when contention is high. Aside from this minor increase, the schemes'

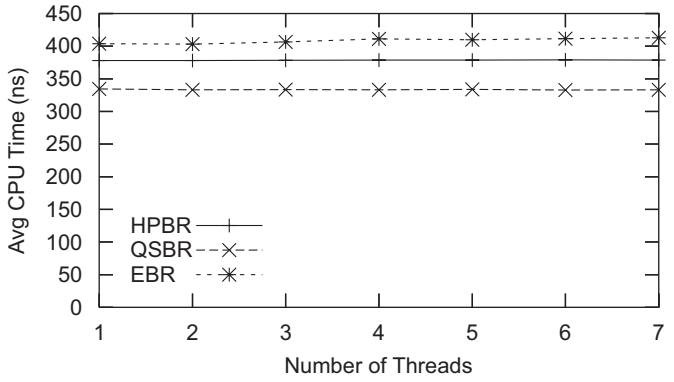

Fig. 10. Effect of adding threads—read-only lock-free list, one element, 8 CPUs.

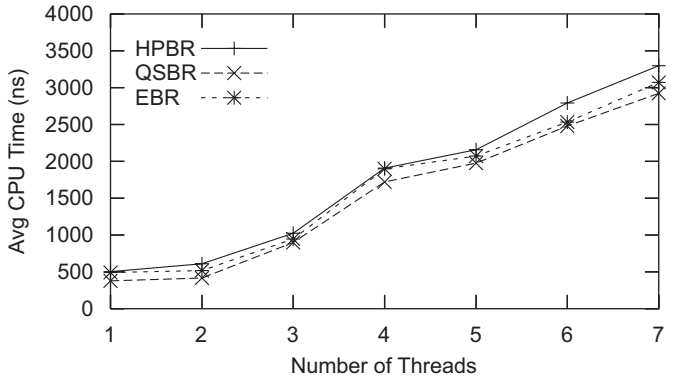

Fig. 11. Effect of adding threads—lock-free queue, 8 CPUs.

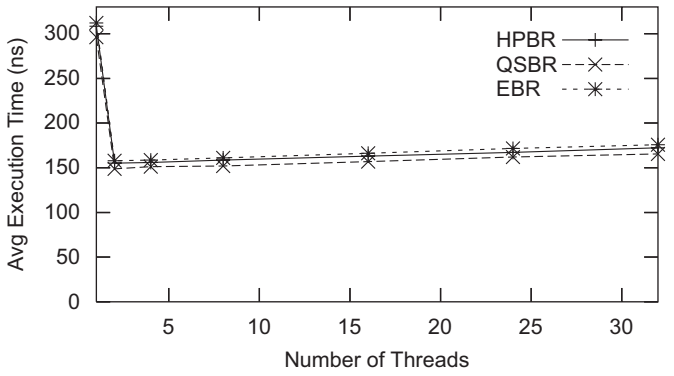

Fig. 12. Effect of preemption—read-only lock-free list, 2 CPUs.

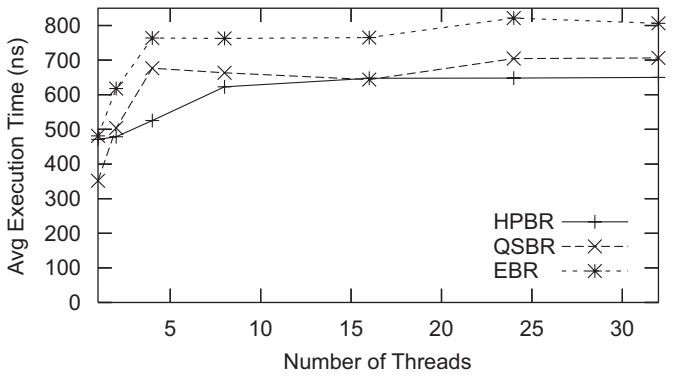

Fig. 13. Effect of preemption—lock-free queue, 2 CPUs.

relative performance is largely unaffected by the number of threads, regardless of whether the workload is read-only or update-only.

5.4.2. With preemption

To evaluate the performance of the reclamation schemes under preemption, we ran our tests on our 2-CPU machine, varying the number of threads from 1 to 32.

Fig. 12 shows the performance of the schemes on a one-element lock-free linked list with a read-only workload. This case eliminates reclamation overhead, focusing solely on read-side and fuzzy barrier overhead. In this case, the algorithms all scale well, with QSBR remaining the most efficient.

For the update-heavy workloads, such as the update-only queue shown in Fig. 13, HPBR performs best. Preempted threads slow down threads using QSBR or EBR, since their failure to go through quiescent states in a timely manner will result in memory exhaustion, leading to allocation failures. We note that HPBR performs best even when the number of threads is equal to the number of CPUs—this is because, when the number of threads and CPUs are equal, other system threads may preempt our threads, as discussed in the previous subsection.

