Nonblocking Memory Management Support for Dynamic-Sized Data Structures

MAURICE HERLIHY
Brown University
and
VICTOR LUCHANGCO, PAUL MARTIN, and MARK MOIR
Sun Microsystems Laboratories

Conventional dynamic memory management methods interact poorly with lock-free synchronization. In this article, we introduce novel techniques that allow lock-free data structures to allocate and free memory dynamically using any thread-safe memory management library. Our mechanisms are lock-free in the sense that they do not allow a thread to be prevented from allocating or freeing memory by the failure or delay of other threads. We demonstrate the utility of these techniques by showing how to modify the lock-free FIFO queue implementation of Michael and Scott to free unneeded memory. We give experimental results that show that the overhead introduced by such modifications is moderate, and is negligible under low contention.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Algorithms

Additional Key Words and Phrases: Multiprocessors, nonblocking synchronization, concurrent data structures, memory management, dynamic data structures

1. INTRODUCTION

A *lock-free* concurrent data structure is one that guarantees that if multiple threads concurrently access that data structure, then *some* thread will complete its operation in a finite number of steps, despite the delay or failure of other threads. Lock-free synchronization aims to avoid many problems that are associated with the use of locking, including convoying, susceptibility to failures and delays, and, in real-time systems, priority inversion.

Early work on lock-free synchronization addressed the circumstances under which it can be accomplished at all [Herlihy 1991]. More recent work has focused on general constructions [Anderson and Moir 1999; Greenwald 1999], and on specific data structures [Agesen et al. 2002; Detlefs et al. 2000; Greenwald 1999; Harris 2001; Michael and Scott 1996], and there is increasing interest and success in achieving practical implementations. In fact, support for lock-free

Authors' addresses: M. Herlihy, Computer Science Department, Brown University, Box 1910, Providence, RI 02912; email: herlihy@cs.brown.edu; V. Luchangco, P. Martin, and M. Moir, Sun Microsystems Laboratories, 1 Network Drive, UBUR02-311, Burlington, MA 01803; email: {victor.luchangco,paul.a.martin,mark.moir}@sun.com.

2005 ACM 0734-2071/05/0500-0146

synchronization has recently been incorporated into the Java standard, and new concurrency libraries for the Java programming language include practical lock-free data structures [Lea 2003]. Most recently, there has been a flurry of interest in providing direct hardware support for lock-free synchronization [Martinez and Torrellas 2002; Oplinger and Lam 2002; Rajwar and Goodman 2002] to avoid some of the overhead inherent in software-based solutions. The trend is clear: data structures employing lock-free synchronization are likely to become more common in multiprocessor applications.

In this article, we address a pragmatic issue that has received surprisingly little attention: how to enable lock-free data structures to grow and shrink. For reasons described below, conventional dynamic memory management methods interact poorly with lock-free synchronization. We present alternative mechanisms that allow lock-free data structures to allocate and free memory dynamically using off-the-shelf thread-safe memory management libraries. Our mechanisms are lock-free in the sense that they do not allow a thread to be prevented from allocating or freeing memory by the failure or delay of other threads. Our mechanisms can be integrated with any thread-safe memory management library (although, of course, the ensemble is lock-free only if the library is also lock-free[1]).

Let us review why lock-free synchronization makes dynamic memory management so hard. The key difficulty is that we cannot free a memory block (e.g., a node in a linked list) unless we can guarantee that no thread will subsequently modify that block. Otherwise, a thread might modify the memory block after it has already been reallocated for another purpose, with potentially disastrous results. Furthermore, in some systems, even read-only accesses to freed memory blocks can be problematic: the operating system may remove the page containing the memory block from the thread's address space, rendering the address invalid [Treiber 1986]. In data structures that use locking, a common pattern is to ensure that a thread can acquire a pointer to a particular block only after acquiring a lock, ensuring that only one active pointer exists for that block. In lock-free data structures, by contrast, it may be difficult for a thread that is about to free a block to ensure that no other thread has that pointer in a local variable.

Memory management for lock-free data structures has received surprisingly little attention. (An exception is the recent work of Michael [2002a], who has independently and concurrently[2] developed a technique very similar to ours; see Section 8 for a comparison.) Prior attempts to support explicit memory management for highly-concurrent data structures have significant drawbacks, including restricted flexibility and/or applicability, and unacceptable behavior in the face of thread failures. An ideal solution is to employ garbage collection (GC), and experience [Detlefs et al. 2000] shows that the availability of GC significantly simplifies the design of lock-free dynamic-sized data structures. However, GC is not always available, especially when implementing GC itself.

[1]For example, see Dice and Garthwaite [2002].
[2]Proceedings of PODC 2002 [Herlihy et al. 2002; Michael 2002a].

1.1 Overview

In Section 2, we present a simple example that illustrates the difficulties that arise in implementing a lock-free dynamic-sized data structure. We also show how to overcome these difficulties using the mechanisms presented in this article. We present the API for our mechanisms in Section 3.

In Section 4, we take on a more substantial example. We show how to use our mechanisms to make the lock-free FIFO queue of Michael and Scott [1998] truly dynamic-sized. In the the original implementation, nodes removed from the queue are stored in a special "freelist" private to the queue. These nodes can be reused for the queue, but can never be freed for use by other data structures. As a result, the queue's space consumption is its maximum historical size, rather than its current size. In Section 5, we present experimental results that show that the ability to free unused queue nodes incurs only a modest penalty.

In Section 6, we describe in detail the "Pass The Buck" (PTB) algorithm, which is one way to implement the mechanisms we propose; we also present a simpler but slightly weaker variant on this algorithm, demonstrating that a variety of solutions can be "plugged into" our techniques, with no impact on user code.

Section 7 presents another application of our mechanisms: *single-word lock-free reference counting* (SLFRC) is a technique for transforming certain kinds of lock-free data structure implementations that rely on garbage collection (i.e., they never explicitly free memory) into dynamic-sized data structures.

We present related work in Section 8, and conclusions in Section 9.

The PTB algorithm is subtle and requires careful explanation; a formal proof that it correctly implements our API is included in Appendix B. A formal proof of correctness requires a formal specification of the problem, which we provide in the *Repeat Offender Problem* (ROP), defined in Appendix A. ROP captures the abstract properties that our API is intended to provide. Because PTB can be replaced by any algorithm that correctly implements the API we have specified, we often refer to "ROP solutions" in general—rather than the PTB algorithm specifically—when discussing applications of our mechanisms.

2. SIMPLE EXAMPLE: A LOCK-FREE STACK

To illustrate the problem that we solve in this article, we consider a simple example: a lock-free integer stack implemented using the compare-and-swap (CAS) instruction. We first present a naive stack implementation, and explain two problems with it. We then show how to address these problems using the mechanisms presented in this article. Finally, for the interested reader, we describe the guarantees made by our mechanisms, and argue that these guarantees, combined with the conditions that the programmer must ensure (explained below), are sufficient to achieve the desired behavior. The following preliminaries apply to all of the algorithms presented in this article.

```
struct nodeT {int val; nodeT *next;}
shared variable
    nodeT *TOS initially NULL;

void Push(int v) {
1   nodeT *oldtos,
            *newnode = malloc(sizeof(nodeT));
2   newnode→val = v;
3   do {
4     oldtos = *TOS;
5     newnode→next = oldtos;
6   } while (!CAS(&TOS, oldtos, newnode));
}

int Pop() {
7   nodeT *oldtos, *newtos;
8   do {
9     oldtos = *TOS;
10    if (oldtos == NULL) return "empty";
11    newtos = oldtos→next;
12  } while (!CAS(&TOS, oldtos, newtos));
13  int val = oldtos→val;
14  free(oldtos);
15  return val;
}
```

Fig. 1. Naive stack code.

2.1 Preliminaries

We assume a sequentially consistent shared-memory multiprocessor system [Lamport 1979][3] without garbage collection (GC); memory on the heap is allocated and deallocated using **malloc** and **free**. We further assume that the machine supports a compare-and-swap (CAS) instruction that accepts three arguments: an *address*, an *expected* value and a *new* value. The CAS instruction atomically compares the contents of the address to the expected value and, if they are equal, stores the new value at the address and returns *true*. If the comparison fails, no changes are made to memory, and the CAS instruction returns *false*.

2.2 A Naive Implementation and Its Pitfalls

An obvious implementation approach for our lock-free integer stack is to represent the stack as a linked list of nodes, with a shared pointer—call it `TOS`—that points to the node at the top of the stack. In this approach, pushing a new value involves allocating a new node, initializing it with the value to be pushed and the current top of stack, and using CAS to atomically change `TOS` to point to the new node (retrying if the CAS fails due to concurrent operations succeeding). Popping is similarly simple: we use CAS to atomically change `TOS` to point to the second node in the list (again retrying if the CAS fails), and retrieve the popped value from the removed node. Because the system does not have GC, we must explicitly free the removed node to avoid a memory leak. Code based on this (incorrect) approach is shown in Figure 1.

One problem with the naive stack implementation is that it allows a thread to access a freed node. To see why, observe that a thread p executing the `Pop` code at line 11 accesses the node it previously observed (at line 9) to be at the top of the stack. However, if another thread q executes the entire `Pop` operation between the times p executes lines 9 and 11, then q will free that node (line 14), and p will access a freed node.

[3]We have implemented our algorithms for SPARC®-based machines providing only TSO (Total Store Ordering) [Weaver and Germond 1994]—a memory model that is slightly weaker than sequential consistency—which required additional memory barrier instructions to be included in places.

A more subtle problem, widely known as the ABA problem [Michael and Scott 1998], involves a variable changing from one value (A) to another (B), and subsequently back to the original value (A). The problem is that CAS does not distinguish between this situation and the one in which the variable does not change at all. The ABA problem manifests itself in the naive stack implementation as follows: Suppose the stack currently contains nodes A and B (with node A being at the top of the stack). Further suppose that thread p, executing a `Pop` operation reads a pointer to node A from `TOS` at line 9, reads a pointer from node A to node B at line 11, and prepares to use CAS to atomically change `TOS` from pointing to node A to pointing to node B. Now, let us suspend thread p while thread q executes the following sequence of operations: First, q executes an entire `Pop` operation, removing and freeing node A. Now, `TOS` contains a pointer to node B. Next, q executes another `Pop` operation, and similarly removes and deletes node B. (Now the stack is empty.) Next, q pushes a new value onto the stack, allocating node C for this purpose. Finally, q pushes yet another value onto the stack, and in this last operation, happens to allocate node A again (this is possible because node A was previously freed). Now `TOS` again points to node A, which points to node C. At this point, p resumes execution and executes its CAS, which succeeds in changing `TOS` from pointing to node A to pointing to node B. This is incorrect, as node B has been freed (and may have subsequently been reallocated and reused for a different purpose). Further, note that node C has been lost from the stack. The root of the problem is that p's CAS did not detect that `TOS` had changed from pointing to node A and later changed so that it was again pointing to node A: the dreaded ABA problem.

2.3 Our Mechanisms: `PostGuard` and `Liberate`

In this article, we provide mechanisms that allow us to efficiently overcome both of the problems described above without relying on GC. Proper use of these mechanisms allows programmers to prevent memory from being freed while it might be accessed by some thread. In this subsection, we describe how these mechanisms should be used, and illustrate such use for the stack example.

The basic idea is that before dereferencing a pointer, a thread *guards* the pointer, and before freeing memory, a thread checks whether a pointer to the memory is guarded. For these purposes, we provide two functions,[4] `PostGuard` and `Liberate`. `PostGuard` takes as an argument a pointer to be guarded. `Liberate` takes as an argument a pointer to be checked and returns a (possibly empty) set of pointers that it has determined are safe to free. The pointers returned by `Liberate` are said to be *liberated* when they are returned. Whenever a thread wants to free memory, instead of directly invoking **free**, it passes the pointer to `Liberate`, and then invokes **free** on each pointer in the set returned by `Liberate`.

[4]We use a restricted interface here, which is sufficient for this example. In the next section, we present an interface that is more widely applicable; for example, it provides support for a thread to guard multiple pointers simultaneously.

The most challenging aspect of using our mechanisms is that simply guarding a pointer is not sufficient to ensure that it is safe to dereference that pointer: another thread might liberate and free a pointer after some thread has decided to guard that pointer but before it actually does so. As explained in more detail below, `Liberate` never returns a pointer that is not safe to free, provided the programmer guarantees the following property:

> At the moment that a pointer is passed to `Liberate`, any thread that might dereference this instance[5] of the pointer has already guarded it, and will keep the pointer guarded until after any such dereferencing.

If a thread posts a guard on a pointer, and subsequently determines that the pointer has not been passed to `Liberate` since it was last allocated, then we say that the guard *traps* the pointer until the guard is subsequently posted on another pointer, or removed from this pointer. We have found this terminology useful in talking about algorithms that use our mechanisms. It is easy to see that the programmer can provide the guarantee stated above by ensuring that the algorithm never dereferences a pointer that is not trapped.

We have found that the following simple and intuitive pattern is often useful for achieving the required guarantee when implementing shared data structures: A thread passes a pointer to `Liberate` only after it has determined that the memory block to which it points has been removed from the shared data structure (and will not be put in the shared data structure again unless it is liberated, freed, and subsequently reallocated). Given this, whenever a thread reads a pointer from the data structure in order to dereference it, the thread uses `PostGuard` to post a guard on that pointer, and then attempts to determine that the memory block is still in the data structure. If it is, then the pointer has not yet been passed to `Liberate`, so it is safe to dereference the pointer; if not, the thread retries. Determining whether a block is still in the data structure may be as simple as rereading the pointer (for example, in the stack example presented next, we reread `TOS` to ensure that the pointer is the same as the one we guarded; see lines 9c and 9d in Figure 2). In other cases, it may be somewhat more complicated; one such example is presented in Section 4.

2.4 Using Our Mechanisms to Fix the Naive Stack Algorithm

In Figure 2, we present modified stack code that uses `PostGuard` and `Liberate` to overcome the problems with the naive stack algorithm presented earlier. To see how the modified code makes the required guarantee, suppose that a thread p passes a pointer to node A to `Liberate` (line 14b) at time t. Prior to t, p changed `TOS` to a node other than node A or to **null** (line 12), and thereafter, until node A is liberated, freed and reallocated, `TOS` does not point to node A. Suppose that after time t, another thread q dereferences (at line 11 or line 13) that instance of a pointer to node A. When q last executes line 9d, at time t',

[5]A particular pointer can be repeatedly allocated and freed, resulting in multiple *instances* of that pointer. Thus, this property refers to threads that might dereference a pointer before that same pointer is subsequently allocated again.

```
struct nodeT int val; nodeT *next;
shared variable nodeT *TOS
                              initially NULL;

void Push(int v) {
1   nodeT *oldtos,
        *newnode = malloc(sizeof(nodeT));
2   newnode→val = v;
3   do {
4     oldtos = *TOS;
5     newnode→next = oldtos;
6   } while (!CAS(&TOS, oldtos, newnode));
}

int Pop() {
7     nodeT *oldtos, *newtos;
8     do {
9a      do {
9b        oldtos = *TOS;
9c        PostGuard(oldtos);
9d      } while (*TOS != oldtos);
10      if (oldtos == NULL) return "empty";
11      newtos = oldtos→next;
12    } while (!CAS(&TOS, oldtos, newtos));
13    int val = oldtos→val;
14a   PostGuard(NULL);
14b   for each n in Liberate(oldtos)
14c     free(n);
15    return val;
}
```

Fig. 2. Modified stack code.

prior to dereferencing the pointer to node A, q sees `TOS` pointing to node A. Therefore, t' precedes t. Prior to t', q guarded its pointer to node A (line 9c), and keeps guarding that pointer until after it dereferences it, as required (note that the pointer is guarded until q executes line 14a).

2.5 Guarantees of `Liberate`

While the above descriptions are sufficient to allow a programmer to correctly apply our mechanisms to achieve dynamic-sized data structures, it may be useful to understand in more detail the guarantees that are provided by the `Liberate` function. Below we describe those guarantees, and argue that they are sufficient, when `PostGuard` and `Liberate` are used properly as described above, to prevent freed pointers from being dereferenced.

We say that a pointer *begins escaping* when `Liberate` is invoked with that pointer. Every liberated pointer—that is, every pointer in the set returned by a `Liberate` invocation—is guaranteed to have the following properties:

—It previously began escaping.
—It has not been liberated (by any `Liberate` invocation) since it most recently began escaping.
—It has not been guarded continuously by any thread since it most recently began escaping.

The first two conditions guarantee that every instance of a pointer is freed at most once. They are sufficient for this purpose because threads only pass pointers to `Liberate` when they would have, in the naive code, freed the pointers, and threads free only those pointers returned by `Liberate` invocations.

The last condition guarantees that a pointer is not liberated while it might still be dereferenced. To see that this last condition is sufficient, recall that the programmer must guarantee that any pointer passed to `Liberate` at time t will be dereferenced only by threads that have already guarded the pointer at time t and will keep the pointer guarded continuously until after such dereferencing. The last condition prevents the pointer from being liberated while any such thread exists.

```
typedef int guard;
typedef (void *) ptr_t;

void PostGuard(guard g, ptr_t p);

guard HireGuard();

void FireGuard(guard g);

set[ptr_t] Liberate(set[ptr_t] S);
```

Fig. 3. An API for guarding and liberating.

3. A REPRESENTATIVE API

In this section, we present an application programming interface (API) for the guarding and liberating mechanisms illustrated in the previous section. This API is more general than the one used in the previous section. In particular, it allows threads to guard multiple pointers simultaneously.

Our API uses an explicit notion of *guards*, which are *posted* on pointers. In this API, a thread invokes `PostGuard` with both the pointer to be guarded and the guard to post on the pointer. We represent a guard by an `int`. A thread can guard multiple pointers by posting different guards on each pointer. A thread may *hire* or *fire* guards dynamically, according to the number of pointers it needs to guard simultaneously, using the `HireGuard` and `FireGuard` functions. We generalize `Liberate` to take a *set* of pointers as its argument, so that many pointers can be passed to `Liberate` in a single invocation. The signatures of all these functions are shown in Figure 3. Below we examine each function in more detail.

3.1 Detailed Function Descriptions

`void PostGuard(guard g, ptr_t p)`

Purpose. Posts a guard on a pointer.

Parameters. The guard `g` and the pointer `p`; `g` must have been hired and not subsequently fired by the thread invoking this function.

Return value. None.