In the above experiments, threads using QSBR or EBR yield the processor on allocation failure using `sched_yield()`, since preempted threads must run in order for grace periods to occur and thus for memory to be reclaimed. Yielding the processor upon allocation failure is a necessary condition for QSBR and EBR to have acceptable performance under an update-heavy workload with preemption. Fig. 14 shows the same test as Fig. 13, but with busy-waiting upon allocation failure. Here, HPBR performs well, but EBR and QSBR quickly exhaust the pool of free memory. Each thread spins waiting for more memory to become free, thereby further preventing grace periods from completing in a timely manner and hence delaying memory reclamation.

Although busy waiting on allocation failure would be a poor design choice in an application using grace periods for memory reclamation, this test demonstrates that preemption and update-heavy workloads can cause QSBR and EBR to exhaust all memory. In situations in which grace periods are not achieved in a timely manner, HPBR's bounds on unfreed memory become valuable.

5.5. Summary

The performance of the different memory reclamation schemes is often comparable, as in Figs. 11 and 12; however, in degenerate cases, reclamation overhead can dominate execution time, as in Figs. 9 and 14. Programmers must

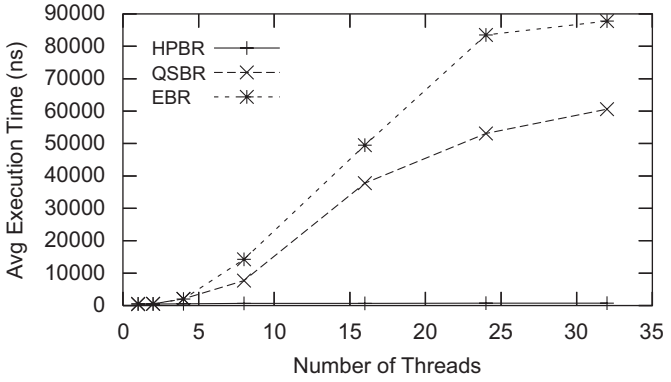

Fig. 14. Effect of busy waiting when out of memory—lock-free queue, 2 CPUs.

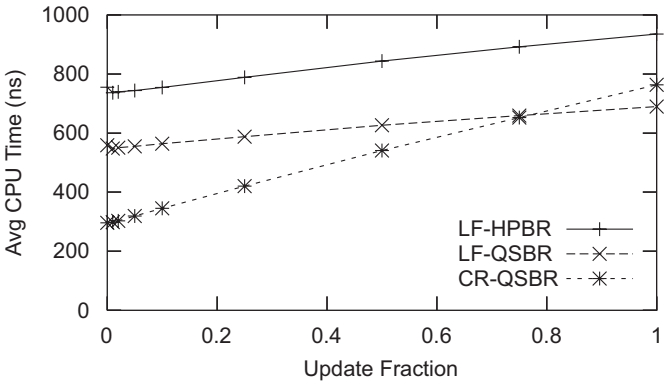

Fig. 15. Lock-free (LF) versus concurrently readable (CR) algorithms—10-element lists, one thread, 8 CPUs.

therefore understand the tradeoffs between the schemes in order to ensure good performance.

In the base case, atomic instructions such as fences are the dominant cost. The overhead due to factors such as entering and exiting a fuzzy barrier or scanning an array of hazard pointers is minor, and only affects the performance noticeably in the case of the queue results in Fig. 6.

HPBR and LFRC require per-element atomic instructions. Since these atomic instructions are expensive, HPBR and LFRC perform poorly when long chains of elements must be traversed. In the case of LFRC, the need for atomic increments is inherent, and in the case of HPBR, the need for fences is a consequence of the weakly consistent memory models on modern processors.

QSBR and EBR depend on grace periods occurring sufficiently often. If grace periods are stalled—for example, due to preemption—and the workload is update-heavy, these schemes may exhaust memory. In our experience, the impact of this problem can be reduced by yielding the processor on allocation failure; experience with the Linux kernel also suggests that this problem can be mitigated in practice [34].

6. Consequences and discussion

We describe the consequences of our analysis for comparing algorithms, designing new reclamation schemes, and choosing reclamation schemes for applications. We also discuss factors other than performance which affect the choice of memory reclamation scheme. Finally, we present system-level performance results obtained from the Linux kernel.

6.1. Fair evaluation of algorithms

Reclamation schemes have profound performance effects that must be accounted for when experimentally evaluating new lockless algorithms.

Fig. 15 shows one of our early, faulty experiments, performed prior to the evaluation in Section 5; it plots CPU time against update fraction for a 10-element list. The goal of this experiment was to compare the performance of a lock-free linked list with HPBR (LF-HPBR) with a concurrently readable equivalent using QSBR (CR-QSBR). Concurrently readable algorithms have lockless reads but use locks for updates; such algorithms are used with QSBR in the Linux kernel. The motivation for these pairings was to compare a combination with strong fault-tolerance guarantees—a thread cannot hold a lock indefinitely or stop other threads from reclaiming memory—with a combination lacking these guarantees. Our intuition was that lock-free algorithms might pay a performance penalty for these fault-tolerance properties.

We initially performed this experiment with only the LF-HPBR and CR-QSBR traces shown in Fig. 15. Since these traces never cross, our original experiment led us to the erroneous conclusion that the concurrently readable algorithm is always faster. A better analysis also performs the experiment with LF-QSBR, noting that as the update fraction increases, lock-free performance improves relative to the concurrently readable approach, since its updates require fewer atomic instructions than does locking. For update fractions above roughly 75% LF-QSBR achieves the best performance. The large gap between the LF-HPBR and CR-QSBR traces is not due to the difference between lock-free and concurrently readable linked lists, but the fact that HPBR requires per-element fences, and this list has 10 elements. This example shows that one can accurately compare two lockless algorithms *only* when each is using the same reclamation scheme.

The lock-free linked list performs some per-element checks which the concurrently readable list does not, independent of which reclamation scheme is used. Since the list in Fig. 15 has 10 elements, these checks impose significant overhead. The lock-free linked list is thus more appealing when fewer elements must be traversed. Fig. 16 shows the performance of hash tables being concurrently accessed by four threads. The hash table consists of arrays of LF-QSBR or CR-QSBR single-element lists, using the same list algorithms as in Fig. 15. For clarity, we omit HPBR from this graph—our intent is to compare the lock-free and concurrently readable algorithms using a common reclamation scheme. Here, since there are fewer elements on which to perform checks, the lock-free algorithm that out-performs the concurrently readable alternative for update fractions above about 20%. Lock-free lists and

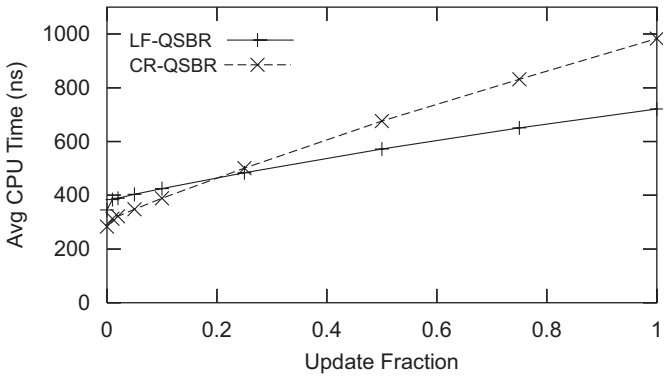

Fig. 16. Lock-free (LF) versus concurrently readable (CR) algorithms—hash tables with load factor 1, four threads, 8 CPUs.

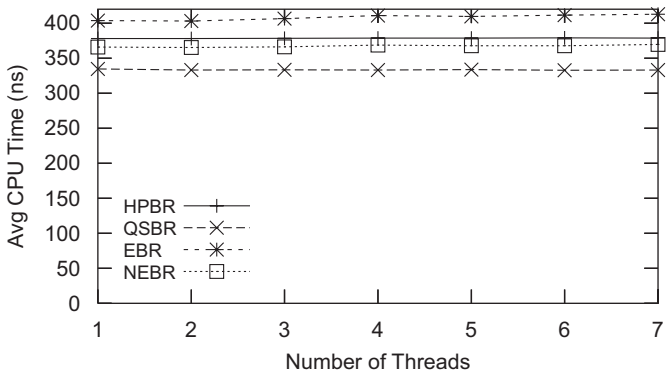

Fig. 17. Performance of NEBR—lock-free list, 8 CPUs, read-only workload, variable number of threads.

hash tables might therefore be practical for update-heavy situations in environments providing QSBR, such as OS kernels like Linux.

New reclamation schemes should also be evaluated by varying each of the factors that can affect their performance. For example, Gidenstam et al. [11] proposed a new non-blocking reclamation scheme that combines reference counting with HPBR, and can be proven to have several attractive properties. However, like HPBR and reference counting, it requires expensive per-element atomic operations. The evaluation of this scheme consisted only of experiments on double-ended queues, thus failing to evaluate scalability with data-structure size, a weakness of HPBR. This failing shows the value of our analysis: it is necessary to vary the experimental parameters we considered to gain a full understanding of a given scheme's performance.

6.2. Improving reclamation performance

Improved reclamation schemes can be designed based on an understanding of the factors that affect the performance. For example, we observe that a key difference between QSBR and EBR is the per-operation overhead of EBR's two fences which it requires when entering and leaving a critical region. This observation allows us to make a modest improvement to EBR called NEBR.

NEBR requires compromising EBR's application-independence. Instead of setting and clearing a flag at the start and end of every lockless operation, we set it at the application level before entering any code that might contain NEBR critical regions. Since our flag is set and cleared at the application level, we can amortize the overhead of the corresponding fence instructions over a larger number of operations. We reran the experiment shown in Fig. 10, but including NEBR, and, as shown in Fig. 17, found that NEBR scaled linearly and performed slightly better than did HPBR.