Remarks. If `p` is `NULL` then `g` is not posted on any pointer after this function returns. If `p` is not `NULL`, then `g` is posted on `p` from the time this function returns until the next invocation of `PostGuard` with the guard `g`. Note that it is *not* guaranteed that `g` is *posted on* `p` between the invocation and return of `PostGuard`. Thus, if `PostGuard(g,p)` is invoked at time t and again at a later time t' (with no intervening operations concerning guard g), we cannot claim that `g` has been posted continuously on `p` since time t.

`guard HireGuard()`

Purpose. "Acquire" a new guard.

Parameters. None.

Return value. A guard.

Remarks. The guard returned is *hired* when it is returned. When a guard is hired, either it has not been hired before, or it has been fired since it was last hired.

void FireGuard(guard g)

Purpose. "Release" a guard.

Parameters. The guard g to be fired; g must have been hired and not subsequently fired by the thread invoking this function.

Return value. None.

Remarks. g is *fired* when `FireGuard(g)` is invoked.

set[ptr_t] Liberate(set[ptr_t] S)

Purpose. Prepare pointers to be freed.

Parameters. A set of pointers to liberate. Each pointer in this set must either never have begun escaping or have been liberated since it most recently began escaping. That is, no pointer in this set was in any set passed to a previous `Liberate` invocation since it was most recently in the set returned by some `Liberate` operation.

Return value. A set of liberated pointers.

Remarks. The pointers in the set `S` *begin escaping* when `Liberate(S)` is invoked. The pointers in the set returned are *liberated* when the function returns. Each liberated pointer must have been contained in the set passed to some invocation of `Liberate`, and not in the set returned by any `Liberate` operation after that invocation. Furthermore, `Liberate` guarantees for each pointer that it returns that no guard has been posted continuously on the pointer since it was most recently passed to some `Liberate` operation.

3.1.1 *Comments on API Design.* We could have rolled the functionality of hiring and firing guards into the PostGuard operation. Instead, we kept this functionality separate to allow implementations to make PostGuard, the most common operation, as efficient as possible. This separation allows the implementation more flexibility in managing resources associated with guards because the cost of hiring and firing guards can be amortized over many PostGuard operations.

In some applications, it may be desirable to be able to quickly "mark" a value for liberation, without doing any of the work of liberating the value. (Consider, for example, an interactive system in which user threads should not execute relatively high-overhead "administrative" work such as liberating values, but additional processor(s) may be available to perform such work.) We did not provide such an operation, as it is straightforward to communicate such values to a worker thread that invokes Liberate—our mechanisms do not need to know anything about this.

3.1.2 *Progress Guarantees.* Two kinds of progress guarantees are desirable. First, we would like every invocation of an operation to eventually return. Minimally, any implementation should ensure that a thread failure cannot prevent progress by operations being executed by other threads. (Even if it is

acceptable to assume that threads do not fail in a particular application, long delays—for example, due to preemption—should not prevent progress by other threads during those delays.) A *wait-free* implementation guarantees for every operation that it completes provided the thread executing it does not fail. A *lock-free* implementation provides a slightly weaker guarantee: if a thread executes an operation and does not fail, then *some* operation eventually completes.

In Section 6, we present a wait-free implementation of our API. This implementation depends on a compare-and-swap (CAS) instruction that can atomically manipulate a pointer and a version number. While such an instruction is widely available in 32-bit architectures, it is not yet available in some 64-bit architectures, so the wait-free implementation is not universally applicable. Therefore, we also explain how to modify the algorithm so that it does not require such an operation; the resulting implementation is technically only lock-free, but in practice makes the same guarantees as the wait-free implementation.

The second kind of desirable progress guarantee is *value progress*: if a pointer begins escaping at some point and is eventually not trapped by any guard, we would like the pointer to eventually be liberated—that is, to eventually be returned by some `Liberate` operation. We call this type of property a value progress condition because it refers to progress for a particular pointer value, not progress for a thread executing an operation.

It is usually not possible to make this ideal value progress guarantee because of the nature of asynchronous systems: without special operating system support it is generally impossible to distinguish between a thread being delayed and failing. As a result, if a thread executing a `Liberate` operation fails, we cannot free the pointers it was responsible for when it failed, because if the thread is in fact only delayed, it might later free the pointers *again*.

Different implementations of our API may provide different value progress guarantees under different assumptions, and the choice of implementations to use for a particular application will depend on the validity of assumptions, and tradeoffs between performance and strength of progress guarantees. While it is inevitable that thread failures can prevent *some* pointers from being liberated, for most applications, we must guarantee some limit on the number of such pointers; otherwise we run the risk of unbounded memory leaks. The implementation presented in Section 6 satisfies the Value Progress condition specified in Section A.1, which implies that it does not allow unbounded memory leaks even if some (finite) number of threads fail. Because the Value Progress condition is stated for our API, rather than for a specific implementation, it is somewhat difficult to understand the implications of meeting this condition. In Section 6, we discuss the precise conditions under which a thread failure can prevent a pointer from being liberated in our implementation.

4. DYNAMIC-SIZED LOCK-FREE QUEUES

In this section, we present two dynamic-sized lock-free queue implementations based on a widely used lock-free queue algorithm by Michael and Scott [1998]. In Michael and Scott's algorithm (M&S), the queue is represented by a linked

list, and nodes that have been dequeued are placed in a "freelist" implemented in the style of Treiber [1986]. (In the remainder of the paper, we refer to freelists as "memory pools" in order to avoid confusing "freeing" a node—by which we mean returning it to the memory allocator through the **free** library routine—and placing a node on a freelist.) In this approach, rather than freeing nodes to the memory allocator when they are no longer required, we place them in a memory pool from which new nodes can be allocated later. An important disadvantage of this approach is that data structures implemented this way are not truly dynamic-sized: after they have grown large and subsequently shrunk, the memory pool contains many nodes that cannot be reused for other purposes, cannot be coalesced, etc. We show how to modify M&S to achieve true dynamic-sized queue implementations.

Our two queue implementations achieve dynamic sizing in different ways. Algorithm 1 eliminates the memory pool, invoking the standard **malloc** and **free** library routines to allocate and deallocate nodes of the queue. Algorithm 2 does use a memory pool, which reduces that cost of memory allocation and deallocation compared with using **malloc** and **free** directly. However, unlike M&S, the nodes in the memory pool can be freed to the system.

We present our algorithms by giving "generic code" for the M&S algorithm (Figure 4). This code invokes procedures that must be instantiated to achieve full implementations. We give the instantiations for the original M&S algorithm and our new algorithms.

4.1 Michael and Scott's Algorithm

The generic code in Figure 4 invokes four procedures, shown in italics in the figure. We obtain the original M&S algorithm by instantiating these procedures with those shown in Figure 5. The *allocNode* and *deallocNode* procedures use a memory pool. The *allocNode* procedure removes and returns a node from the memory pool if possible and calls **malloc** if the pool is empty. The *deallocNode* procedure puts the node being deallocated into the memory pool. As stated above, nodes in the memory pool cannot be freed to the system. Michael and Scott do not specify how nodes are added to and removed from the memory pool. M&S does not use guards, so *GuardedLoad* is an ordinary load and *Unguard* is a no-op.

We do not describe M&S in detail; see Michael and Scott [1998] for such details. We also do not argue it is correct; we refer the interested reader to Doherty et al. [2004a] for a formal proof of a slight variation on this algorithm. Below we discuss the aspects of M&S that are relevant to memory management. In M&S, although nodes in the memory pool have been deallocated, they cannot be freed to the system because some thread may still intend to perform a CAS (line 12) on the node. As discussed in Section 1, various problems can arise from accesses to memory that has been freed. Thus, although it is not discussed at length in Michael and Scott [1998], the use of the memory pool is necessary for correctness. Because Enqueue may reuse nodes from the memory pool, M&S uses version numbers to avoid the ABA problem, in which a CAS succeeds even though the pointer it accesses has changed because the node pointed to was deallocated and then subsequently allocated. The version numbers are stored

```
struct pointer_t { node_t *ptr; int version; }
struct node_t { int value; pointer_t next; }
struct queue_t { pointer_t Head, Tail; }

queue_t *newQueue() {
    queue_t *Q = malloc(sizeof(queue_t));
    node_t *node = allocNode();
    node→next.ptr = null;
    Q→Head.ptr = Q→Tail.ptr = node;
    return Q;
}

bool Enqueue(queue_t *Q, int value) {
1   node_t *node = allocNode();
2   if (node == null)
3       return FALSE;
4   node→value = value;
5   node→next.ptr = null;
6   while (TRUE) {
7       pointer_t tail;
8       GuardedLoad(0,&Q→Tail, &tail);
9       pointer_t next = tail.ptr→next;
10      if (tail == Q→Tail) {
11          if (next.ptr == null) {
12              if (CAS(&tail.ptr→next, next,
                        ⟨node, next.version + 1⟩))
13                  break;
14          } else
15              CAS(&Q→Tail, tail,
                    ⟨next.ptr, tail.version + 1⟩)
        }
    }
16  CAS(&Q→Tail, tail,
                ⟨node, tail.version + 1⟩)
17  Unguard(0);
18  return TRUE;
}
```

```
bool Dequeue(queue_t *Q, int *pvalue) {
19  while (TRUE) {
20      pointer_t head;
21      GuardedLoad(0, &Q→Head, &head);
22      pointer_t tail = Q→Tail;
23      pointer_t next;
24      GuardedLoad(1, &head.ptr→next, &next);
25      if (head == Q→Head) {
26          if (head.ptr == tail.ptr) {
27              if (next.ptr == null) {
28                  Unguard(0);
29                  Unguard(1);
30                  return FALSE;
                }
31              CAS(&Q→Tail, tail,
                        ⟨next.ptr, tail.version + 1⟩)
32          } else {
33              *pvalue = next.ptr→value;
34              if (CAS(&Q→Head, head,
                        ⟨next.ptr, head.version + 1⟩))
35                  break;
            }
        }
    }
36  Unguard(0);
37  Unguard(1);
38  deallocNode(head.ptr);
39  return TRUE;
}
```

Fig. 4. Generic code for M&S.

```
node_t *allocNode() {
1   if (memory pool is empty)
2       return malloc(sizeof(node_t));
3   else {
4       return node removed
                    from memory pool;
    }
}

void deallocNode(node_t *n) {
5   add n to memory pool
}
```

```
void GuardedLoad(int h, pointer_t *s, pointer_t *t) {
6   *t = *s;
7   return;
}

void Unguard(int h) {
8   return;
}
```

Fig. 5. Auxiliary procedures for M&S.

```
node_t *allocNode() {
1  return malloc(sizeof(node_t));
}

void deallocNode(node_t *n) {
2  for each m ∈ Liberate({n})
3      free(m);
}
```

```
void GuardedLoad(int g, pointer_t *s, pointer_t *t) {
4  while (TRUE) {
5      *t = *s;
6      if (t→ptr == null)
7          return;
8      PostGuard(guards[p][g], t→ptr);
9      if (*t == *s)
10         return;
   }
}

void Unguard(int g) {
11 PostGuard(guards[p][g], null);
}
```

Fig. 6. Auxiliary procedures for Algorithm 1. Code is shown for thread p.

with each pointer and are atomically incremented each time the pointer is modified. This causes such "late" CAS's to fail, but it does not prevent them from being attempted.

The queue is represented by two node pointers: the `Head`, from which nodes are dequeued, and the `Tail`, where nodes are enqueued. The `Head` and `Tail` pointers are never **null**; the use of a "dummy" node ensures that the list always contains at least one node. When a node is deallocated, no path exists from either the `Head` or the `Tail` to that node. Furthermore, such a path cannot subsequently be established before the node is allocated again in an Enqueue operation. Therefore, if such a path exists, then the node is in the queue. Also, once a node is in the queue and its `next` field has become non-**null**, its `next` field cannot become **null** again until the node is initialized by the next Enqueue operation to allocate that node. These properties are used to argue the correctness of our dynamic-sized variants of M&S.

4.2 Algorithm 1

Algorithm 1 closely follows the lock-free stack example in Section 2, eliminating the memory pool and using **malloc** and **free** directly to allocate and deallocate memory. As in the previous example, this use of our API also eliminates the ABA problem, and thus, the need for version numbers. Thus, Algorithm 1 can be used on systems that support CAS only on pointer-sized values.[6]

Algorithm 1 is achieved by instantiating the generic code with the procedures shown in Figure 6. As in Section 2, we use ROP to ensure that every pointer that may be dereferenced is trapped by some guard. We assume that before accessing the queue, each thread p has hired two guards and stored identifiers for these guards in `guards`[p][0] and `guards`[p][1]. The *allocNode* procedure simply invokes **malloc**. However, because some thread may have a pointer to a node being deallocated, *deallocNode* cannot simply invoke **free**. Instead, *deallocNode* passes the node being deallocated to Liberate and then frees any nodes returned by Liberate. The properties of ROP ensure that a node is never returned by an invocation of Liberate while some thread might still access that node.

[6]This of course requires that we use an implementation of our techniques that is applicable in such systems. In Section 6, we present two implementations, one of which has this property.

The *GuardedLoad* procedure loads a value from the address specified by its second argument and stores the value loaded in the address specified by its third argument. The purpose of this procedure is to ensure that the value loaded is guarded by the guard specified by the first argument *before* the value is loaded. This is accomplished by a lock-free loop that retries if the value loaded changes after the guard is posted (**null** values do not have to be guarded, as they will never be dereferenced). As explained below, *GuardedLoad* helps ensure that guards are posted soon enough to trap the pointers they guard, and therefore to prevent the pointers they guard from being freed prematurely. The *Unguard* procedure removes the specified guard.

4.2.1 *Correctness Argument for Algorithm* 1. Algorithm 1 is equivalent to M&S except for issues involving memory allocation and deallocation. (To see this, observe that *GuardedLoad* implements an ordinary load, and *Unguard* does not affect any variables of the M&S algorithm.) Therefore, we need only argue that no instruction accesses a freed node. Because nodes are freed only after being returned by Liberate, it suffices to argue for each access to a node,[7] that, at the time of the access, a pointer to the node has been continuously guarded since some point at which the node was in the queue (i.e., a node is accessed only if it is trapped). As discussed earlier, if there is a path from either `Head` or `Tail` to a node, then the node is in the queue. We can exploit code already included in M&S, together with the specialization code in Figure 6, to detect the existence of such paths.

We first consider the access at line 9 of Figure 4. In this case, the pointer to the node being accessed was acquired from the call to *GuardedLoad* at line 8. Because the pointer is loaded directly from `Tail` in this case, the load in line 9 of Figure 6 serves to observe a path (of length one) from `Tail` to the accessed node. The argument is similarly straightforward for the access at line 12 and the accesses in *GuardedLoad* when invoked from line 24.

The argument for the access at line 33 is not as simple. First, observe that the load at line 9 of *GuardedLoad* (in the call at line 24 of Figure 4) determines that there is a pointer from the node specified by `head.ptr` to the node accessed at line 33. Then, the test at line 25 determines that there is a pointer from `Head` to the node specified by `head.ptr` . If these two pointers existed simultaneously at some point between the guard being posted as a result of the call at line 24 and the access at line 33, then the required path existed. As argued above, the node pointed to by `head.ptr` is guarded and was in the queue at some point since the guard was posted in the call to *GuardedLoad* at line 21, and this guard is not removed or reposted before the execution of line 33. Therefore, by the properties of ROP, this node cannot be freed and reallocated in this interval. Also, in the M&S algorithm, a node that is dequeued does not become reachable from `Head` again before it has been reallocated by an Enqueue. Therefore, the load at line 25 confirmed that `Head` contained the same value continuously

[7]As stated earlier, it is sometimes possible to determine that a node will not be freed before certain accesses without using ROP. The accesses in lines 4 and 5 of Figure 4 are examples because they access a newly-allocated node that will not be freed before these statements are executed. Therefore, there is nothing to argue for these accesses.

since the execution of line 21. This in turn implies that the two pointers existed simultaneously at the point at which the load in *GuardedLoad* invoked from line 24 was executed.[8] This concludes our argument that Algorithm 1 never accesses freed memory.

4.2.2 *Eliminating Version Numbers in Algorithm* 1. We now argue that the version numbers for the node pointers are unnecessary in Algorithm 1. In addition to eliminating the overhead involved with managing them, eliminating these version numbers allows the algorithm to be used in systems that support CAS only on pointer-sized values.

By inspecting the code for Algorithm 1, we can see that the only effect of the version numbers is to make some comparisons fail that would otherwise have succeeded. These comparisons are always between a shared variable V and a value previously read from V. The comparisons would fail anyway if V's pointer component had changed, and would succeed in any case if V had not been modified since the V was read. Therefore, version numbers change the algorithm's behaviour only in the case that a thread p reads value A from V at time t, V subsequently changes to B, and later still, at time t', V changes back to a value that contains the same pointer component as A, and p compares V to A. With version numbers, the comparison would fail, and without them it would succeed. We begin by arguing that version numbers never affect the outcome of comparisons other than the one in line 9 of Figure 6; we deal with that case later.

We first consider cases in which A's pointer component is non-**null**. It can be shown for each shared pointer variable V in the algorithm that the node pointed to by A is freed and subsequently reallocated between times t and t' in this case (see Lemma 1 below). Furthermore, it can be shown that each of the comparisons mentioned above occurs only if a guard was posted on A before time t and is still posted when the subsequent comparison is performed, and that the value read from A was in the queue at some point since the guard was posted when the comparison is performed (see Lemma 2 below). Because ROP prohibits nodes from being returned by Liberate (and therefore from being freed) in this case, this implies that these comparisons never occur in Algorithm 1.

We next consider the case in which A's pointer component is **null**. The only comparison of a shared variable to a value with a **null** pointer component is the comparison performed at line 12 (because the `Head` and `Tail` never contain **null** and therefore neither do the values read from them). As argued earlier, the access at line 12 is performed only when the node being accessed is trapped. Also, as discussed earlier, the `next` field of a node in the queue does not become **null** again until the node is initialized by the next Enqueue operation to allocate that node. However, ROP ensures that the node is not returned from Liberate, and is therefore not subsequently freed and reallocated, before the guard is removed or reposted.

[8]The last part of this argument can be made much more easily by observing that the version number (discussed next) of `Head` did not change. However, we later observe that the version numbers can be eliminated from Algorithm 1, so we do not want to rely on them in our argument.