Furthermore, NEBR does not need the expensive per-element atomic operations that ruin HPBR's performance for long traversals. NEBR's overhead relative to QSBR can be attributed to its need to attempt to check and update epochs when starting a lockless operation.

NEBR is attractive because its overhead is close to that of QSBR, but it simplifies the job of an application programmer. With QSBR, a programmer must identify an appropriate set of quiescent states in the application code, and mark them. With NEBR, the programmer's job is simplified to marking only the beginning and end of regions of the application code in which lockless calls are made; updating the global epoch is still periodically attempted after some fixed number of lockless operations. Interestingly, recent real-time variants of the Linux-kernel RCU also dispense with quiescent states [27]. Ongoing work is expected to substantially reduce real-time RCU read-side overhead.

There are many opportunities for future work in the area of memory reclamation. Our recommendation that threads using QSBR or EBR yield the CPU on allocation failure is similar in spirit to the idea of scheduler-conscious synchronization [21]. It may be possible to further apply the ideas of scheduler-conscious synchronization to create a user-level QSBR implementation with less overhead or a lower memory footprint than the one used in our experiments.

HPBR requires per-element fences on any machine with a weakly consistent memory model. A scheme which avoids per-element atomic instructions yet still makes some guarantees about memory usage would be a desirable technique. Creating such a scheme, and formalizing what progress guarantees it makes, is an interesting challenge.

We note that it is possible to combine aspects of different reclamation schemes. Seigh [35] has proposed a scheme which combines HPBR with QSBR in an attempt to get good flexibility and performance.

6.3. Blocking memory reclamation for non-blocking data structures

Schemes relying on grace periods are blocking in the sense that if grace periods do not happen, the system will run out of memory, and threads will fail to make progress. We have shown that non-blocking data structures often perform better when using these blocking schemes than they do with HPBR, which is non-blocking. One might question why one would want to use a non-blocking data structure in this case, since a halted thread would cause an infinite memory leak, thus

destroying the non-blocking data structure's fault-tolerance guarantees.

However, non-blocking data structures are often used for reasons other than fault-tolerance; for example, Qprof [5] and Cache Kernel [13] both use such structures because they can be accessed from signal handlers without risk of self-deadlock. Blocking memory reclamation does not remove this benefit. In fact, Cache Kernel uses a blocking implementation of type-stable memory to guard against read-reclamation races; its implementation [12] has similarities to QSBR. Non-blocking algorithms with blocking reclamation schemes similarly continue to benefit from resistance to preemption-induced convoying and priority inversion.

We view combining a non-blocking algorithm with a blocking reclamation scheme as part of a trend toward weakened non-blocking properties [5,19], designed to preserve selected advantages of non-blocking synchronization while improving performance. In this case, threads have all the advantages of non-blocking synchronization, *unless* the system runs out of memory.

Conversely, one may also use non-blocking reclamation with blocking algorithms to reduce the amount of memory awaiting reclamation in the face of preempted or failed threads.

6.4. Non-performance factors

Although we have analyzed memory reclamation schemes purely from a performance standpoint, performance is not the only factor to consider when choosing a memory reclamation scheme. The schemes have slightly different semantics which may make one scheme or another more attractive for a particular application.

Reference counting, for its part, has well-known difficulties in dealing with cyclic garbage [11]. However, it has the advantage that elements can be accessed after having been removed from a shared data structure; this feature was exploited by Sundell and Tsigas [36,38] to let threads accessing a doubly linked list avoid restarting their traversals when elements are removed.

Identifying appropriate quiescent states in an application may not be straightforward [30]; furthermore, choosing inappropriate quiescent states may have unforeseen consequences. Sarma and McKenney [34] discussed how the use of QSBR in the Linux kernel's IPv4 route cache introduced the possibility of denial-of-service (DoS) attacks. By sending a large number of packets to a machine, an attacker could force the victim to spend all its time in interrupt handlers, which had no quiescent states in older versions of Linux. The attacker could thus indefinitely extend grace periods, resulting in unbounded memory consumption that would eventually cause the system to hang. Enhancements to Linux's QSBR infrastructure now provide a new set of quiescent states for use in interrupt handlers that can prevent such attacks from indefinitely extending grace periods. It is an open question, however, whether Linux's QSBR infrastructure can be formally proven to be robust in the face of arbitrary DoS attacks.

As noted in Section 2.3, some algorithms may require an unbounded number of hazard pointers; this requirement stymied an attempt to use hazard pointers in Linux to improve real-time response [27]. The potentially unbounded number of hazard pointers necessitates dynamic hazard pointer allocation [30], requiring that the HPBR reference-acquisition primitive be able to either block or return failure in low-memory situations. In certain OS kernel situations neither blocking nor failing is an acceptable response. For example, when physical memory is exhausted the OS will write pages to external swap space in order to free memory. Use of a deferred-destruction scheme based on HPBR in this case can lead to memory deadlock because memory is exhausted and hazard pointer allocation will likely fail. In general, allowing hazard pointer allocation to block would prohibit the use of this scheme in OS kernel situations where blocking is not acceptable (such as when interrupts are disabled). The alternative solution of introducing a failure return introduces new software engineering difficulties. Although these problems can be dealt with in principle, they may be prohibitively difficult to handle in practice in a large and complex code base like an OS kernel.

In addition, the not-uncommon case of aggregated data structures (for example, nested C-language structs) can pose software-engineering obstacles for the use of hazard pointers. In such cases, constraints must be imposed on memory allocation, such that a pointer to any component of the aggregate will be deemed equivalent to an alias, that is to say, a pointer to some other component of that same aggregate.

Finally, achieving the potential memory-footprint advantages of hazard pointers depends on the use of a fixed, finite number of hazard pointers per thread. In this case, hazard pointers do not need to be explicitly released, since they will be reused in a relatively short time period. If hazard pointers are dynamically allocated, however, then they must also be explicitly released to provide bounds on the amount of unreclaimed memory.

It might still be advantageous to use HPBR in operating system kernels like Linux, but only if its use is restricted to code that (1) is not used when freeing up memory, thus avoiding out-of-memory deadlocks, (2) requires a strictly bounded number of hazard pointers, and (3) avoids the pointer-aliasing problems posed by aggregated data structures. These limitations, however, make HPBR unsuitable for use as a memory reclamation scheme when implementing the current Linux RCU API. In the following section, we take a closer look at this API and several of its uses in the Linux kernel.

7. Example uses: the Linux RCU API

The Linux RCU API provides a set of operations that allow lockless concurrent reads of shared data structures and deferred destruction of elements removed from these structures. Writers may not prevent readers from accessing the shared data, but must coordinate with each other in some way. Typically, traditional spinlocks are used to prevent concurrent updates, however, other methods could be used as well; the RCU API does not dictate how updates should be coordinated. In this section, we review the Linux RCU API and its use of memory, and then consider a specific example where RCU is used in the Linux kernel.

7.1. Requirements and memory reclamation

The RCU API was designed for use in OS kernels; it is defined to neither block nor fail for readers, and cannot feasibly be altered to do so. As noted in Section 6.4, allowing blocking would prohibit the use of RCU in many situations where it has proven valuable (such as when interrupts are disabled). Introducing a failure return instead would be extremely difficult in many of the 244 uses of this primitive in the Linux 2.6.20 kernel. As a result, RCU uses QSBR for memory reclamation.

The RCU system tracks quiescent states and grace periods. In the original version, designed for non-preemptable kernels, context switches are used to identify quiescent states. To allow preemptable kernels the API requires read-side critical regions to be identified with calls to `rcu_read_lock` and `rcu_read_unlock`. These calls disable and enable preemption to guarantee that context switches (and thus quiescent states) do not occur while a thread is in the middle of accessing an RCU-protected data structure. The RCU API has been further extended with a new set of quiescent states for use in interrupt handlers, motivated by the need to prevent DoS attacks, as mentioned in Section 6.4. Finally, other extensions have targeted real-time response.

The requirements on the RCU API in an OS kernel make QSBR a natural choice for memory reclamation. Preemption can be suppressed in a kernel environment, so delayed grace periods are unlikely. In addition, uses of RCU in Linux have targeted read-mostly data structures that are rarely updated in common uses. Since updates are rare, memory reclamation is also rare and the most important performance consideration is the overhead required for reads. Our microbenchmark results show that QSBR has the lowest per-operation overhead for read-only workloads (see Fig. 6), and we would therefore expect it to have the best performance for these real uses as well. Furthermore, since QSBR does not have expensive per-element atomic instructions, it is well fitted to the concurrently readable lists which are common in Linux.

Although EBR could also be used in Linux, it has no performance advantage over QSBR; any usability advantages are less relevant given that an implementation with QSBR already exists. Attempts to use HPBR require dynamic hazard pointer allocation, leading to the difficulties of failing or blocking noted earlier. In addition, the RCU API releases references implicitly at the end of the corresponding RCU read-side critical region, rather than at the point where the reference is no longer needed. Therefore, an HPBR-based RCU implementation that traverses an arbitrary-length list in a single RCU read-side critical section could consume an arbitrarily large number of hazard pointers. On the other hand, modifying Linux's RCU API to include explicit release of RCU references would require prohibitively large and pervasive changes to Linux [27].

7.2. System V IPC

We now focus on one specific use of RCU in the Linux kernel—namely, System V IPC. Since alternate implementations using different memory reclamation schemes do not exist, we cannot compare their performance in these cases. Our performance comparisons are thus between traditional locking and RCU. These results show that lockless approaches are valuable and can out-perform lock-based approaches by a considerable margin, and that blocking reclamation schemes can be practical in large applications.