It remains to consider the comparison in line 9 of Figure 6, which *can* have a different outcome if version numbers are used than it would if they were not used. We argue that this does not affect the externally observable behaviour of the *GuardedLoad* procedure, and therefore does not affect correctness. The only property of the *GuardedLoad* procedure on which we have depended for our correctness argument is the following: *GuardedLoad* stores a value v in the location pointed to by its third argument such that v was in the location pointed to by *GuardedLoad*'s second argument at some point during the execution of *GuardedLoad* and that a guard was posted on (the pointer component of) v before that time and has not subsequently been reposted or removed. It is easy to see that this property is guaranteed by the *GuardedLoad* procedure, with or without version numbers. This concludes our informal argument that version numbers are not necessary in Algorithm 1. The following two lemmas support the argument above.

LEMMA 1. *Suppose that in an execution of Algorithm* 1*, a thread p reads a pointer value A from some shared pointer variable V at time t, V subsequently changes to some other value B, and later still, at time t′, p again observes V containing the same pointer component as A has. Then if A's pointer component is non-***null***, the node pointed to by it is deallocated and subsequently reallocated between times t and t′.*

PROOF SKETCH. This lemma follows from properties of the original Michael and Scott algorithm; here we argue informally based on intuition about that algorithm. The basic intuition is that a node pointed to by either the `Head` or the `Tail` will not be pointed to again by that variable before the node is allocated by an Enqueue operation, which cannot happen before the node has been freed. For the `next` field of nodes, it is easy to see that the only modifications to these fields are the change from **null** to some non-**null** node pointer (line 1) and the initialization to **null** (line 12). Thus, if A is a non-**null** pointer value, then the node pointed to by A must be deallocated and subsequently reallocated before A is stored into the `next` field of any node. □

LEMMA 2. *Suppose that in an execution step of Algorithm* 1 *other than line* 9 *of Figure* 6*, a thread p compares a shared variable V to a value A, which p read from V at a previous time t. Further suppose that the pointer component of A is non-***null***. Then p posts a guard on the pointer component of A before time t and the guard is not reposted or removed before the comparison. Furthermore, at some time at or after t and before the comparison occurs, it is the case that this value has not been passed to* Liberate *since it was last allocated (which implies that the value is trapped when the comparison occurs).*

PROOF SKETCH. First, M&S has the property that a node is never deallocated while a path from either the `Head` or the `Tail` to the node exists. Thus, in Algorithm 1, if such a path is determined to exist, then a pointer to the node has not been passed to Liberate since it was last allocated. Therefore, to show that a node pointer is trapped, it suffices to show that such a path exists at some point after the guard is posted. The guards posted as a result of

```
pointer_t Pool;

node_t *allocNode() {
1  pointer_t oldPool, newPool;
2  while (TRUE) {
3      GuardedLoad(0, &Pool, &oldPool);
4      if (oldPool.ptr == null) {
5          Unguard(0);
6          return malloc(sizeof(node_t));
       }
7      newPool = oldPool.ptr→next;
8      Unguard(0);
9      newPool.version = oldPool.version + 1;
10     if (CAS(&Pool, oldPool, newPool)) {
11         return oldPool.ptr;
       }
   }
}

void deallocNode(node_t *n) {
12 pointer_t oldPool, newPool;
13 while (TRUE) {
14     oldPool = Pool;
15     n→next.ptr = oldPool.ptr;
16     newPool.ptr = n;
17     newPool.version = oldPool.version + 1;
18     if (CAS(&Pool, oldPool, newPool))
19         return;
   }
}
```

Fig. 7. Revised *allocNode* and *deallocNode* procedures for Algorithm 2 (*GuardedLoad* and *Unguard* are unchanged from Figure 6).

the calls to *GuardedLoad* at lines 8 and 21 are therefore always determined to be trapping their respective values before *GuardedLoad* returns (by the test at line 9). This is because in these cases *GuardedLoad* loads directly from the `Tail` and the `Head`, so a path is trivially observed in these cases. Therefore, the lemma holds for the comparisons on lines 10, 15, 16, 25, 31 (observe that head.`ptr` = tail.`ptr` holds when line 31 is executed), and 34. The comparison at line 12 always compares a value with a **null** pointer component, so the lemma holds vacuously in that case. □

4.3 Algorithm 2

One drawback of Algorithm 1 is that every Enqueue and Dequeue operation involves a call to the **malloc** or **free** library routine,[9] introducing significant overhead. In addition, every Dequeue operation invokes Liberate, which is also likely to be expensive. Algorithm 2 overcomes these disadvantages by reintroducing the memory pool. However, unlike the M&S algorithm, nodes in the memory pool of Algorithm 2 can be freed to the system.

Algorithm 2 is achieved by instantiating the generic code in Figure 4 with the same *GuardedLoad* and *Unguard* procedures used for Algorithm 1 (see Figure 6) and the *allocNode* and *deallocNode* procedures shown in Figure 7. As in the original M&S algorithm, the *allocNode* and *deallocNode* procedures, respectively, remove nodes from and add nodes to the memory pool. Unlike the original algorithm, however, the memory pool is implemented so that nodes can be freed. Thus, by augmenting Algorithm 2 with a policy that decides between freeing nodes and keeping them in the memory pool for subsequent use, a truly dynamic-sized implementation can be achieved.

The procedures in Figure 7 use a linked list representation of a stack for a memory pool. This implementation extends Treiber's straightforward

[9]The invocation of the **free** routine for a dequeued node may be delayed if that node is trapped when it is dequeued. However, it will be freed in the Dequeue operation of some later node.

implementation [Treiber 1986] by guarding nodes in the pool before accessing them; this allows us to pass removed nodes to Liberate and to free them when returned from Liberate without the risk of a thread accessing a node after it has been freed. Our memory pool implementation is described in more detail below.

The node at the top of the stack is pointed to by a global variable `Pool`. We use the `next` field of each node to point to the next node in the stack. The *deallocNode* procedure uses a lock-free loop; each iteration uses CAS to attempt to add the node being deallocated onto the top of the stack. As in Treiber's implementation, a version number is incremented atomically with each modification of the `Pool` variable to avoid the ABA problem.

The *allocNode* procedure is more complicated. In order to remove a node from the top of the stack, *allocNode* must determine the node that will become the new top of the stack. This is achieved by reading the `next` field of the node that is currently at the top of the stack. As before, we use ROP to protect against the possibility of accessing (at line 7) a node that has been freed. Therefore, the node at the top of the stack is guarded and then confirmed by the *GuardedLoad* call at line 3. As in the easy cases discussed above for Algorithm 1, the confirmation of the pointer loaded by the call to *GuardedLoad* establishes that the pointer is trapped, because a node will not be passed to Liberate while it is still at the head of the stack.

We have not specified when or how nodes are passed to Liberate. There are many possibilities and the appropriate choice depends on the application and system under consideration. One possibility is for the *deallocNode* procedure to liberate nodes when the size of the memory pool exceeds some fixed limit. Alternatively, we could have an independent "helper" thread that periodically checks the memory pool and decides whether to liberate some nodes in order to reduce the size of the memory pool. Such decisions could be based on the size of the memory pool or on other criteria. There is no need for the helper thread to grow the memory pool because this will occur naturally: when there are no nodes in the memory pool, *allocNode* invokes **malloc** to allocate space for a new node.

Observe that Algorithm 2 cannot be used in systems in which pointers are 64 bits and CAS can access only 64 bits atomically, because of the version number required by Treiber's algorithm. Doherty et al. [2004b] present a "64-bit-clean" freelist implementation that overcomes this problem. While this freelist is somewhat more expensive than that of Treiber, this cost can be amortized over several operations by some amount of per-thread buffering, for example, as discussed further in Sections 5 and 8.

5. PERFORMANCE EXPERIMENTS

We now present the results of our performance experiments, which show that the cost of being able to reclaim memory used by Michael and Scott's lock-free FIFO queue is negligible, provided contention for the queue is not too high, and that it is modest even under high contention. Our experiments are quite conservative, in that they do not incorporate obvious optimizations to reduce synchronization and contention costs, some of which we discuss later.

We ran experiments on several multiprocessor machines, with qualitatively similar results on all of them. Below we present representative results from two machines running Solaris™ 8: a Sun E6500 with 16 400 MHz UltraSPARC® II processors and 16 GB of memory, and a Sun E10000 with 64 466 MHz UltraSPARC® II processors and 64 GB of memory. In all of our experiments, each point plotted represents the average execution time over three trials.

We compared Michael and Scott's algorithm (M&S) against our dynamic-sized versions of their algorithm in several configurations. In each configuration, we used the *Pass The Buck* algorithm presented in the next section to solve the Repeat Offender Problem. One configuration was achieved by instantiating the generic algorithm in Figure 4 with the procedures shown in Figure 6; this version liberates each node as it is removed from the queue. We expected this simplistic version to perform poorly because it invokes Liberate for every dequeue and uses **malloc** and **free** for every node allocation and deallocation. It did not disappoint! We therefore concentrate our presentation on M&S and the two configurations described below.

In both configurations, we use the auxiliary procedures shown in Figure 7. That is, dequeued nodes are placed in a memory pool. In the first configuration, as in M&S, the nodes are never actually freed, so these tests measure the cost of being *able* to free queue nodes, without actually doing so. This configuration gives an indication of how the modified queue performs when no Liberate work is being done (e.g., in a phase of an application in which we expect the queue size to be relatively stable or growing). The second configuration is the same as the first, except that we also create an additional thread that repeatedly checks the size of the memory pool, and if there are more than ten nodes,[10] removes some nodes, passes them to Liberate, and then frees the set of values returned by Liberate. To be conservative, we made this thread run repeatedly in a loop, with no delays between loop iterations. In practice, we would likely run this thread only occasionally in order to make sure that the memory pool did not grow too large.

Our first experiment tested the scalability of the algorithms with the number of threads; this experiment was conducted on the 16-processor machine. The threads performed 2,000,000 queue operations in total (each performing an approximately equal number of operations). We started with an empty queue, and each thread chose at random between Enqueue and Dequeue for each operation. In this experiment, we tested the algorithm under maximum contention; the threads executed the queue operations repeatedly with no delay between operations. The results are shown in Figure 8(a). Qualitatively, all three configurations behaved similarly. The performance became worse going from one thread to two; this is explained by the fact that a single thread runs on a single processor and therefore has a low cache miss rate, while multiple threads run on different processors, so each thread's cache miss rate is significantly higher. As the number of threads increases beyond two, there is some initial improvement gained by executing the operations in parallel. However, performance starts to

[10]To facilitate this check, we modified the code of Figure 7 to include a count field in the header node of the memory pool.

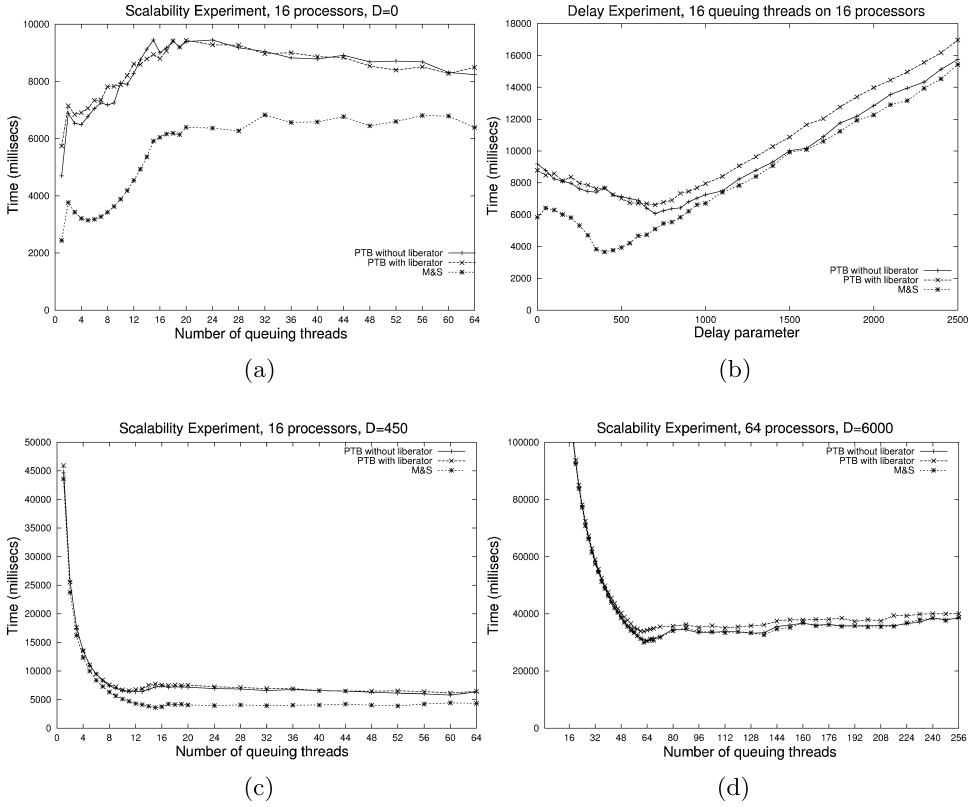

Fig. 8. Performance experiments.

degrade as the number of threads continues to increase. This is explained by the increased contention on the queue (more operations have to retry because of interference with concurrent operations). After the number of threads exceeds the number of processors, all of the configurations perform more or less the same for any number of threads.

While the three configurations are qualitatively similar, the M&S algorithm performs significantly better than either of the other two configurations. This difference is primarily due to the need to post guards, which we cannot avoid using our approach, and the fact that every Dequeue operation immediately puts the removed node into a single memory pool that is shared amongst all threads. A variety of straightforward and standard techniques are applicable to reduce the latter cost. For example, we could use multiple memory pools to reduce contention and/or buffer removed nodes on a per-thread basis to avoid memory pool access for most Dequeue operations. Tradeoffs regarding the size of per-thread buffers are discussed in Section 8.

The results discussed so far compare the algorithms under maximum contention, but it is more realistic to model some non-queue activity between each queue operation. Figure 8(b) shows the results of an experiment we conducted to study the effects of varying the amount of time between queue operations.

In this experiment, we ran 16 threads on the 16-processor machine, and varied a delay parameter D, which controlled the delay between operations as follows: After each queue operation, each thread chose at random[11] a number between 90% and 110% of D, and then executed a short loop that many times; the loop simply copied one local integer variable to another. The results show that performance initially improves, despite the increase in the amount of work done between operations increasing: a clear indication that this delay reduced contention on the queue significantly. The effect of this contention reduction comes sooner and is more dramatic for the M&S configuration than for either of the other two. We believe that this is because the additional work of posting guards on values already serves to reduce contention on the queue variables in the other two configurations. After contention is reduced sufficiently that it is no longer a factor, the time for all of the algorithms increases linearly with the delay parameter. This happened at about $D = 450$ for M&S and at about $D = 900$ for the other two configurations.

Next, we ran the scalability experiment again, using nonzero delay parameters. We chose $D = 450$ (where Figure 8(b) suggests that M&S is no longer affected by contention, but the other two configurations still are) and $D = 900$ (where all configurations appear not to be affected by contention). The results for $D = 450$ are shown in Figure 8(c). These results are more in line with simple intuition about parallelism: performance improves with each additional processor. However, observe that the two configurations that incorporate the Pass The Buck algorithm start to perform slightly worse as the number of threads approaches 16. This is consistent with our choice of D for this experiment: these configurations are still affected by changes in the contention level at this point, whereas M&S is not.

We also conducted a similar sequence of experiments on a 64-processor machine. (On the 64-processor machine, each trial performed 8,000,000 operations in total across all participating threads.) The results were qualitatively similar on the two machines. However, the 64-processor counterpart to Figure 8(b) showed the knee of the curve for the M&S configuration at about $D = 4000$ for M&S and at about $D = 5500$ for the other two configurations. This is explained by the fact that the ratio of memory access time to cycle time is larger on the larger machine. We therefore conducted the scalability experiments for $D = 4000$ and $D = 6000$ on this machine. The results for $D = 4000$ looked qualitatively similar to the 16-processor results for $D = 450$. The results for $D = 6000$ are shown in Figure 8(d). (The counterpart experiment on the 16-processor machine with $D = 900$ yielded qualitatively similar results, with the curve bottoming out at about 6000 ms.) Here we see that when there is little contention, the results of the three configurations are almost indistinguishable.

Based on the above results, we believe that the penalty for using our dynamic-sized version of the M&S algorithm will be negligible in practice: contention for

[11]Anomalous behaviour exhibited by our initial experiments turned out to be caused by contention on a lock in the random number generator; therefore, we computed a large array of random bits in advance, and used this for choosing between enqueuing and dequeuing and for choosing the number of delay loop iterations.

a particular queue should usually be low because typical applications will do other work between queue operations.

6. THE PASS THE BUCK ALGORITHM

In this section, we describe one ROP solution, the *Pass The Buck* (PTB) algorithm in detail; we also describe an alternative algorithm that is simpler but provides slightly weaker progress guarantees, at least in principle. To avoid confusion and to emphasize the abstract nature of the ROP problem, we describe our solution in terms of the *values* it manages, rather than referring to the pointers that these values represent when the solution is used as described in the previous sections.

Our primary goal when designing PTB was to minimize the performance penalty to the application when no values are being liberated. That is, the PostGuard operation should be implemented as efficiently as possible, perhaps at the cost of a more expensive Liberate operation. Such solutions are desirable for at least two reasons. First, PostGuard is necessarily invoked by the application, so its performance always impacts application performance. On the other hand, Liberate work can be done by a spare processor, or by a background thread, so that it does not directly impact application performance. Second, solutions that optimize PostGuard performance are desirable for scenarios in which values are liberated infrequently, but we must retain the ability to liberate them. An example is the implementation of a dynamic-sized data structure that uses a memory pool to avoid allocating and freeing objects under "normal" circumstances but can free elements of the memory pool when it grows too large. In this case, no liberating is necessary while the size of the data structure is relatively stable. With this goal in mind, we describe our Pass The Buck algorithm below.

The Pass The Buck algorithm is shown in Figure 9. The GUARDS array is used to allocate guards to threads. Here we assume a bound MG on the number of guards simultaneously employed; it is straightforward to remove this restriction [Herlihy et al. 2003]. The POST array consists of one location per guard, which holds the value the guard is currently assigned to guard if one exists, and **null** otherwise. The HNDOFF array is used by Liberate to "hand off" responsibility for a value to a later Liberate operation if the value has been trapped by a guard.