It would be desirable to make all such comparisons using a contemporary version of the Linux kernel; however, where RCU has proved valuable it is difficult and largely pointless to re-engineer the kernel to re-introduce locking again. An alternative approach would map the RCU uses in the Linux kernel to conventional locking; however, such a mapping is in general problematic. Some of the difficulties include:

(1) RCU read-side critical sections may be entered unconditionally in any software environment within the kernel, including even the non-maskable interrupt handlers that result in deadlocks when locking is used.
(2) RCU read-side critical sections may include update code. Attempts to map this usage into reader–writer locking again result in deadlocks.

Such difficulties are in fact common in the Linux kernel, as was learned in a failed attempt to replace the RCU API with locking in order to improve real-time response [25].

Given the impracticality of mapping RCU onto conventional locking, we instead present the performance comparisons that were made on server-class machines when proposing concurrently readable algorithms with RCU for acceptance into the Linux kernel. We refer to this use of RCU, where updates use locking, as CR-QSBR. Specifically, we analyze the use of CR-QSBR in the implementation of the System V IPC subsystem in the Linux kernel, showing the system- and application-level performance implications. Further system-level examples are discussed in detail by McKenney [24].

The System V IPC subsystem implements System V semaphores, message queues, and shared memory. Applications access these resources using an integer ID, and the Linux kernel uses an array to map from this ID to in-kernel data structures that represent the corresponding resource. The array is expandable, and prior to the conversion to use CR-QSBR, was protected by a spinlock. The array is frequently accessed read-only when System V IPC objects are used and infrequently accessed for writing when objects are created or deleted or the array is resized. Because each element of the array is a single aligned pointer, object creation and deletion events may be done in place, hence the array need only be copied for expansions. After conversion to use CR-QSBR, all writes continue to use the original spinlock, while the more common read operations simply flag entry and exit from critical regions using the `rcu_read_lock` and `rcu_read_unlock` functions.

Two experiments were used to compare the performance of the Linux 2.5.42-mm2 kernel, with and without CR-QSBR. The first experiment used a System V semaphore user-level benchmark on an 8-CPU 700 MHz Intel PIII system. In this benchmark, multiple user-level processes each repeatedly acquire and release different semaphores in parallel, with the benchmark

Table 3
Semopbench application-level results (seconds)

Kernel	Run 1	Run 2	Avg.
2.5.42-mm2	515.1	515.4	515.3
2.5.42-mm2+ipc-rcu	46.7	46.7	46.7

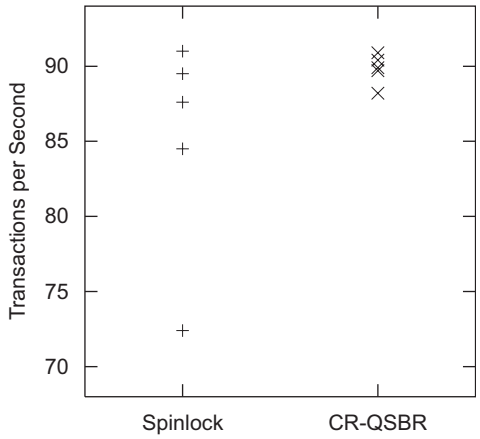

Fig. 18. DBT1 database benchmark raw results.

Table 4
DBT1 database benchmark results (TPS)

Kernel	Average	Standard deviation
2.5.42-mm2	85.0	7.5
2.5.42-mm2+ipc-rcu	89.8	1.0

metric being the length of time for each process to complete a fixed number of such operations. The second experiment used the DBT1 [33] database-webserver benchmark on an Intel dual-CPU 900 MHz PIII with 256 MB of memory. The results of the first experiment are shown in Table 3 and illustrate an order-of-magnitude increase in the performance for this user-level benchmark. The raw results for the second experiment are presented in Fig. 18, with a summary presented in Table 4. The results show that the RCU-based kernel performs over 5% better in terms of transactions per second (TPS) than does the stock kernel. More importantly, the results for the RCU-based kernel are much more stable; the erratic results for the stock kernel are not unusual for workloads with lock contention.

8. Related work

Relevant work on reclamation scheme design was discussed in Section 2. Previous work on the performance of these schemes, however, is limited. Michael [30] criticized QSBR for its unbounded memory use, but did not compare the performance of QSBR to that of HPBR, or determine when this limitation can affect a program. Fraser [9] noted, but did not thoroughly evaluate, HPBR's fence overhead and used his EBR instead. Our work extends Fraser's, showing that EBR itself has high overhead, often exceeding that of HPBR.

Auslander implemented a lock-free hash table with QSBR in K42 [24]. No performance evaluation, either between different reclamation methods or between concurrently readable and lock-free hash tables, was provided. We are unaware of any work combining QSBR with update-intensive non-blocking algorithms such as queues.

McKenney [24] details many uses of QSBR in the Linux and K42 OS kernels. We discuss one use of QSBR—namely, System V IPC—but the focus of our work is on the comparative performance of memory reclamation schemes.

9. Conclusions

We have performed the first fair comparison of blocking and non-blocking reclamation across a range of workloads, showing that reclamation has a huge effect on lockless algorithm performance. Choosing the right scheme for the environment in which a concurrent algorithm is expected to run is essential to having the algorithm perform well.

Our results, starting with Fig. 6, show that quiescent-state-based reclamation (QSBR) is usually the best-performing reclamation scheme; however, the performance of both QSBR and epoch-based reclamation (EBR) can suffer due to memory exhaustion in the face of thread preemption or failure. Hazard-pointer-based reclamation (HPBR) and EBR have higher base costs than QSBR due to their required fences; for EBR, the worst-case overhead of fences is constant, while for HPBR it is unbounded. Lock-free reference counting (LFRC) has even higher overhead due to the per-element atomic instructions it requires. HPBR and LFRC both scale poorly when many elements must be traversed.

Our analysis helped us to identify the main source of overhead in EBR and decrease it, resulting in our new epoch-based reclamation (NEBR) scheme. Furthermore, understanding the impact of reclamation schemes on algorithm performance enables fair comparison of different algorithms—in our case, lock-free and concurrently readable lists and hash tables.

The results of our analysis indicate that QSBR is, in fact, the scheme best suited to an OS kernel environment. Our performance data from the Linux kernel shows that lockless approaches using QSBR are practical and can outperform locking approaches by a large margin.

We reiterate that blocking reclamation can be useful with non-blocking algorithms: in the absence of thread failure, non-blocking algorithms still benefit from deadlock-freedom, signal handler safety, and avoidance of priority inversion. Nevertheless, one important question remains open, namely, what sort of weakened non-blocking property could be satisfied by a reclamation scheme that avoids the per-element overhead that is incurred by all currently known non-blocking reclamation schemes.

Legal statement

Acknowledgments

We owe thanks to Maged Michael and Keir Fraser for helpful comments on their respective work, to Faith Fich and Cristiana Amza for much helpful advice, and to Dan Frye for his support of this effort. We are indebted to Martin Bligh, Andy Whitcroft, and the ABAT team for access to the 8-CPU machine used in our experiments. Finally, we wish to thank our colleagues at the University of Toronto who suggested several clarifications.

References

[1] S.V. Adve, K. Gharachorloo, Shared memory consistency models: a tutorial, IEEE Comput. 29 (12) (1996) 66–76.

[2] T.E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors, IEEE Trans. Parallel Distrib. Syst. 1 (1) (1990) 6–16.

[3] A. Arcangeli, M. Cao, P.E. McKenney, D. Sarma, Using read-copy update techniques for System V IPC in the Linux 2.5 kernel, in: Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track), USENIX Association, 2003.

[4] E. Berger, K. McKinley, R. Blumofe, P. Wilson, Hoard: a scalable memory allocator for multithreaded applications, in: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 2000.

[5] H.-J. Boehm, An almost non-blocking stack, in: Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 2004.

[6] J. Bonwick, J. Adams, Magazines and Vmem: extending the slab allocator to many CPUs and arbitrary resources, in: USENIX Technical Conference, General Track, 2001.

[7] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.

[8] D.L. Detlefs, P.A. Martin, M. Moir, G.L. Steele Jr., Lock-free reference counting, Distrib. Comput. 15 (4) (2002) 255–271.

[9] K. Fraser, Practical lock-freedom, Ph.D. Thesis, University of Cambridge Computer Laboratory, 2004.

[10] B. Gamsa, O. Krieger, J. Appavoo, M. Stumm, Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system, in: Proceedings of the Third Symposium on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, 1999.

[11] A. Gidenstam, M. Papatriantafilou, H. Sundell, P. Tsigas, Efficient and reliable lock-free memory reclamation based on reference counting, in: Proceedings of the Eighth International Symposium on Parallel Architectures, Algorithms and Networks, IEEE Computer Society, Washington, DC, USA, 2005.

[12] M. Greenwald, Non-blocking synchronization and system design, Ph.D. Thesis, Stanford University, 1999.

[13] M. Greenwald, D. Cheriton, The synergy between non-blocking synchronization and operating system structure, in: Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation, ACM Press, 1996.

[14] R. Gupta, The fuzzy barrier: a mechanism for high speed synchronization of processors, in: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 1989.

[15] T.L. Harris, A pragmatic implementation of non-blocking linked-lists, in: Proceedings of the 15th International Conference on Distributed Computing, Springer, Berlin, 2001.

[16] M. Herlihy, Wait-free synchronization, ACM Trans. Prog. Lang. Syst. 13 (1) (1991) 124–149.