The HireGuard and FireGuard procedures essentially implement long-lived renaming; we use the renaming algorithm presented in Anderson and Moir [1997]. Specifically, for each guard g, we maintain an entry GUARDS[g], which is initially *false*. Thread p hires guard g by atomically changing GUARDS[g] from *false* (unemployed) to *true* (employed); p attempts this with each guard in turn until it succeeds (lines 2 and 3). The FireGuard procedure simply sets the guard back to *false* (line 7). The HireGuard procedure also maintains the shared variable MAXG, which is used by the Liberate procedure to determine how many guards to consider. Liberate must consider every guard for which a HireGuard operation has completed. Therefore, it suffices to have each HireGuard operation ensure that MAXG is at least the index of the guard returned. This is achieved with the simple loop at lines 4 and 5.

```
struct { value val; int ver } HO_t
    // HO_t fits into CAS-able location
constant MG: max. number of guards
shared variable
  GUARDS: array[0..MG-1] of bool
                                  init false;
  MAXG: int init 0;
  POST: array[0..MG-1] of value
                                  init null;
  HNDOFF: array[0..MG-1] of HO_t
                                  init ⟨null, 0⟩;

int HireGuard() {
1  int i = 0, max;
2  while (!CAS(&GUARDS[i],false,true))
3    i++;
4  while ((max = MAXG) < i)
5    CAS(&MAXG,max,i);
6  return i;
}

void FireGuard(int i) {
7  GUARDS[i] = false;
8  return
}

void PostGuard(int i, value v) {
9  POST[i] = v;
10 return
}

value_set Liberate(value_set vs) {
11 int i = 0;
12 while (i <= MAXG) {
13   int attempts = 0;
14   HO_t h = HNDOFF[i];
15   value v = POST[i];
16   if (v != null && vs→search(v)) {
17     while (true) {
18       if (CAS(&HNDOFF[i], h,
                              ⟨v, h.ver+1⟩)) {
19         vs→delete(v);
20         if (h.val != null) vs→insert(h.val);
21         break;
         }
22       attempts++;
23       if (attempts == 3) break;
24       h = HNDOFF[i];
25       if (attempts == 2 && h.val != null)
           break;
26       if (v != POST[i]) break;
       }
27   } else {
28     if (h.val != null && h.val != v)
29       if (CAS(&HNDOFF[i], h, ⟨null,
                                  h.ver+1⟩))
30         vs→insert(h.val);
     }
31   i++;
   }
32 return vs;
}
```

Fig. 9. Code for Pass The Buck.

PostGuard is implemented as a single store of the value to be guarded in the specified guard's POST entry (line 9), in accordance with our goal of making PostGuard as efficient as possible.

The most interesting part of PTB lies in the Liberate procedure. Recall that Liberate should return a set of values that have been passed to Liberate and have not since been returned by Liberate, subject to the constraint that Liberate cannot return a value that has been continuously guarded by the same guard since before the value was most recently passed to Liberate (i.e., Liberate must not return trapped values).

Liberate is passed a set of values, and it adds values to and removes values from its value set as described below before returning (i.e., liberating) the remaining values in the set. Because we want the Liberate operation to be wait-free, if some guard g is guarding a value v in the value set of some thread p executing Liberate, then p must either determine that g is not trapping v or remove v from p's value set before returning that set. To avoid losing values, any value that p removes from its set must be stored somewhere so that, when the value is no longer trapped, another Liberate operation may pick it up and return it. The interesting details of PTB concern how threads determine that a value is not trapped, and how they store values while keeping space overhead

for stored values low. Below we explain the Liberate procedure in more detail, paying particular attention to these issues.

6.1 The Liberate Procedure in Detail

The loop at lines 12 through 31 iterates over all guards ever hired. For each guard g, if p cannot determine for some value v in its set that v is not trapped by g, then p attempts to "hand v off to g". If p succeeds in doing so (line 18), it removes v from its set (line 19) and proceeds to the next guard (lines 21 and 31). If p repeatedly attempts and fails to hand v off to g, then, as we explain below, v cannot be trapped by g, so p can move on to the next guard (lines 23 and 25). Also, as explained in more detail below, p might simultaneously pick up a value previously handed off to g by another Liberate operation, in which case this value can be shown not to be trapped by g, so p adds this value to its set (line 20). When p has examined all guards (see line 12), it can safely return any values remaining in its set (line 32).

We describe the processing of each guard in more detail below. First, however, we present a central property of the correctness proof of this algorithm, which will aid the presentation that follows; this lemma is quite easy to see from the code and the high-level description given thus far; it is formalized in Invariant 12 of the correctness proof given in the appendix.

SINGLE LOCATION LEMMA. *For each value v that has been passed to some invocation of* Liberate *and not subsequently returned by any invocation of* Liberate, *either v is handed off to exactly one guard, or v is in the value set of exactly one* Liberate *operation (but not both). Also, any value handed off to a guard or in the value set of any* Liberate *operation has been passed to* Liberate *and not subsequently returned by* Liberate.

The processing of each guard g proceeds as follows: At lines 15 and 16, p determines whether the value (if any) currently guarded by g—call it v—is in its set. If so, p executes the loop at lines 17 through 26 in order to either determine that v is not trapped, or to remove v from its set. In order to avoid losing v in the latter case, p "hands v off to g" by storing v in HNDOFF[g]. In addition to the value, an entry in the HNDOFF array contains a version number, which, for reasons that will become clear later, is incremented with each modification of the entry.[12] Because at most one value may be trapped by guard g at any time, a single location HNDOFF[g] for each guard g is sufficient. To see why, observe that if p needs to hand v off because it is guarded, then the value (if any)—call it w—previously stored in HNDOFF[g] is no longer guarded, so p can pick w up and add it to its set. (Because p attempts to hand off v only if v is in p's set, the Single Location Lemma implies that $v \neq w$.) The explanation above gives the basic idea of our algorithm, but it is oversimplified; there are various subtle race conditions that must be avoided. Below, we explain in more detail how the algorithm deals with these race conditions.

[12]As is usual with version numbers, we assume that enough bits are used for the version numbers that "wraparound" is impossible in practice; see Moir [1997] for discussion and justification.

To hand v off to g, p uses a CAS operation to attempt to replace the value previously stored in HNDOFF[g] with v (line 18); this ensures that, upon success, p knows which value it replaced, so it can add that value to its set (line 20). We explain later why it is safe to do so. If the CAS fails due to a concurrent Liberate operation, then p rereads HNDOFF[g] (line 24) and loops around to retry the handoff. There are various conditions under which we break out of this loop and move on to the next guard. In particular, the loop completes after at most three CAS attempts; see lines 13, 22, and 23. (Because the CAS instruction is relatively expensive, it is worth noting that when our algorithm is used as described in the previous sections, it uses CAS only when some thread is still using a pointer that has already been removed from the data structure—a transient condition that we expect to occur rarely.) It follows that our algorithm is wait-free. We explain later why it is safe to stop trying to hand v off in each of these cases.

We first consider the case in which p exits the loop due to a successful CAS at line 18. In this case, as described earlier, p removes v from its set (line 19), adds the previous value in HNDOFF[g] to its set (line 20), and moves on to the next guard (lines 21 and 31). An important part of understanding our algorithm is to understand why it is safe to take the previous value—call it w—of HNDOFF[g] to the next guard. The reason is that we read POST[g] (line 15 or 26) between reading HNDOFF[g] (line 14 or 24) and attempting the CAS at line 18. Because each modification to HNDOFF[g] increments its version number, it follows that w was in HNDOFF[g] when p read POST[g]. Also, recall that $w \neq v$ in this case. Therefore, when p read POST[g], w was not guarded by g. Furthermore, because w remained in HNDOFF[g] from that moment until the CAS, w cannot become trapped in this interval. To see why, recall that a value can become trapped only if it has not been passed to Liberate since it was last allocated. However, each value in the HNDOFF array (including w) has been passed to some invocation of Liberate and not yet returned by any invocation of Liberate (and has therefore not been freed and reallocated since being passed to Liberate).

It remains to consider how p can break out of the loop *without* performing a successful CAS. In each case, p can infer that v is not trapped by g, so it can give up on its attempt to hand v off. If p breaks out of the loop at line 26, then v is not trapped by g at that moment simply because it is not even guarded by g. The other two cases (lines 23 and 25) occur only after a certain number of times around the loop, implying a certain number of failed CAS operations.

To see why we can infer that v is not trapped in each of these two cases, consider the timing diagram in Figure 10. (For the rest of this section, we use the notation v_p to indicate the value of thread p's local variable v in order to distinguish between the local variables of different threads.) In this diagram, we construct an execution in which p fails its CAS three times. The bottom line represents thread p: at (A), p reads HNDOFF[g] for the first time (line 14); at (B), p's CAS fails; at (C), p rereads HNDOFF[g] at line 24; and so on for (D), (E), and (F). Because p's CAS at (B) fails, some other thread q_0 executing Liberate performed a successful CAS after (A) and before (B); choose one and call it (G). (The arrows between (A) and (G) and between (G) and (B) indicate that we know (G) comes after (A) and before (B).) Similarly, some thread q_1

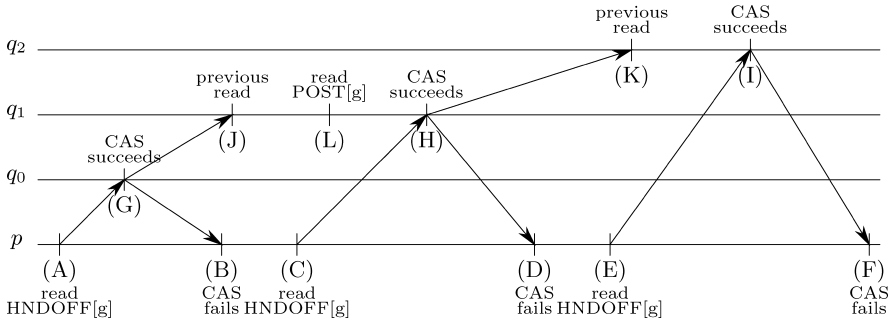

Fig. 10. Timing diagram illustrating interesting cases for Pass The Buck.

executes a successful CAS on HNDOFF[g] after (C) and before (D)—call it (H); and some thread q_2 executes a successful CAS on HNDOFF[g] after (E) and before (F)—call it (I). (Threads q_0 through q_2 might not be distinct, but there is no loss of generality in treating them as if they were.)

Consider the CAS at (H). Because every successful CAS of HNDOFF[g] increments its version number field, q_1's previous read of HNDOFF[g] (at line 14 or line 24)—call it (J)—must come after (G). Similarly, q_2's previous read of HNDOFF[g] before (I)—call it (K)—must come after (H).

We consider two cases. First, suppose (H) is an execution of line 18 by q_1. In this case, v_{q_1} is in q_1's value set, and between (J) and (H), q_1 read POST[g] $= v_{q_1}$, either at line 15 or at line 26; call this read (L). By the Single Location Lemma, because v_p is in p's set, $v_p \neq v_{q_1}$, so the read at (L) implies that v_p was not guarded by g at (L). Therefore, v_p was not trapped by g at (L), which implies that it is safe for p to break out of the loop after (D) in this case (observe that attempts$_p = 2$ in this case).

For the second case, suppose (H) is an execution of line 29 by thread q_1. In this case, because q_1 is storing **null** instead of a value in its own set, the above argument does not work. However, because p breaks out of the loop at line 25 only if it reads a non-**null** value from HNDOFF[g] at line 24, it follows that if p does so, then *some* successful CAS stored a non-**null** value to HNDOFF[g] at or after (H), and in this case the above argument can be applied to that CAS to show that v_p was not trapped. If p reads **null** at line 24 after (D), then it continues through its next loop iteration.

In this case, there is a successful CAS (I) that comes after (H). Because (H) stored **null** in the current case, no subsequent execution of line 29 by any thread will succeed before the next successful execution of the CAS in line 18 by some thread. To see why, observe that the CAS at line 29 never succeeds while HNDOFF[g] contains **null** (see line 28). Therefore, for (I) to exist, there is a successful execution of the CAS at line 18 by some thread after (H) and at or before (I). Using this CAS, we can apply the same argument as before to conclude that v_p was not trapped. This argument is formalized in an appendix. It is easy to see that PTB is wait-free.

As described so far, p picks up a value from HNDOFF[g] only if its value set contains a value that is guarded by guard g. Therefore, without some additional mechanism, a value stored in HNDOFF[g] might never be picked up from there.

To avoid this problem, even if p does not need to remove a value from its set, it still picks up the previously handed off value (if any) by replacing it with **null** (see lines 28 through 30). We know it is safe to pick up this value by the argument above that explains why it is safe to pick up the value stored in HNDOFF[g] in line 18. Thus, if a value v is handed off to guard g, then the first Liberate operation to begin processing guard g after v is not trapped by g will ensure that v is picked up and taken to the next guard (or returned from Liberate if g is the last guard), either by that Liberate operation or some concurrent Liberate operation.

6.2 Progress Guarantees

As noted above, the PTB algorithm is wait-free: The FireGuard and PostGuard procedures are obviously wait-free; they consist of straightline code. The HireGuard procedure has two loops, both of which terminate after a finite number of iterations. The first loop increments i with every iteration and terminates before i exceeds the total number of guards (see Anderson and Moir [1997] or Herlihy et al. [2003]). The second loop terminates if MAXG $\geq i$; Otherwise, it tries to set MAXG to i; if it does not succeed, some other thread must have increased the value of MAXG, and because MAXG never decreases, this loop has at most i iterations. Finally, the Liberate procedure is wait-free, as argued above, because it executes the loop at lines 17 to 26 at most three times per iteration of the outer loop, which it executes exactly once per guard.

To see that PTB guarantees the Value Progress property stated in Section A.1, suppose a value v is passed to Liberate by a thread p at time t, and that no thread fails after t. Suppose further that v is not guarded at some time t' and that v remains unguarded after time t' until v is liberated. Finally, suppose that some thread invokes Liberate after time t' (note that if $t' < t$, this may be the same as the invocation of Liberate by thread p).

We consider two cases. If $t \geq t'$, p will never find v guarded and so it will never try to hand v off, so v will be liberated when p returns from Liberate. (Note that it does so because Liberate is wait-free and no thread fails after t.)

Otherwise, $t < t'$. By the Single Location Lemma, if v is not liberated before t' (i.e., it is still escaping at time t'), then at time t', v is either in the value set of some thread q executing Liberate or in HNDOFF[i] for some i. In the first case, v will be liberated when q returns from Liberate. (The Single Location Lemma also implies that v was not in q's value set immediately before time t because it was not escaping at that time. Therefore, q added v to its value set at or after time t. Because no thread fails at or after time t, this implies that q eventually returns from Liberate.) In the second case, the next thread that checks guard i will pick up and liberate v. (Note that thread q implies the existence of such a thread.)

The Value Progress property stated in the appendix may seem somewhat weak, because of its premise that no threads fail after time t. However, it is important to note that, because threads cannot determine that there will be thread failures in the future, they must behave exactly the same as if there were none. Thus, the Value Progress property actually implies that any value that is unguarded and is passed to Liberate is eventually liberated provided

```
value_set Liberate(value_set vs) {
11 int i = 0;
12 while (i <= MAXG) {
13   value v = POST[i];
14   if (v != null && vs→search(v)) {
15     while  (v != null && v == POST[i]) {
16       HO_t h = HNDOFF[i];
17       if (CAS(&HNDOFF[i], h, v)) {
18         vs→delete(v);
19         if (h.val != null) vs→insert(h.val);
20         v = HNDOFF[i];
         }
       }
21   } else {
22     HO_t h = HNDOFF[i];
23     if (h.val != null && h.val != v)
24       if (CAS(&HNDOFF[i], h, null))
25         vs→insert(h.val);
     }
26   i++;
   }
27 return vs;
}
```

Fig. 11. Lock-free Liberate using pointer-sized CAS. (Other operations are the same as in Figure 9.)

no threads fail *while it is being liberated*, even if threads do fail in the future. The Value Progress property is stated the way it is because we express it as a requirement of *any* implementation of our API.

For the PTB algorithm, we can state more precisely the conditions under which a value that is unguarded and is passed to Liberate can fail to be liberated: this occurs *only* if some thread fails while it is executing Liberate and has v in its value set, or if it is in HNDOFF[i] for some i and no thread ever checks guard i again. Because the size of the value set of each Liberate operation is bounded, and the number of guards is bounded, this implies that only a bounded number of values are ever passed to Liberate and not subsequently liberated, assuming only a bounded number of threads fail. This is important when using PTB to reclaim memory because it implies that a thread failure cannot cause an unbounded memory leak. (See our discussion of Treiber's algorithm [Treiber 1986] in Section 8.)

6.3 An Alternative Solution

A potential disadvantage of the PTB algorithm is that it relies on the ability to atomically manipulate a pointer and a version number using CAS. All current 32-bit shared-memory multiprocessors we are aware of provide 64-bit synchronization instructions (such as CAS) that support such techniques, and, as pointed out by Michael [2002b], 128-bit synchronization instructions for 64-bit architectures are likely to follow. Nonetheless, in the meantime, the PTB algorithm presented above is not applicable in 64-bit architectures, so it is desirable to modify the algorithm so that it is.

Figure 11 shows a simpler variant of the Liberate operation, which uses CAS only on pointer-sized variables. This Liberate operation can be substituted for the one shown in Figure 9, and makes the same correctness guarantees.

In the Liberate operation in Figure 9, we can move on to the next guard after at most three failures to hand off a value; as we reason above, the value cannot be trapped by that guard in this case. However, this reasoning depended on the version numbers, to show for example that event (J) occurs after event (G) in Figure 10, which in turn implies that event (L) occurs within this execution of Liberate, so we know that a value other than the one we are trying to hand off was guarded by this guard. Without version numbers, this may not be true, so we cannot reach the same conclusion. Similarly, we cannot be sure that a value that we replace in POST[i] is not now trapped by guard i. Therefore, if our CAS fails we must retry, and even if it succeeds, we must check to see that the replaced value is not now guarded (and therefore possibly trapped).

The advantages of the Liberate operation in Figure 11 are that it is simpler and somewhat more widely applicable. The only disadvantage is that is only lock-free, not wait-free. This is because the loop at lines 15–20 can in principle loop repeatedly. We argue that this is extremely unlikely in practice, so this disadvantage is just theoretical.

To see why, suppose that the CAS at line 17 repeatedly succeeds, but the loop does not terminate. This requires the value in POST[i] to alternate between two different values in perfect concert with the Liberate operation checking it at line 15. If a process guards one of these values during this time, then either it is guarding a value that has already been passed to liberate, which implies that it was slow to post the guard, or the value is liberated, recycled, and passed to liberate again, and repeatedly stored in the HNDOFF[i], again in perfect concert with the Liberate operation under consideration. It is inconceivable that this would happen repeatedly.