[17] M. Herlihy, A methodology for implementing highly concurrent data objects, ACM Trans. Prog. Lang. Meth. 15 (5) (1993) 745–770.

[18] M. Herlihy, V. Luchangco, M. 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, 2002.

[19] M. Herlihy, V. Luchangco, M. Moir, Obstruction-free synchronization: double-ended queues as an example, in: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society, 2003.

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

[21] L.I. Kontothanassis, R.W. Wisniewski, M.L. Scott, Scheduler-conscious synchronization, ACM Trans. Comput. Syst. 15 (1) (1997) 3–40.

[22] S. Kumar, D. Jiang, R. Chandra, J.P. Singh, Evaluating synchronization on shared address space multiprocessors: methodology and performance, in: Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, ACM Press, New York, NY, USA, 1999.

[23] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. 28 (9) (1979) 690–691.

[24] P.E. McKenney, Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels, Ph.D. Thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[25] P.E. McKenney, Real-time preemption and RCU, available: ⟨http://lkml.org/lkml/2005/3/17/199⟩ March 2005, (Viewed September 5, 2005).

[26] P.E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni, Read-copy update, in: Proceedings of the 2001 Ottawa Linux Symposium, 2001.

[27] P.E. McKenney, D. Sarma, Towards hard realtime response from the Linux kernel on SMP hardware, in: linux.conf.au, Canberra, AU, 2005.

[28] P.E. McKenney, J.D. Slingwine, Read-copy update: using execution history to solve concurrency problems, in: Proceedings of the 1998 International Conference on Parallel and Distributed Computing and Systems, Las Vegas, NV, 1998.

[29] M.M. Michael, Safe memory reclamation for dynamic lock-free objects using atomic reads and writes, in: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, 2002.

[30] M.M. Michael, Hazard pointers: safe memory reclamation for lock-free objects, IEEE Trans. Parallel Distrib. Syst. 15 (6) (2004) 491–504.

[31] M.M. Michael, Scalable lock-free dynamic memory allocation, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2004.

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

[33] Open Source Development Labs, Inc., Database test suite, available: ⟨http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_su%ite/⟩ 2003, (Viewed June 29, 2005).

[34] D. Sarma, P.E. McKenney, Issues with selected scalability features of the 2.6 kernel, in: Proceedings of the 2004 Ottawa Linux Symposium, 2004.

[35] J. Seigh, RCU + SMR for preemptive kernel/user threads, linux-kernel mailing list, available: ⟨http://lkml.org/lkml/2005/5/9/129⟩, 2005.

[36] H. Sundell, Efficient and practical non-blocking data structures, Ph.D. Thesis, Chalmers University of Technology, 2004.

[37] H. Sundell, Wait-free reference counting and memory management, in: Proceedings of the 19th International Parallel and Distributed Processing Symposium, 2005.

[38] H. Sundell, P. Tsigas, Lock-free and practical doubly linked list-based deques using single-word compare-and-swap, in: Proceedings of the Eighth International Conference on Principles of Distributed Systems, 2004.
[39] J.D. Valois, Lock-free linked lists using compare-and-swap, in: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, 1995.

Thomas E. Hart is a Ph.D. student in the Department of Computer Science at the University of Toronto. His research interests are in computer security, automated verification, and synchronization. He holds a B.Sc. in Mathematics and Computer Science from Brandon University (2003) and an M.Sc. in Computer Science from the University of Toronto (2005). He is the holder of an NSERC Canada Graduate Scholarship.

Paul E. McKenney is a Distinguished Engineer at IBM's Linux Technology Center, where he works on parallel real-time synchronization primitives in the Linux kernel. Prior to that, he worked on a parallel UNIX database-server operating system at Sequent Computer Systems, and on Internet and packet-radio congestion-control protocols at SRI International. He holds more than 20 patents and has published more than 30 papers. He holds B.Sc. degrees in Computer Science and in Mechanical Engineering and an M.Sc. degree from Oregon State University (1981 and 1988), and a Ph.D. from Oregon Health and Sciences University (2004).

Angela Demke Brown is an Assistant Professor in the Department of Computer Science at the University of Toronto. Her research interests include operating systems, dynamic compilers, and interpreters. Her focus is on delivering the underlying performance of emerging computer architectures to user applications through the interaction of language tools and operating system services. She holds M.Sc. and Ph.D. degrees in Computer Science from the University of Toronto (1997) and Carnegie Mellon University (2005), respectively.

Jonathan Walpole is a Full Professor in the Computer Science Department at Portland State University. His research interests are in operating systems, distributed systems, and networking. His current research focuses on scalable synchronization mechanisms for shared memory multiprocessor systems. He has published over 100 research papers and has served as a program committee member and reviewer for numerous International scientific conferences and journals. He holds B.Sc. and Ph.D. degrees in Computer Science from Lancaster University, UK (1984 and 1987) and was awarded a Post-Doctoral Fellowship by the UK Science and Engineering Research Council in 1988.