Similarly, if the CAS at line 17 repeatedly fails, then other values are repeatedly stored in HNDOFF[i], and the threads that store them observe those values in POST[i]. Some of those threads may have observed those values a long time ago, but eventually this is not the case, and we again have a situation in which the value in POST[i] "conveniently" alternates between the value the Liberate operation is trying to hand off and a different value. As in the previous case, this eventually requires repeated coincidences in which a value is liberated, recycled, and reintroduced.

Because this Liberate is not wait-free, it also satisfies a slightly weaker value progress property than the one described above. However, it still guarantees that the delay or failure of a thread can prevent only a bounded number of values from being liberated. Concretely, a value can be prevented from being liberated only by a failed thread, or by an unlikely scenario such as the ones described above, and in any case, only a bounded number of escaping values are not liberated at any point in time.

7. SINGLE-WORD LOCK-FREE REFERENCE COUNTING (SLFRC)

In Section 4, we showed how to use ROP to make Michael and Scott's lock-free queue algorithm dynamic-sized. Although few changes to the algorithm were required, determining that the changes preserved correctness required careful reasoning and a detailed understanding of the original algorithm. In this

section, we present *single-word lock-free reference counting* (SLFRC), a technique that allows us to transform many lock-free data structure implementations that assume garbage collection (i.e., they never explicitly free memory) into dynamic-sized data structures in a straightforward manner. This technique enables a general methodology for designing dynamic-sized data structures, in which we first design the data structure as if GC were available, and then we use SFLRC to make the implementation independent of GC. Because SLFRC is based on reference counts, it shares the disadvantages of all reference counting techniques, including space and time overheads for maintaining reference counts and the need to deal with cyclic garbage.

SLFRC is closely related to *lock-free reference counting* (LFRC) [Detlefs et al. 2001]. Rather than present all the details, we begin with an overview of LFRC and present details only of the differences between SLFRC and LFRC. Therefore, for a complete understanding of SLFRC, the reader should read [Detlefs et al. 2001] first.

7.1 Overview of LFRC

The LFRC methodology provides a set of operations for manipulating pointers (LFRCLoad, LFRCStore, LFRCCAS, LFRCCopy, LFRCDestroy, etc.). These operations are used to maintain reference counts on objects, so that they can be freed when no more references remain. The reference counts are not guaranteed to be always perfectly accurate; they may be too high because reference counts are sometimes incremented in anticipation of the future creation of a new reference. However, such creations might never occur, for example, because of a failed CAS. In this case, the LFRC operations decrement the reference count to compensate.

Most of the LFRC pointer operations act on objects to which the invoking thread knows that a pointer exists and will not disappear before the end of the operation. For example, the LFRCCopy operation makes a copy of a pointer, and therefore increments the reference count of the object to which it points. In this case, the reference count can safely be accessed because we know that the first copy of the pointer has been included already in the reference count, and this copy will not be destroyed before the LFRCCopy operation completes.

The LFRCLoad operation, which loads a pointer from a shared variable into a private variable, is more interesting. Because this operation creates a new reference to the object to which the pointer points, we need to increment the reference count of this object. The problem is that the object might be freed after a thread p reads a pointer to it, and before p can increment its reference count. The LFRC solution to this problem is to use DCAS to atomically confirm the existence of a pointer to the object while incrementing the object's reference count. This way, if the object had previously been freed, then the DCAS would fail to confirm the existence of a pointer to it, and would therefore not modify the reference count.

7.2 From LFRC to SLFRC

The SLFRC methodology described here overcomes two shortcomings of LFRC: it does not depend on DCAS, and it never allows threads to access freed objects.

```
void SLFRCDestroy(Obj *ptr) {
1  if (ptr != null && add_to_rc(ptr, −1) == 1) {
2      // Recursively call SLFRCDestroy with each
       // pointer in the object pointed to by ptr.
3      for each v ∈ Liberate({ptr}) do
4          free(v);
   }
}

long add_to_rc(Obj *ptr, int v) {
5  long oldrc;
6  while (true) {
7      oldrc = ptr→rc;
8      if (CAS(&ptr→rc, oldrc, oldrc+v))
9          return oldrc;
   }
}

void SLFRCLoad(Obj **A, Obj **dest) {
10 Obj *a, *olddest = *dest;
11 long r;
12 while (true) {
13     a = *A;
14     if (a == null) break;
15     PostGuard(g_p, a);
16     if (a == *A)
17         while ((r = a→rc) > 0)
18             if (CAS(&a→rc, r, r+1))
19                 goto 20;
   }
20 if (a != null)
21     PostGuard(g_p, null);
22 *dest = a;
23 SLFRCDestroy(olddest);
}
```

Fig. 12. Code for SLFRCDestroy and SLFRCLoad. Code for add_to_rc is repeated from Detlefs et al. [2001]. Code is shown for thread p; g_p is a guard owned by p.

SLFRC provides the same functionality as LFRC does except that it does not support a LFRCDCAS operation. The implementation of each SLFRC operation, except SLFRCLoad and SLFRCDestroy, is identical to its LFRC counterpart. The implementations of these two operations, shown in Figure 12, are discussed below.

SLFRC avoids the need for DCAS by using ROP to coordinate access to an object's reference count. To accommodate this change, the SLFRCDestroy operation must be modified slightly from the LFRCDestroy operation used in Detlefs et al. [2001]. The LFRCDestroy operation decrements the reference count of the object O pointed to by its argument and, if the reference count becomes zero as a result, recursively destroys each of the pointers in O, and finally frees O. The SLFRC version of this operation must arrange for pointers to be passed to Liberate, rather than freeing them directly. When a pointer has been returned by Liberate, it can be freed. One way to achieve this, which we adopt in Figure 12, is to follow the simple technique used in Section 2: SLFRCDestroy invokes Liberate directly, passing as a parameter the singleton set containing the pointer to be freed, and then frees all pointers in the set returned by Liberate. Various alternatives are possible. For example, a thread might "buffer" pointers to be passed together to Liberate later, either by that thread, or by some other thread whose sole purpose is executing Liberate operations. The latter approach allows us greater flexibility in scheduling when and where this work is done, which is useful for avoiding inconvenient pauses to application code.

We now describe the SLFRCLoad operation and explain how it overcomes the need for the DCAS operation required by LFRCLoad. In the loop at lines 12 to 19, SLFRCLoad attempts to load a pointer value from the location specified by the argument `A` (line 13), and to increment the reference count of the object to which it points (line 18). To ensure that the object is not freed before the reference count is accessed, we employ ROP. Specifically, at line 15, we post a guard on the value read previously. This is not sufficient to prevent the object

from being freed before its reference count is accessed: we must ensure that the object being guarded has a nonzero reference count *after* the guard has been posted. This is achieved by rereading the location at line 16: if the value no longer exists in this location, then SLFRCLoad retries. (This retrying does not compromise lock-freedom because some other thread successfully completes a pointer store for each time around the loop.) If the pointer is still (or again) in the location, then the object has not been passed to Liberate since it was last allocated (because objects are passed to Liberate only after their reference counts become zero, which happens only when no pointers to them remain).

If the value read at line 13 is **null**, then there is no reference count to update, so there is no need to post a guard (see line 14). Otherwise, because of the guarantees of ROP, it is safe to access the reference count of the object pointed to by the loaded value for as long as the guard remains posted. This is achieved by a simple lock-free loop (lines 17 to 19) in which we repeatedly read the reference count and use CAS to attempt to increment it. Upon success, it simply remains to remove the guard, if any (lines 20 and 21), arrange for the return of the pointer read (line 22), and destroy the previous contents of the destination variable (see lines 10 and 23).

Observe that we increment the reference count of an object only if it is nonzero (line 17); if the reference count becomes zero, we retry the outer loop to get a new pointer. The reason for this is that if the reference count is zero, some thread has already begun recursively destroying outgoing pointers from the object in SLFRCDestroy, so it is too late to "resurrect" the object.

It is interesting to note that we could easily apply SLFRC to Michael and Scott's queue algorithm [Michael and Scott 1998],[13] to achieve a dynamic-sized version of that algorithm. However, because of the overhead associated with reference counting, the resulting implementation would likely perform significantly worse than the implementations we presented in Section 4, which were achieved by applying our mechanisms directly, rather than through SLFRC.

8. RELATED WORK

In this section, we discuss other work on nonblocking implementations of dynamic-sized data structures that do not depend on garbage collection.[14]

One common approach is to augment values in objects with version numbers or tags, and to access such values only through the use of CAS, such that if a CAS executes on an object after it has been deallocated, the value of the version number or tag will ensure that the CAS fails [Michael and Scott 1998; Moir 1997; Treiber 1986]. In this case, the version number or tag value must be carried with the object through deallocation and reallocation, which is usually achieved through the use of explicit memory pools. As explained in Section 4,

[13]Because Michael and Scott's algorithm stores pointers atomically with version numbers, some minor and straightforward changes to (S)LFRC would be required for extracting actual pointer values from stored values.

[14]These approaches are all forms of *type stable memory* (TSM), defined by Greenwald [1999] as follows: "TSM [provides] a guarantee that an object O of type T remains type T as long as a pointer to O exists."

this approach results in implementations that cannot free memory that is no longer required.

Valois [1995] proposed another approach, in which the memory allocator maintains reference counts for objects in order to determine when they can be freed. Valois's approach allows the reference count of an object to be accessed even after the object has been released to the memory allocator. This behaviour restricts what the memory allocator can do with released objects. For example, the released objects cannot be coalesced. Thus, the disadvantages of maintaining explicit memory pools are shared by Valois's approach. Furthermore, application designers sometimes need to switch between different memory allocation libraries for performance or other reasons. Valois's approach requires the memory allocator to support certain nonstandard functionality, and therefore precludes this possibility.

Interestingly, the previous work that comes closest to meeting the goal of this article—to provide support for explicit nonblocking memory management that depends only on standard hardware and system support—predates the work discussed above by almost a decade. Treiber [1986] proposes a technique called *obligation passing*.[15] The instance of this technique for which Treiber presents specific details is in the implementation of a lock-free linked list supporting search, insert, and delete operations. This implementation allows freed nodes to be returned to the memory allocator through standard interfaces and without requiring any special functionality of the memory allocator. However, it employs a "use counter" such that memory is reclaimed only by the "last" thread to access the linked list in any period. As a result, this implementation can be prevented from ever recovering any memory by a failed thread (which defeats one of the main purposes of using lock-free implementations). Another disadvantage of this implementation is that the obligation passing code is bundled together with the linked-list maintenance code (all of which is presented in assembler code). Because it is not clear what aspects of the linked-list code the obligation passing code depends on, it is difficult to apply this technique to other situations.

Michael [2002a, 2004] has independently and concurrently developed an approach to nonblocking memory management that is basically the same idea as ours. Technically, Michael's algorithm is not a solution to the Repeat Offenders Problem as we define it (see Appendix A). Nevertheless, Michael's algorithm can easily be adapted to conform to our interface, and to provide the safety properties we require (liveness is another issue). Despite the fundamental similarities between our work and Michael's we have suggested some different design choices; we discuss some of these choices and compare our approach to Michael's in more detail below.

Michael's equivalent of the Liberate operation is called *Scan*. His algorithm buffers to-be-freed pointers on a per-thread basis in order to control how many are passed to *Scan* at a time. As mentioned in Section 5 some local buffering of to-be-freed pointers is desirable to reduce contention and synchronization costs.

[15]We named our technique "Pass The Buck" before we were aware of Treiber's work on obligation passing, which indicates the similarity of the underlying philosophies of our approaches, although the details are quite different.

Michael's approach is to have application threads process the buffers when they become full, and to use large buffers in order to achieve an attractive amortized bound on the time spent per pointer freed.

Using Michael's buffering approach, there are typically $O(GP)$ pointers that could be freed but are not, where P is the number of participating threads, and G is the number of "hazard pointers"—the equivalent of our guards. Clearly, there is a tradeoff between the direct benefits gained from such buffering and the space cost (and potential indirect performance effects thereof) for the buffers. There is also a tradeoff between the cost of processing buffered pointers in application threads, and the cost of putting them into a shared memory pool so that they can be reused or recycled in the background or on a different processor.

Our opinion is that, as the number of threads increases, the cost of executing *Scan* inline in application threads will become unacceptable for many applications, despite the amortized cost, and it will be preferable to allow the processing of to-be-freed pointers to be done in the background, or concurrently on another processor, rather than burdening application threads with performing this work on their critical paths. Similar lessons have been learned in the context of garbage collection [Bacon et al. 2001; Printezis and Detlefs 2000].

According to Michael [2004], it has been demonstrated experimentally that his approach outperforms ours. However, he presents no data to support this claim, and does not describe any implementation of our approach or any experiments or metrics by which he has compared the approaches. We are not aware of any head-to-head performance comparison of the two approaches. However, we do not believe that such a comparison is useful or meaningful. The cost that is fundamentally required by both approaches to be executed inline by application threads is identical: posting a guard (or equivalently, setting a hazard pointer) is a single store, followed by a memory barrier operation if required in the target system. The best combination of per-thread buffering, recycling through memory pools, and processing through Liberate (or equivalently *Scan*) is highly dependent on the particular system and the requirements and behaviour of the particular application with respect to latency, memory usage, etc. While we have made different choices in approaching the tradeoffs in our presentation than Michael has in his, both approaches are equally amenable to a variety of design choices to suit a particular situation.

Michael's requirements for *Scan* are weaker than ours for Liberate. In particular, *Scan* may return some pointers to the buffer if they cannot yet be freed, while our algorithm "hands off" such values. A possible advantage of these weaker requirements is that Michael's algorithm is very simple and needs only read and write primitives, while ours requires CAS. However, note that CAS (or an equivalent) is available in all modern shared-memory multiprocessors, and is needed anyway for almost all practical lock-free data structures. Also we do not expect our use of the relatively expensive CAS instruction in Liberate to be detrimental to performance, because it uses CAS only under circumstances we expect to be rare.

A disadvantage of Michael's weaker requirement is that a thread may have to wait for other threads in order to empty its buffer completely. If a thread

that wishes to terminate does so while pointers remain in its buffer, then those pointers will never be freed, and even a single unfreed pointer might tie up a large amount of memory. In contrast, our algorithm allows a thread to terminate without waiting for other threads and without leaking memory. (We characterize the liveness property that facilitates this functionality more formally as the *Value Progress* property stated in the appendix.) It is straightforward to combine aspects of both algorithms, periodically interrupting Michael's buffering strategy with our more aggressive approach. For example, such a hybrid algorithm could be configured to achieve the same amortized bound that Michael favors while also satisfying the Value Progress property.

Michael [2002a] has shown how to apply his technique to dynamic-sized implementations of different data structures such as queues, double-ended queues, list-based sets, and hash tables. The interfaces and safety properties of our approaches are similar enough that one could also use any ROP solution for these examples. An advantage of using our PTB implementation of ROP is that it satisfies the Value Progress condition.

9. CONCLUDING REMARKS

We have presented a set of mechanisms designed to support memory management for nonblocking shared data structures. We have explained how these mechanisms can be implemented and used, and we have provided a formal proof of their correctness. We have used our mechanisms to achieve the first dynamic-sized nonblocking shared data structures that cannot be prevented from future memory reclamation by thread failures, and depend only on widely available hardware support for synchronization. Our performance experiments show that the overhead introduced by applying our techniques to the lock-free FIFO queue of Michael and Scott is always modest, and is negligible under low contention.

When designing the interface to our mechanisms, we paid particular attention to allowing much of the work to be scheduled separately from application work. Future research includes experimenting with ways of doing this scheduling in various application and system configurations, as well as applying our techniques to the design of other data structures.

The ideas in this paper came directly from insights gained and questions raised in our work on LFRC [Detlefs et al. 2001]. This demonstrates the value of research that assumes stronger synchronization primitives than are currently widely supported.

APPENDIXES

A. THE REPEAT OFFENDER PROBLEM

In this appendix, we define the *Repeat Offender Problem* (ROP), which captures the abstract properties of the support mechanisms for nonblocking memory management described in this paper. ROP is defined with respect to a set of *values*, a set of *application clients*, and a set of *guards*. Each value may be free, injail, or escaping; initially, all values are free. An application-dependent external Arrest action can cause a free value to become injail at any time. A client

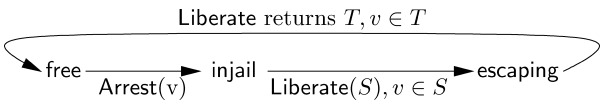

Fig. 13. Transition diagram for value v.

can help injail values to begin escaping, which causes them to become escaping. Values that are escaping can finish escaping and become free again.

Clients can use values, but must never use a value that is free. A client can attempt to prevent a value v from escaping while it is being used by "posting a guard" on v. However, if the guard is posted too late, it may fail to prevent v from escaping. Thus, to safely use v, a client must ensure that v is injail at some time *after* it posted a guard on v. Clients can hire and fire guards dynamically, according to their need.

ROP solutions can be used by threads (clients) to avoid dereferencing (using) a pointer (value) to an object that has been freed. In this context, an injail pointer is one that has been allocated (arrested) since it was last freed, and can therefore be used.

ROP solutions provide the following procedures: A client *hires* a guard by invoking HireGuard(), and it *fires* a guard g that it employs by invoking FireGuard(g). An ROP solution ensures that a guard is never simultaneously employed by multiple clients. A client *posts* a guard g on a value v by invoking PostGuard(g, v); this removes the guard from any value it previously guarded. (A special **null** value is used to remove the guard from the previously guarded value without posting the guard on a new value). A client may not post (or remove) a guard that it does not currently employ. A client helps a set of values S to *begin escaping* by invoking Liberate(S); the application must ensure that each value in S is injail before this call, and the call causes each value to become escaping. The Liberate procedure returns a (possibly different) set of escaping values causing them to be *liberated*; each of these values becomes free on the return of this procedure. These transitions are summarized in Figure 13. An ROP solution does not implement the functionality of the Arrest action—this is application-specific—but ROP must be aware of arrests in order to know when a free value becomes injail.

If a guard g is posted on a value v, and v is injail at some time t after g is posted on v and before g is subsequently removed or reposted, then we say that g *traps* v from time t until g is removed or reposted. The operational specification of the main correctness condition for ROP is that it does not allow a value to escape (i.e., become free) while it is trapped.

A precise formulation of ROP is given by the I/O automaton shown in Figure 14, explained below. (See Lynch and Tuttle [1989] for details of the I/O automata model.) We begin by adopting some notational conventions.

Notational Conventions. Unless otherwise specified, p and q denote clients (threads) from P, the set of all clients (threads); g denotes a guard from G, the set of all guards; v denotes a value from V, the set of all values, and S and T denote sets of values (i.e., subsets of V). We assume that V contains a special **null** value that is never used, arrested, or passed to Liberate.

actions

Environment	ROP output
$\mathsf{HireInv}_p()$	$\mathsf{HireResp}_p(g)$
$\mathsf{FireInv}_p(g)$	$\mathsf{FireResp}_p()$
$\mathsf{PostInv}_p(g, v)$	$\mathsf{PostResp}_p()$
$\mathsf{LiberateInv}_p(S)$	$\mathsf{LiberateResp}_p(T)$
$\mathsf{Arrest}(v)$	

state variables

For each client $p \in P$:
  pc_p: {idle, hire, fire, post(g, v), liberate} **init** idle
  $guards_p$: set of guards **init** empty
For each value $v \in V$:
  $status[v]$: {injail, escaping, free} **init** free
For each guard $g \in G$:
  $post[g] : V$ **init null**;
  $trapping[g]$: **bool init** false;
$numescaping$: **int init** 0

transitions

$\mathsf{HireInv}_p()$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p = \mathsf{hire}$

$\mathsf{FireInv}_p(g)$
Pre: $pc_p = \mathsf{idle}$
  $g \in guards_p$
  $post[g] = \mathbf{null}$
Eff: $pc_p = \mathsf{fire}$
  $guards_p = guards_p - \{g\}$

$\mathsf{PostInv}_p(g, v)$
Pre: $pc_p = \mathsf{idle}$
  $g \in guards_p$
Eff: $pc_p = \mathsf{post}(g, v)$
  $post[g] = \mathbf{null}$
  $trapping[g] = \text{false}$

$\mathsf{LiberateInv}_p(S)$
Pre: $pc_p = \mathsf{idle}$
  for all $v \in S$,
    $v \neq \mathbf{null}$ and $status[v] = \mathsf{injail}$
Eff: $pc_p = \mathsf{liberate}$
  $numescaping = numescaping + |S|$
  for all $v \in S$, $status[v] = \mathsf{escaping}$

$\mathsf{Arrest}(v)$
Pre: $status[v] = \mathsf{free}$
  $v \neq \mathbf{null}$
Eff: $status[v] = \mathsf{injail}$
  for all g such that $post[g] = v$,
    $trapping[g] = \text{true}$

$\mathsf{HireResp}_p(g)$
Pre: $pc_p = \mathsf{hire}$
  $g \in G$
  $g \notin \bigcup_q guards_q$
Eff: $pc_p = \mathsf{idle}$
  $guards_p = guards_p \cup \{g\}$

$\mathsf{FireResp}_p()$
Pre: $pc_p = \mathsf{fire}$
Eff: $pc_p = \mathsf{idle}$

$\mathsf{PostResp}_p()$
Pre: for some g, v, $pc_p = \mathsf{post}(g, v)$
Eff: $pc_p = \mathsf{idle}$
  $post[g] = v$
  $trapping[g] = (status[v] = \mathsf{injail})$

$\mathsf{LiberateResp}_p(T)$
Pre: $pc_p = \mathsf{liberate}$
  for all $v \in T$,
    $status[v] = \mathsf{escaping}$
    and for all $g \in \bigcup_q guards_q$,
      ($post[g] \neq v$ or $\neg trapping[g]$)
Eff: $pc_p = \mathsf{idle}$
  $numescaping = numescaping - |T|$
  for all $v \in T$, $status[v] = \mathsf{free}$

Fig. 14. I/O Automaton specifying the Repeat Offender Problem.

The automaton consists of a set of environment actions and a set of ROP output actions. Each action consists of a *precondition* for performing the action and the *effect* on state variables of performing the action. Most environment actions are invocations of ROP operations, and are paired with corresponding ROP output actions that represent the system's response to the invocations. In particular, the $\mathsf{PostInv}_p(g, v)$ action models client p invoking PostGuard(g, v), and the $\mathsf{PostResp}_p()$ action models the completion of this procedure. The $\mathsf{HireInv}_p()$ action models client p invoking HireGuard(), and the corresponding $\mathsf{HireResp}_p(g)$ action models the system assigning guard g to p. The $\mathsf{FireInv}_p(g)$ action models client p calling FireGuard(g), and the $\mathsf{FireResp}_p()$ action models

the completion of this procedure. The LiberateInv$_p(S)$ action models client p calling Liberate(S) to help the values in S start escaping, and the LiberateResp$_p(T)$ action models the completion of this procedure with a set of values T that have finished escaping. Finally, the Arrest(v) action models the environment arresting value v.

The state variable *status*[v] records the current status of value v, which can be free, injail, or escaping. Transitions between the status values are caused by calls to and returns from ROP procedures, as well as by the application-specific Arrest action, as described above. The *post* variable maps each guard to the value (if any) it currently guards. The pc_p variable models control flow of client p, for example ensuring that p does not invoke a procedure before the previous invocation completes; pc_p also encodes parameters passed to the corresponding procedures in some cases. The $guards_p$ variable represents the set of guards currently employed by client p. The *numescaping* variable is an auxiliary variable used to specify nontriviality properties, as discussed later. Finally, *trapping* maps each guard g to a boolean value that is true iff g has been posted on some value v, and has not subsequently been reposted (or removed), and at some point since the guard was posted on v, v has been injail (i.e., it captures the notion of guard g trapping the value on which it has been posted). This is used by the LiberateResp action to determine whether v can be returned. (Recall that a value should not be returned if it is trapped.)

Preconditions on the invocation actions specify assumptions about the circumstances under which the application invokes the corresponding ROP procedures. Most of these preconditions are mundane well-formedness conditions, such as the requirement that a client posts only guards that it currently employs. The precondition for LiberateInv captures the assumption that the application passes only injail values to Liberate, and the precondition for the Arrest action captures the assumption that only free values are arrested. The application designer must determine how these guarantees are made.

Preconditions on the response actions specify the circumstances under which the ROP procedures can return. Again, most of these preconditions are quite mundane and straightforward. The interesting case is the precondition of LiberateResp, which states that Liberate can return a value only if it has been passed to (some invocation of) Liberate, it has not subsequently been returned by (any invocation of) Liberate, and no guard g has been continually guarding the value since the last time it was injail (recall that this is captured by *trapping*[g]).

A.1 Desirable Properties

As specified so far, an ROP solution in which Liberate always returns the empty set, or simply does not terminate, is correct. Clearly, in the context motivating our work, such solutions are unacceptable because each escaping value represents a resource that will be reclaimed only when the value is liberated (returned by some invocation of Liberate). One might be tempted to specify that every value that begins escaping is liberated (i.e., every value passed to a Liberate operation is eventually returned by some Liberate operation). However, without special operating system support, it is not possible to guarantee such

a strong property in the face of failing threads. Nonetheless, some nontriviality condition is necessary. In this section, we define the nontriviality condition that our PTB algorithm satisfies.

As mentioned in Section 3, we specify two kinds of progress properties. First, every ROP operation is wait-free: any thread that has invoked an ROP operation—that is, its *pc* is not idle—will eventually complete that operation provided that it continues to take steps. Second, the ROP implementation must guarantee the following *Value Progress* property, which says that every value that begins escaping will be liberated unless a thread fails after the value began escaping or guards the value after the last invocation of Liberate:

> Suppose that $\mathsf{LiberateInv}_p(S)$ with $v \in S$ occurs at time t, and that no thread fails at or after this event. Suppose also that for all time after some time t', $post[g] \neq v$ for all g $\in \bigcup_q guards_q$, and that $\mathsf{LiberateInv}_q(T)$ occurs at some time after t'. (If $t' < t$, this event may be the $\mathsf{LiberateInv}_p(S)$ event at time t.) Then $\mathsf{LiberateResp}(S')$ with $v \in S'$ occurs at some time after t.

In addition to the progress properties above, PTB also bounds *numescaping*, the number of escaping values, by a function of the maximum size of any set passed to Liberate, the number of threads that may be concurrently executing Liberate, and the number of guards hired. Specifically, if $k = |\bigcup_p guards_p|$, s is the maximum size of any set passed to Liberate and n is the number of threads that may be concurrently executing Liberate, then $numescaping \leq n(k + s)$.

B. FORMAL PROOF THAT PTB IMPLEMENTS ROP

In this appendix, we give a formal proof that the Pass the Buck algorithm (PTB) of Section 6 implements ROP, that is, it solves the Repeat Offender Problem.

To prove that PTB implements ROP, we first define an I/O automaton model [Lynch and Tuttle 1989] for the PTB algorithm and then show that this automaton implements the ROP automaton in Section A.

The PTB automaton, which appears in Figures 15, 16 and 17, has the same environment actions and output actions as the ROP automaton in Figure 14.

The PTB automaton also has internal actions that model the execution of code that implements the operations. Because an action changes the state of the automaton atomically, each action corresponds to code that includes at most one access to shared variables. (Access to local variables always appears atomic because other threads cannot see the effects of such access.) The names of the internal actions indicate the lines of the code that implement them in Figure 9. For $\mathsf{line2\text{-}3}_p$, an occurrence of the action corresponds to a single iteration of the **while** loop. Not all lines of code are modeled by internal actions: Lines 1 and 11 are collapsed into the invocation actions because they access only local variables, and lines 6, 8, 10 and 32 are modeled by the response actions. We discuss the internal actions of Liberate in more detail below.

The PTB automaton specified here can be viewed as the composition of two subautomata, representing the environment and the PTB algorithm, and the state variables defined in Figure 15 can be partitioned between these

actions

Environment	PTB output	PTB internal
$\mathsf{HireInv}_p()$	$\mathsf{HireResp}_p(g)$	$\mathsf{line2\text{-}3}_p$
$\mathsf{FireInv}_p(g)$	$\mathsf{FireResp}_p()$	$\mathsf{line4}_p$
$\mathsf{PostInv}_p(g, v)$	$\mathsf{PostResp}_p()$	$\mathsf{line5}_p$
$\mathsf{LiberateInv}_p(S)$	$\mathsf{LiberateResp}_p(T)$	$\mathsf{line7}_p$
$\mathsf{Arrest}(v)$		$\mathsf{line9}_p$
		$\mathsf{line12}_p$
		$\mathsf{line13+}_p$
		$\mathsf{line15+}_p$
		$\mathsf{line18+}_p$
		$\mathsf{line22+}_p$
		$\mathsf{line25+}_p$
		$\mathsf{line28+}_p$
		$\mathsf{line31}_p$

state variables

(internal PTB state variables)
GUARDS: **array**[0..MG-1] **of bool init** all false;
MAXG: **int init** 0;
POST: **array**[0..MG-1] **of value init** all **null**;
HNDOFF: **array**[0..MG-1] **of value_set init** all empty;
For each thread $p \in P$:
ln_p: $\{\mathsf{idle}, 2, 4, 5, 6, 7, 8, 9, 10, 12, 13, 15, 18, 22, 25, 28, 31, 32\}$ **init** idle
i_p: **int**
v_p: **value**
vs_p: **value_set**
max_p: **int**
$\mathrm{attempts}_p$: **value**

(environment state variables)
For each thread $p \in P$:
pc_p: $\{\mathsf{idle}, \mathsf{hire}, \mathsf{fire}, \mathsf{post}(h, v), \mathsf{liberate}\}$ **init** idle
$guards_p$: set of guards **init** empty
For each value $v \in V$:
$status[v]$: {injail, escaping, free} **init** free
For each guard $g \in G$:
$post[g]$: V **init null**
$trapping[g]$: **bool init** false
$numescaping$: **int init** 0

Fig. 15. Actions and state variables for the PTB automaton.

subautomata. The *environment state variables* are exactly the variables of the ROP automaton. These variables are used to determine when the environment actions are enabled. The *internal PTB state variables* represent state used by the PTB algorithm. These consist of the shared global variables GUARDS, POST, HNDOFF and MAXG; the local variables (including the parameters) of each operation; and a local program counter ln_p that records the next line of code to be executed by thread p.

The environment actions of the PTB automaton are identical to their counterparts in the ROP automaton except that they may also set some local variables. The response actions are enabled by the value of the local program counter being equal to the line number corresponding to a return statement. They set the program counter to idle and modify the environment state variables in the

HireGuard

$\mathsf{HireInv}_p()$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p = \mathsf{hire}$
$\mathrm{i}_p = 0$
$ln_p = 2$

$\mathsf{line2\text{-}3}_p$
Pre: $ln_p = 2$
Eff: if $\neg\mathrm{GUARDS}[\mathrm{i}_p]$ then
$\mathrm{GUARDS}[\mathrm{i}_p] = \mathrm{true}$
$ln_p = 4$
else
$\mathrm{i}_p = \mathrm{i}_p + 1$

$\mathsf{line4}_p$
Pre: $ln_p = 4$
Eff: $\mathrm{max}_p = \mathrm{MAXG}$
if $\mathrm{max}_p < \mathrm{i}_p$ then
$ln_p = 5$
else
$ln_p = 6$

$\mathsf{line5}_p$
Pre: $ln_p = 5$
Eff: if $\mathrm{MAXG} = \mathrm{max}_p$ then $\mathrm{MAXG} = \mathrm{i}_p$
$ln_p = 4$

$\mathsf{HireResp}_p(g)$
Pre: $ln_p = 6$
$g = \mathrm{i}_p$
Eff: $ln_p = \mathsf{idle}$
$pc_p = \mathsf{idle}$
$guards_p = guards_p \cup \{g\}$

FireGuard

$\mathsf{FireInv}_p(g)$
Pre: $pc_p = \mathsf{idle}$
$g \in guards_p$
$post[g] = \mathbf{null}$
Eff: $pc_p = \mathsf{fire}$
$guards_p = guards_p - \{g\}$
$\mathrm{i}_p = g$
$ln_p = 7$

$\mathsf{line7}_p$
Pre: $ln_p = 7$
Eff: $\mathrm{GUARDS}[\mathrm{i}_p] = \mathrm{false}$
$ln_p = 8$

$\mathsf{FireResp}_p()$
Pre: $ln_p = 8$
Eff: $ln_p = \mathsf{idle}$
$pc_p = \mathsf{idle}$

PostGuard

$\mathsf{PostInv}_p(g, v)$
Pre: $pc_p = \mathsf{idle}$
$g \in guards_p$
Eff: $pc_p = \mathsf{post}(g, v)$
$post[g] = \mathbf{null}$
$trapping[g] = \mathrm{false}$
$\mathrm{v}_p = v$
$\mathrm{i}_p = g$
$ln_p = 9$

$\mathsf{line9}_p$
Pre: $ln_p = 9$
Eff: $\mathrm{POST}[\mathrm{i}_p] = \mathrm{v}_p$
$ln_p = 10$

$\mathsf{PostResp}_p()$
Pre: $ln_p = 10$
Eff: $ln_p = \mathsf{idle}$
$pc_p = \mathsf{idle}$
$post[\mathrm{i}_p] = \mathrm{v}_p$
$trapping[\mathrm{i}_p] = (status[\mathrm{v}_p] = \mathsf{injail})$

Arrest

$\mathsf{Arrest}(v)$
Pre: $status[v] = \mathsf{free}$
$v \neq \mathbf{null}$
Eff: $status[v] = \mathsf{injail}$
for all g such that $post[g] = v$,
$trapping[g] = \mathrm{true}$

Fig. 16. Transitions for the PTB automaton, part 1: All but Liberate.

same way as the ROP automaton does.[16] Like the response actions, the internal PTB actions are enabled by the local program counter having the appropriate line number, and their effects clauses straightforwardly express the effects of executing the corresponding lines of code.

In addition to the environment assumptions expressed in the ROP automaton, the PTB algorithm assumes that the number of guards simultaneously

[16]That the environment state variables are modified in the same way by both automata is immediate from the definition in all cases except that of PostResp actions. Invariant 1 below establishes that in any reachable state of this automaton, the PostResp actions also update the environment state variables in the same way as in the ROP automaton.

Liberate LiberateInv$_p(S)$
Pre: $pc_p =$ idle
for all $v \in S$,
$status[v] =$ injail
$v \neq$ **null**
Eff: $pc_p =$ liberate
$numescaping = numescaping + |S|$
for all $v \in S$, $status[v] =$ escaping
$\mathrm{vs}_p = S$
$\mathrm{i}_p = 0$
$ln_p = 12$

line12$_p$
Pre: $ln_p = 12$
Eff: if $\mathrm{i}_p \leq$ MAXG then
$ln_p = 13$
else
$ln_p = 32$

line13-14$_p$
Pre: $ln_p = 13$
Eff: $\mathrm{attempts}_p = 0$
$\mathrm{h}_p = \mathrm{HNDOFF}[\mathrm{i}_p]$
$ln_p = 15$

line15-17$_p$
Pre: $ln_p = 15$
Eff: $\mathrm{v}_p = \mathrm{POST}[\mathrm{i}_p]$
if $\mathrm{v}_p \neq$ **null** $\wedge\ \mathrm{v}_p \in \mathrm{vs}_p$ then
$ln_p = 18$
else
$ln_p = 28$

line28-30$_p$
Pre: $ln_p = 28$
Eff: if $\mathrm{h}_p.val \neq$ **null** $\wedge\ \mathrm{h}_p.val \neq \mathrm{v}_p$
$\wedge\ \mathrm{HNDOFF}[\mathrm{i}_p] = \mathrm{h}_p$ then
$\mathrm{HNDOFF}[\mathrm{i}_p] = \langle$**null**$, \mathrm{h}_p.ver + 1\rangle$
$\mathrm{vs}_p = \mathrm{vs}_p \cup \{\mathrm{h}_p.val\}$
$ln_p = 31$

line18-21$_p$
Pre: $ln_p = 18$
Eff: if $\mathrm{HNDOFF}[\mathrm{i}_p] = \mathrm{h}_p$ then
$\mathrm{HNDOFF}[\mathrm{i}_p] = \langle \mathrm{v}_p, \mathrm{h}_p.ver + 1\rangle$
$\mathrm{vs}_p = \mathrm{vs}_p - \{\mathrm{v}_p\}$
if $\mathrm{h}_p.val \neq$ **null** then
$\mathrm{vs}_p = \mathrm{vs}_p \cup \{\mathrm{h}_p.val\}$
$ln_p = 31$
else
$ln_p = 22$

line22-24$_p$
Pre: $ln_p = 22$
Eff: $\mathrm{attempts}_p = \mathrm{attempts}_p + 1$
if $\mathrm{attempts}_p = 3$ then
$ln_p = 31$
else
$\mathrm{h}_p = \mathrm{HNDOFF}[\mathrm{i}_p]$
$ln_p = 25$

line25-26$_p$
Pre: $ln_p = 25$
Eff: if $(\mathrm{attempts}_p = 2 \wedge \mathrm{h}_p.val \neq$ **null**$)$
$\vee\ \mathrm{v}_p \neq \mathrm{POST}[\mathrm{i}_p]$ then
$ln_p = 31$
else
$ln_p = 18$

line31$_p$
Pre: $ln_p = 31$
Eff: $\mathrm{i}_p = \mathrm{i}_p + 1$
$ln_p = 12$

LiberateResp$_p(T)$
Pre: $ln_p = 32$
$T = \mathrm{vs}_p$
Eff: $ln_p =$ idle
$pc_p =$ idle
$numescaping = numescaping - |T|$
for all $v \in T$, $status[v] =$ free

Fig. 17. Transitions for the PTB automaton, continued: Liberate.

employed (including those in the midst of being hired or fired) is bounded by MG; the set G of guards is $\{0, \ldots, \mathrm{MG} - 1\}$. Failure to satisfy this assumption may result in an attempt at line 2 to write beyond the end of the POST array, possibly overwriting unrelated data structures and causing arbitrary failures. As in the code in Figure 9, we simply assume that this does not happen; we do not specify the behavior of the automaton when $\mathrm{i}_p \geq \mathrm{MG}$ in the execution of either line2-3$_p$ or line7$_p$.

We now prove several invariants and lemmas about the PTB automaton. Of these, Invariants 1, 2, 12 and 18 are the ones actually used in the proof of Theorem B.5, which says that PTB implements ROP. The rest are used to prove Invariants 12 and 18. Informally, Invariant 1 says that pc_p has the appropriate value when p is executing a PTB operation. Invariant 2 says that every guard is employed by at most one thread, and that if g is employed then

GUARDS[g] = *true*.[17] Invariant 12 says that the escaping values are those that appear in the HNDOFF array or in the vs_p set of some thread p with pc_p = liberate, and that each escaping value appears in exactly one of these locations. Invariant 18 says that if a guard g traps a value v and v is in the vs_p set of some thread executing Liberate, then that thread has not yet finished processing g.

We prove most invariants by induction on the number of transitions in an execution. That is, we prove that the invariant holds in the initial state and that for each reachable state s, if the invariant holds in s and $\text{s} \xrightarrow{a} s'$ then the invariant holds in s'. For many of the invariants, checking this is straightforward, and we omit the details of the proof. Because we assume s (and thus s') is reachable, we can assume the earlier invariants hold in s and s' in the proofs of later invariants.

INVARIANT 1. *If* $ln_p \in \{2, 4, 5, 6\}$, *then* pc_p = hire. *If* $ln_p \in \{7, 8\}$, *then* pc_p = fire. *If* $ln_p \in \{9, 10\}$, *then* pc_p = post(i_p, v_p). *If* $ln_p \in \{12, 13, 15, 18, 22, 25, 28, 31, 32\}$, *then* pc_p = liberate.

PROOF. Straightforward by induction. □

INVARIANT 2. *Every guard* g *is in exactly one of the following sets:* $guards_p$ *for some* p, $\{\text{i}_p\}$ *for some* p *with* $ln_p \in \{4, 5, 6, 7\}$, *or* $\{g : \neg\text{GUARDS}[g]\}$.

PROOF. This invariant holds initially because $guards_p$ is empty and and $\neg$GUARDS[g] for all g. Suppose that the invariant holds in s and that $s \xrightarrow{a} s'$. Let S_p be $\{\text{i}_p\}$ if $ln_p \in \{4, 5, 6, 7\}$ and $\emptyset$ otherwise. The only actions that change any of the sets listed in the invariant are line2-3$_p$ for some p with $\neg s.\text{GUARDS}[s.\text{i}_p]$, HireResp$_p(g)$ and FireInv$_p(g)$ for some p and g, and line7$_p$ for some p. In each case, one guard is removed from one set and added to another, so the invariant is preserved.

For line2-3$_p$ with $\neg s.\text{GUARDS}[s.\text{i}_p]$, $s.\text{i}_p$ is removed from $\{g : \neg$ GUARDS[g]$\}$ and added to S_p.

For HireResp$_p(g)$, $g = s.\text{i}_p$, $s.ln_p = 6$ and $s'.ln_p$ = idle, so g is removed from S_p and added to $guards_p$.

For FireInv$_p(g)$, g is removed from $guards_p$ and added to S_p.

Finally, for line7$_p$, $s.\text{i}_p$ is removed from S_p and added to $\{g : \neg$ GUARDS[g]$\}$. □

INVARIANT 3. *For all* p, $ln_p \in \{9, 10\} \Longrightarrow \text{i}_p \in guards_p$.

PROOF. Straightforward by induction. □

INVARIANT 4. *For all* p *and* q, *if* $ln_p \in \{9, 10\}$ *and* $ln_q \in \{9, 10\}$, *then* $i_p \neq i_q$.

PROOF. Immediate from Invariants 2 and 3. □

INVARIANT 5. *For all* p, $ln_p = 10 \Longrightarrow \text{POST}[i_p] = v_p$.

PROOF. Straightforward by induction with Invariant 4. □

[17]To make it provable by induction, the actual invariant is strengthened slightly. This is also true for Invariant 18.

INVARIANT 6. *For all* g,

$$post[g] = \begin{cases} \textbf{null} & \text{if } \exists_p(ln_p \in \{9, 10\} \wedge i_p = g) \\ POST[g] & \text{otherwise.} \end{cases}$$

PROOF. Straightforward by induction with Invariants 1, 4 and 5. □

INVARIANT 7. *For all* p, $ln_p = 5 \Longrightarrow max_p < i_p$.

PROOF. Straightforward by induction. □

INVARIANT 8. *For all* p, $ln_p \in \{2, 4, 5\} \Longrightarrow i_p \geq 0$.

PROOF. Straightforward by induction. □

INVARIANT 9. *For all* g *and* p, *if* $g \in guards_p$ *or* $ln_p = 6 \wedge \mathrm{i}_p = g$ *then* $0 \leq g \leq \mathrm{MAXG}$.

PROOF. Straightforward by induction with Invariants 7 and 8. □

INVARIANT 10. *For all* g, *if* $post[g] \neq \textbf{null}$ *then* $0 \leq g \leq MAXG$.

PROOF. Straightforward by induction with Invariants 3, 7 and 9. □

INVARIANT 11. *For all* p, *if* $ln_p \in \{18, 22, 25\}$ *then* $v_p \in vs_p$.

PROOF. Straightforward by induction. □

INVARIANT 12. *Every non-***null** *value is in exactly one of the following sets:* $\{HNDOFF[g].value\}$ *for some* g, vs_p *for some* p *with* $pc_p =$ liberate, *and* $\{v : status[v] \neq \text{escaping}\}$.

PROOF. This invariant holds initially because HNDOFF$[g]$.value $=$ **null** for all g, $pc_p =$ idle for all p, and $status[v] =$ injail $\neq$ escaping for all v. Suppose that s is a reachable state in which the invariant holds, and that $s \xrightarrow{a} s'$. Let $HS_g = \{\text{HNDOFF}[g].\text{value}\}$ and VS_p be vs_p if $pc_p =$ liberate and $\emptyset$ otherwise. With Invariant 1, it is easy to see that the only actions that change any of the sets in the invariant are LiberateInv$_p(S)$ and LiberateResp$_p(S)$ for some p and S, line18-21$_p$ for some p with s.HNDOFF$[s.\mathrm{i}_p] = s.\mathrm{h}_p$, and line28-30$_p$ for some p with $s.\mathrm{h}_p$.value $\notin \{\textbf{null}, s.\mathrm{v}_p\}$ and s.HNDOFF$[s.\mathrm{i}_p] = s.\mathrm{h}_p$.

For LiberateInv$_p(S)$, every value in S is removed from $\{v : status[v] \neq$ escaping$\}$ and added to VS_p.

For LiberateResp$_p(S)$, $s.pc_p =$ liberate by Invariant 1, so $S = VS_p$. Thus, every value in S is removed from VS_p and added to $\{v : status[v] \neq \text{escaping}\}$.

For line18-21$_p$ with s.HNDOFF$[s.\mathrm{i}_p] = s.\mathrm{h}_p$, values other than $s.\mathrm{v}_p$ and $s.\mathrm{h}_p$ are not added to or removed from any of the sets. By Invariant 11, $s.\mathrm{v}_p \in s.\mathrm{vs}_p$ (because $s.ln_p = 18$), so $s.\mathrm{v}_p$ is removed from VS_p and added to $HS_{s.\mathrm{i}_p}$. If $s.\mathrm{h}_p$.value $\neq$ **null** then by the inductive hypothesis, $s.\mathrm{h}_p \notin s.\mathrm{vs}_p$ and so $s.\mathrm{h}_p$ is removed from $HS_{s.\mathrm{i}_p}$ and added to VS_p.

For line28-30$_p$ with $s.\mathrm{h}_p$.value $\notin \{\textbf{null}, s.\mathrm{v}_p\}$ and s.HNDOFF$[s.\mathrm{i}_p] = s.\mathrm{h}_p$, by the inductive hypothesis, $s.\mathrm{h}_p$.value $\notin s.\mathrm{vs}_p$ and so $s.\mathrm{h}_p$.value is removed from $HS_{s.\mathrm{i}_p}$ and added to VS_p.

Thus, the invariant that each value is in exactly one of the sets is preserved. □

INVARIANT 13. *For all p, if* $ln_p \in \{15, 18, 22, 25, 28, 31\}$, *then* $h_p.ver \leq$ HNDOFF$[i_p].ver$.

PROOF. Straightforward by induction. □

LEMMA B.1. *For all p and g, if s is a reachable state in which* $s.i_p = g$ *and* $s.h_p \neq s.$HNDOFF$[g]$, *and* $s \xrightarrow{a} s'$, *then* $s'.h_p \neq s'.HNDOFF[g]$ *or* $a \in \{$*line13-14*$_p$, *line22-24*$_p$. $\}$

PROOF. Fix p and g. If $a \notin \{$line13-14$_p$, line22-24$_p\}$ then $s'.\mathrm{h}_p = s.\mathrm{h}_p$ and, unless there is some q with $s.\mathrm{i}_q = g$, $s.\mathrm{h}_q = s.$HNDOFF$[g]$ and either $a =$ line18-21$_q$ or $a =$ line28-30$_q$ and $s.\mathrm{h}_q.$value $\notin \{$**null**, $s.\mathrm{v}_q\}$, $s'.$HNDOFF$[g] = s.$HNDOFF$[g]$. If there is such a q then $s'.$HNDOFF$[g].ver = s.$HNDOFF$[g].ver + 1$, and by Invariant 13, $s'.\mathrm{h}_p.ver = s.\mathrm{h}_p.ver \leq s.$HNDOFF$[g].ver$, so $s'.\mathrm{h}_p \neq s'.$HNDOFF$[g]$. □

LEMMA B.2. *For all g and* $v \neq$ **null**, *if s is a reachable state,* $s \xrightarrow{a} s'$, *and* $s'.post[g] = v$ *and* $s'.trapping[g]$ *then either* $s.post[g] = v$ *and* $s.trapping[g]$ *or* $s'.status[v] =$ injail.

PROOF. Fix g and v. If $s'.post[g] = v$ and $s'.trapping[g]$ and $s.post[g] \neq v \vee \neg s.trapping[g]$ then either $a =$ PostResp$_p()$ for some p with $s.\mathrm{i}_p = g, s.\mathrm{v}_p = v$ and $s.status[v] =$ injail, or $a =$ Arrest(v) with $s.post[g] = v$. In either case, $s'.status[v] =$ *injail*. □

INVARIANT 14. *For all* g, $v \neq$ **null** *and p, if* $post[g] = v = HNDOFF[g].value, trapping[g]$, $\mathrm{i}_p = g$ *and* $h_p = HNDOFF[g]$ *then* $ln_p \neq 18$ *and* $ln_p = 28 \Longrightarrow v_p = v$.

PROOF. Fix g, v and p. This invariant holds initially because $trapping[g] = false$. Suppose it holds in a reachable state s, $s \xrightarrow{a} s'$, $s'.post[g] = v = s'.$HNDOFF$[g]$.value , $s'.trapping$[g], and $s'.\mathrm{h}_p = s'.$HNDOFF$[g]$.

If $s.\mathrm{i}_p \neq g$ then $s'.\mathrm{i}_p \neq g$ or $s'.ln_p \notin \{18, 28\}$, so the invariant holds in s'.

If $s.\mathrm{i}_p = g$ and $s.\mathrm{h}_p \neq s.$HNDOFF$[g]$, then, by Lemma B.1, because $s'.\mathrm{h}_p = s'.$HNDOFF$[g]$, we have $a = \{$line13-14$_p$, line22-24$_p\}$, so $s'.ln_p \notin \{18, 28\}$.

If $s.$HNDOFF$[g]$.value $\neq v$ then $a =$ line18-21$_q$ for some q with $s.\mathrm{i}_q = g$ and $s.\mathrm{h}_q = s.$HNDOFF$[g]$. If $s'.ln_p \notin \{18, 28\}$ then the invariant holds in s'. Otherwise, $s'.ln_p \in \{18, 28\}$, so $s.ln_p \in \{18, 28\}$ (because $a =$ line18-21$_q$, so $p \neq q$). In the latter case, by Invariant 13, $s.\mathrm{h}_p.ver \leq s.$HNDOFF$[g].ver$, and because $s'.$HNDOFF$[g].ver = s.$HNDOFF$[g].ver + 1$ and $s'.\mathrm{h}_p.ver = s.\mathrm{h}_p.ver$, we have $s'.\mathrm{h}_p \neq s'.$HNDOFF$[g]$, so the invariant holds in s'.

Otherwise, $s.$HNDOFF$[g]$.value $= v$, $s.\mathrm{i}_p = g$ and $s.\mathrm{h}_p = s.$HNDOFF$[g]$. By Invariant 12, $s'.status[v] =$ escaping $\neq$ injail, so by Lemma B.2, $s.post[g] = v$ and $s.trapping[g]$. Thus, by the inductive hypothesis, $s.ln_p \neq 18$ and $s.ln_p = 28 \Longrightarrow s.\mathrm{v}_p = v$. If $a \notin \{$line15-17$_p$, line25-26$_p\}$ then $s'.ln_p \neq 18$ and $s'.ln_p = 28 \Longrightarrow s'.\mathrm{v}_p = v$, so the invariant holds in s'. Otherwise, $s.ln_p \in \{15, 25\}$, so by

Invariant 1, $s.pc_p = \mathsf{liberate}$, and by Invariant 12, $v \notin s.\mathrm{vs}_p$. Also, by Invariant 6, $s.\mathrm{POST}[g] = s.post[g] = v$.

If $a = \mathsf{line15\text{-}17}_p$ then $s'.\mathrm{v}_p = s.\mathrm{POST}[g] = v$ and $s'.ln_p = 28$, so the invariant holds in s'.

If $a = \mathsf{line25\text{-}26}_p$ then by Invariant 11, $s.\mathrm{v}_p \in s.\mathrm{vs}_p$, so $s.\mathrm{v}_p \neq v = s.\mathrm{POST}[g]$. Thus, $s'.ln_p = 31$ and the invariant holds in s'. □

LEMMA B.3. *For all g, $v \neq$ **null** and p, if s is a reachable state, $s \xrightarrow{a} s'$, $s.post[g] = v = s.$HNDOFF$[g].value$, $s.trapping[g]$, $s.i_p = g$ and $s.ln_p \in \{18, 28\}$ then $v \notin s'.vs_p$.*

PROOF. Because $v = s.$HNDOFF$[g]$.value, by Invariants 1 and 12, $v \notin s.\mathrm{vs}_p$. Either $s'.\mathrm{vs}_p = s.\mathrm{vs}_p$, and so $v \notin s'.\mathrm{vs}_p$, as required, or $a \in \{\mathsf{line18\text{-}21}_p, \mathsf{line28\text{-}30}_p\}$ and $s.\mathrm{h}_p = s.$HNDOFF$[s.\mathrm{i}_p]$. In the latter case, because $s.\mathrm{i}_p = g$ and $s.ln_p \in \{18, 28\}$, by Invariant 14, we have $s.ln_p \neq 18$ and $s.\mathrm{v}_p = v \notin s.\mathrm{vs}_p$. Thus, $a = \mathsf{line28\text{-}30}_p$, and because $s.\mathrm{h}_p.\mathrm{value} = v$, $v \notin s.\mathrm{vs}_p = s'.\mathrm{vs}_p$. □

LEMMA B.4. *For all g and p, if $s \xrightarrow{a} s'$, $s'.i_p = g$ and $s'.ln_p \in \{15, 18, 22, 25\}$, then $s.\mathrm{i}_p = g$ and $s.\mathrm{vs}_p = s'.vs_p$.*

PROOF. Straightforward by inspection. □

INVARIANT 15. *For all g, $v \neq$ **null**, p and $q \neq p$, if $post[g] = v \in vs_p$, $trapping[g]$, $i_p = i_q = g$, $ln_p \in \{15, 18, 22, 25\}$, $ln_q = 18$ and $h_q =$ HNDOFF$[g]$ then $h_p =$ HNDOFF$[g]$, attempts$_p = 0$ and $ln_p \neq 22$.*

PROOF. Fix g, v, p and q. This invariant holds initially because $trapping[g] = false$. Suppose that it holds in a reachable state s, $s \xrightarrow{a} s'$, and $s'.post[g] = v \in s'.\mathrm{vs}_p$, $s'.trapping[g]$, $s'.\mathrm{i}_p = s'.\mathrm{i}_q = g$, $s'.ln_p \in \{15, 18, 22, 25\}$, $s'.ln_q = 18$ and $s'.\mathrm{h}_q = s'.$HNDOFF$[g]$. By Lemma B.4, $s.\mathrm{i}_p = s.\mathrm{i}_q = g$ and $v \in s.\mathrm{vs}_p = s'.\mathrm{vs}_p$.

If $s.ln_p \notin \{15, 18, 22, 25\}$, then $a = \mathsf{line13\text{-}14}_p$, so $s'.\mathrm{h}_p = s.$HNDOFF$[s.\mathrm{i}_p] = s'.$HNDOFF$[g]$, $s'.$attempts$_p = 0$ and $s'.ln_p = 15 \neq 22$, as required.

Otherwise, $s.ln_p \in \{15, 18, 22, 25\}$, so by Invariant 1, $s.pc_p = \mathsf{liberate}$, and by Invariant 12, $s'.status[v] = \mathsf{escaping} \neq \mathsf{injail}$ (because $v \in s'.\mathrm{vs}_p$), so by Lemma B.2, $s.post[g] = v$ and $s.trapping$[g]. Because $s'.ln_q = 18$, we have $a \notin \{\mathsf{line13\text{-}14}_q, \mathsf{line22\text{-}24}_q\}$, so by Lemma B.1, $s.\mathrm{h}_q = s.$HNDOFF$[g]$. Also, $s.ln_q = 18$ because otherwise, either $a = \mathsf{line15\text{-}17}_q$ and $s.\mathrm{POST}[g] \in s.\mathrm{vs}_q$ or $a = \mathsf{line25\text{-}26}_q$ and $s.\mathrm{v}_q = s.\mathrm{POST}[g]$, which, by Invariant 11, also implies $s.\mathrm{POST}[g] \in s.\mathrm{vs}_q$. However, both these cases are impossible because, by Invariant 6, $s.\mathrm{POST}[g] = v$, and by Invariants 1 and 12, $v \notin s.\mathrm{vs}_q$ because $v \in s.\mathrm{vs}_p$ and $p \neq q$.

Thus, by the inductive hypothesis, $s.\mathrm{h}_p = s.$HNDOFF$[g]$, $s.$attempts$_p = 0$ and $s.ln_p \neq 22$. So $s'.$attempts$_p = s.$attempts$_p = 0$, and $s'.\mathrm{h}_p = s.\mathrm{h}_p = s.$HNDOFF$[g]$. Because $s.ln_q = s'.ln_p[q] = 18$, we have $s'.\mathrm{h}_q = s.\mathrm{h}_q$, thus, $s'.\mathrm{h}_p = s.$HNDOFF$[g] = s.\mathrm{h}_q = s'.\mathrm{h}_q = s'.$HNDOFF$[g]$. Finally, $s'.ln_p \neq 22$ because $s.ln_p \neq 22$ and $s.$HNDOFF$[s.\mathrm{i}_p] = s.\mathrm{h}_p$. Thus, the invariant holds in s'. □

INVARIANT 16. *For all p,* $ln_p = 15 \Longrightarrow attempts_p = 0$.

PROOF. Straightforward by induction. □

INVARIANT 17. *For all p, if* $ln_p = 22$ *then* $\mathrm{h}_p \neq HNDOFF[i_p]$.

PROOF. Straightforward by induction with Lemma B.1. □

INVARIANT 18. *For all g and* $v \neq$ **null**, *if* $post[g] = v$ *and* $trapping[g]$, *then*

(1) *for all* $h > g$, $v \neq \mathrm{HNDOFF}[h].value$;
(2) *for all p, if* $pc_p =$ liberate *and* $v \in \mathrm{vs}_p$, *then* $ln_p \in \{12, 13, 15, 18, 22, 25, 28, 31\}$ *and* $i_p \leq g$;
(3) *for all p, if* $pc_p =$ liberate, $v \in vs_p$, $i_p = g$ *and* $ln_p = \{18, 22, 25, 28, 31\}$, *then*
—$\mathrm{v}_p = v$,
—$ln_p \notin \{28, 31\}$,
—$attempts_p = 2 \Longrightarrow \mathrm{h}_p = HNDOFF[g] \wedge h_p.value =$ **null**,
—$attempts_p = 1 \Longrightarrow h_p = HNDOFF[g] \vee HNDOFF[g].value =$ **null**.

PROOF. Fix g and v. This invariant holds initially because $trapping[g] = false$. Suppose it holds in a reachable state s, $s \xrightarrow{a} s'$, and $s'.post[g] = v$ and $s'.trapping[g]$. By Lemma B.2, either $s.post[g] = v$ and $s.trapping[g]$ or $s'.status[v] =$ injail $\neq$ escaping. In the latter case, by Invariant 12, $v \neq s'.\mathrm{HNDOFF}[h].\mathrm{value}$ for all h and $v \notin s'.\mathrm{vs}_p$ for any p such that $pc_p =$ liberate, so this invariant holds in s'. For the rest of the proof, we consider the case in which $s.post[g] = v$ and $s.trapping[g]$.

For the first clause, fix $h > g$. By the inductive hypothesis, $v \neq s.\mathrm{HNDOFF}[h].\mathrm{value}$, and thus, $v \neq s'.\mathrm{HNDOFF}[h].\mathrm{value}$ unless $a =$ line18-21$_p$ for some p with $s.\mathrm{i}_p = h$, $s.\mathrm{v}_p = v$ and $s.\mathrm{h}_p = s.\mathrm{HNDOFF}[h]$. However, if this action is enabled in s then $s.ln_p = 18$, so by Invariant 1, $s.pc_p =$ liberate, and by the inductive hypothesis (second clause), either $v \notin s.\mathrm{vs}_p$ or $s.\mathrm{i}_p \leq g$. The first possibility contradicts Invariant 11 because $s.\mathrm{v}_p = v$; the second contradicts $s.\mathrm{i}_p = h > g$. Thus, $v \neq s'.\mathrm{HNDOFF}[h].\mathrm{value}$.

For the second and third clauses, fix p. If $s.pc_p \neq$ liberate then either $s'.pc_p \neq$ liberate, in which case these clauses hold in s', or $a =$ LiberateInv(S) for some S. In the latter case, $s'.ln_p = 12$ and, by Invariant 10, $s'.\mathrm{i}_p = 0 \leq g$, so both clauses hold in s'.

If $s.pc_p =$ liberate and $v \notin s.\mathrm{vs}_p$ then either $v \notin s'.\mathrm{vs}_p$, in which case both clauses hold in s', or $a \in \{$line18-21$_p$, line28-30$_p\}$, $s.\mathrm{HNDOFF}[s.\mathrm{i}_p] = s.\mathrm{h}_p$ and $s.\mathrm{h}_p.\mathrm{value} = v$. In the latter case, $s'.ln_p = 31$ and $s'.\mathrm{i}_p = s.\mathrm{i}_p \leq g$ because by the inductive hypothesis (first clause), $s.\mathrm{HNDOFF}[h].\mathrm{value} \neq v = s.\mathrm{HNDOFF}[s.\mathrm{i}_p]$ for all $h > g$. Furthermore, because $s.ln_p \in \{18, 28\}$, if $s.\mathrm{i}_p = g$ then by Lemma B.3, $v \notin s'.\mathrm{vs}_p$. So both clauses hold in s'.

It remains only to check the case in which $s.pc_p =$ liberate and $v \in s.\mathrm{vs}_p$. By the inductive hypothesis (second clause), $s.ln_p \in \{12, 13, 15, 18, 22, 25, 28, 31\}$ and $s.\mathrm{i}_p \leq g$. Thus, $s'.ln_p \in \{12, 13, 15, 18, 22, 25, 28, 31\}$ and $s'.\mathrm{i}_p \leq g$ unless either $a =$ line12$_p$ and $s.\mathrm{i}_p > s.\mathrm{MAXG}$ or $a =$ line31$_p$ and $s.\mathrm{i}_p = g$. The first case is impossible because, by Invariant 10, $s.\mathrm{MAXG} \geq g \geq s.\mathrm{i}_p$; the second

case is impossible because $s.\mathrm{i}_p = g \implies ln_p \neq 31$ by the third clause of the inductive hypothesis. So the second clause of the invariant holds in s'.

For the third clause, if $s.\mathrm{i}_p \neq g$ then $s'.\mathrm{i}_p \neq g$ unless $a = \mathsf{line31}_p$, in which case $s'.ln_p = 12$, so the third clause holds in s'. If $s.\mathrm{i}_p = g$ and $s.ln_p \notin \{18, 22, 25, 28, 31\}$ then $s'.ln_p \notin \{18, 22, 25, 28, 31\}$ unless $a = \mathsf{line15\text{-}17}_p$. In this case, by Invariant 6, $s.\mathrm{POST}[s.\mathrm{i}_p] = s.post[\mathrm{s.i}_p] = v \in s.\mathrm{vs}_p$, so $s'.\mathrm{v}_p = v$, $s'.ln_p = 18$ and, by Invariant 16, $s'.\mathrm{attempts}_p = s.\mathrm{attempts}_p = 0$. So the invariant holds in s'.

If $s.\mathrm{i}_p = g$ and $s.ln_p \in \{18, 22, 25, 28, 31\}$, then, by the inductive hypothesis (third clause), so $s.ln_p \notin \{28, 31\}$ and $s.\mathrm{v}_p = v$. The only actions we need to consider are $\mathsf{line18\text{-}21}_p$, $\mathsf{line22\text{-}24}_p$, $\mathsf{line25\text{-}26}_p$, and $\mathsf{line18\text{-}21}_q$ and $\mathsf{line28\text{-}30}_q$ for some $q \neq p$ with $s.\mathrm{i}_q = g$ and $s.\mathrm{h}_q = s.\mathrm{HNDOFF}[g]$. We check for each of these actions that the third clause of the invariant holds in s'.

For $a = \mathsf{line18\text{-}21}_p$, we have two cases. If $s.\mathrm{h}_p = s.\mathrm{HNDOFF}[g]$ then by Invariant 11, $v = s.\mathrm{v}_p \in s.\mathrm{vs}_p$, and by Invariants 1 and 12, $s.\mathrm{h}_p.\mathrm{value} \neq v$, so $v \notin s'.\mathrm{vs}_p$, and the third clause holds in s'. Otherwise, $s'.ln_p = 22$ and the third clause is preserved by a.

For $a = \mathsf{line22\text{-}24}_p$, by Invariant 17 and the inductive hypothesis, $s.\mathrm{attempts}_p \neq 2$, and $s.\mathrm{attempts}_p = 1 \implies s.\mathrm{HNDOFF}[g].\mathrm{value} = \mathbf{null}$. Thus, $s'.ln_p = 25$, $s'.\mathrm{v}_p = s.\mathrm{v}_p = v$, $s'.\mathrm{h}_p = s.\mathrm{HNDOFF}[g] = s'.\mathrm{HNDOFF}[g]$ and $s'.\mathrm{attempts}_p = 2 \implies s'.\mathrm{h}_p.\mathrm{value} = \mathbf{null}$.

For $a = \mathsf{line25\text{-}26}_p$, by Invariant 6 and the inductive hypothesis, $s.\mathrm{v}_p = v = s.post[\mathrm{s.i}_p] = s.\mathrm{POST}[s.\mathrm{i}_p]$ and $s.\mathrm{attempts}_p = 2 \implies s.\mathrm{h}_p.\mathrm{value} = \mathbf{null}$. Thus, $s'.ln_p = 18$ and the third clause is preserved by a.

For $a = \mathsf{line18\text{-}21}_q$ for some $q \neq p$ with $s.\mathrm{i}_q = g$ and $s.\mathrm{h}_q = s.\mathrm{HNDOFF}[g]$, by Invariant 15, $s'.\mathrm{attempts}_p = s.\mathrm{attempts}_p = 0$. Also, $s'.\mathrm{v}_p = s.\mathrm{v}_p = v$ and $s'.ln_p = s.ln_p \notin \{28, 31\}$.

For $a = \mathsf{line28\text{-}30}_q$ for some $q \neq p$ with $s.\mathrm{i}_q = g$ and $s.\mathrm{h}_q = s.\mathrm{HNDOFF}[g]$, we consider three cases. If $s.\mathrm{attempts}_p = 2$ then $s.\mathrm{h}_q.\mathrm{value} = s.\mathrm{HNDOFF}[g].\mathrm{value} = \mathbf{null}$ by the inductive hypothesis, so a preserves the third clause. If $s.\mathrm{attempts}_p = 1$ and $s.\mathrm{h}_q.\mathrm{value} \in \{\mathbf{null}, s.\mathrm{v}_q\}$ then again, a preserves the third clause. If $s.\mathrm{attempts}_p = 1$ and $s.\mathrm{h}_q.\mathrm{value} \notin \{\mathbf{null}, s.\mathrm{v}_q\}$, then $s'.\mathrm{attempts}_p = s.\mathrm{attempts}_p = 1$ and $s'.\mathrm{HNDOFF}[g].\mathrm{value} = \mathbf{null}$. □

We now prove the main theorem, which says that PTB implements ROP (assuming a well-behaved environment). This proof uses a special case of the *simulation relation*, or *refinement*, method [Lynch and Tuttle 1989].

THEOREM B.5. *The PTB automaton (including the environment part) implements the ROP automaton (including the environment part).*

PROOF. We can prove that one automaton A implements another automaton B with the same input and output actions (but different internal actions) by giving a function f from the states of A to the states of B that satisfies the following properties:

(1) If s is an initial state of A then $f(s)$ is an initial state of B.
(2) If s is a reachable state of A and $s \xrightarrow{a} s'$ then $f(s) \xrightarrow{a} f(s')$ if a is external and $f(s') = f(s)$ if a is internal.

The existence of such a function, called a *refinement* or a *simulation*, implies that A implements B [Lynch and Tuttle 1989].[18] Because the state variables of the ROP automaton (except for *numescaping*, which is a history variable) are exactly the environment state variables of the PTB automaton, we simply use the function that maps states of the PTB automaton to the states of the ROP automaton with identical values for those state variables. The first property above is satisfied because the environment state variables are initialized to the same values in both automata.

Suppose that s is a reachable state of the PTB automaton and $s \xrightarrow{a} s'$. If a is an internal action, then $f(s') = f(s)$ because the internal actions do not modify any of the environment state variables. If a is an environment action, then the precondition of a is identical in the two automata and the environment state variables are updated in the same way by both automata, so $f(s) \xrightarrow{a} f(s')$, as required.

For the PTB output actions, we need to argue that a is enabled in $f(s)$ and that both automata update the environment state variables in the same way. For HireResp, FireResp and LiberateResp actions, the latter is obvious because the statements of the effects clause that affect the environment state variables are identical.

For $\mathsf{HireResp}_p(g)$, $s.ln_p = 6$ and $s.\mathrm{i}_p = g$, so by Invariant 1, $s.pc_p = \mathsf{hire}$, and by Invariant 2,

For $\mathsf{FireResp}_p()$, $s.ln_p = 8$, so by Invariant 1, $s.pc_p = \mathsf{fire}$.

For $\mathsf{LiberateResp}_p(S)$, $s.ln_p = 32$ and $S = s.\mathrm{vs}_p$, so $s.pc_p = \mathsf{liberate}$ by Invariant 1, $s.status[v] = \mathsf{escaping}$ for all $v \in s.\mathrm{vs}_p = S$ by Invariant 12, and for all g and $v \neq \mathbf{null}$ such that $s.post[g] = v$ and $s.trapping[g]$, we have $v \notin s.\mathrm{vs}_p = S$ by Invariant 18 (second clause).

For $\mathsf{PostResp}_p()$ for some p, $s.ln_p = 10$, so by Invariant 1, $s.pc_p = \mathsf{post}(s.\mathrm{i}_p, s.\mathrm{v}_p)$. Thus, $\mathsf{PostResp}_p()$ is enabled in $f(s)$, and the environment state variables are updated in the same way in both automata, so $f(s) \xrightarrow{a} f(s')$, as required. □

REFERENCES

AGESEN, O., DETLEFS, D., FLOOD, C., GARTHWAITE, A., MARTIN, P., MOIR, M., SHAVIT, N., AND STEELE, G. 2002. DCAS-based concurrent deques. *Theory of Computing Systems 35*, 349–386. (A preliminary version appeared in the *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures*, 2000.)

ANDERSON, J. AND MOIR, M. 1997. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distrib. Comput. 11*, 1–20. (A preliminary version appeared in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 141–150.)

ANDERSON, J. AND MOIR, M. 1999. Universal constructions for large objects. *IEEE Trans. Paral. Distrib. Syst. 10*, 12, 1317–1332. (A preliminary version appeared in the *Proceedings of the 9th Annual International Workshop on Distributed Algorithms*, 1995.)

BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM, New York, 92–103.

[18]This is a special case of refinements and simulations, but it is sufficient for our purposes.

DETLEFS, D., FLOOD, C., GARTHWAITE, A., MARTIN, P., SHAVIT, N., AND STEELE, G. 2000. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*. 59–73.

DETLEFS, D., MARTIN, P., MOIR, M., AND STEELE, G. 2001. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 190–199.

DICE, D. AND GARTHWAITE, A. 2002. Mostly lock-free malloc. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ACM, New York.

DOHERTY, S., GROVES, L., LUCHANGCO, V., AND MOIR, M. 2004a. Formal verification of a practical lock-free queue algorithm. In *Proceedings of the 24th Annual International Conference on Formal Techniques for Networked and Distributed Systems* (FORTE). Lecture Notes in Computer Science, Vol. 3235, Springer Verlag, New York, 97–114.

DOHERTY, S., HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 2004b. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 31–39.

GREENWALD, M. 1999. Non-blocking synchronization and system design. Ph.D. dissertation, Stanford University Tech. Rep. STAN-CS-TR-99-1624, Palo Alto, Calif.

HARRIS, T. 2001. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing*.

HERLIHY, M. 1991. Wait-free synchronization. *ACM Trans. Prog. Lang. and Systems 13*, 1, 124–149.

HERLIHY, M., LUCHANGCO, V., MARTIN, P., AND MOIR, M. 2002. Brief announcement: Dynamic-sized lock-free data structures. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*. ACM, New York, 131.

HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 2003. Space- and time-adaptive nonblocking algorithms. In *Proceedings of Computing: The Australasian Theory Symposium (CATS)*.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C-28*, 9 (Sept.), 241–248.

LEA, D. 2003. Concurrency jsr-166 interest site. `http://gee.cs.oswego.edu/dl/concurrency-interest/`.

LYNCH, N. AND TUTTLE, M. 1989. An introduction to input/output automata. Tech. Rep. CWI-Quarterly, 2(3), Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

MARTINEZ, J. F. AND TORRELLAS, J. 2002. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

MICHAEL, M. 2002a. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Distributed Computing*. ACM, New York, 21–30.

MICHAEL, M. M. 2002b. High performance dynamic lock-free hash tables and list-based sets. In *Proceeding of the 14th Annual Symposium on Parallel Algorithms and Architectures*. ACM, New York, 73–82.

MICHAEL, M. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parall. Distrib. Syst. 15*, 6 (June), 491–504.

MICHAEL, M. AND SCOTT, M. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*. ACM, New York, 267–276.

MICHAEL, M. AND SCOTT, M. 1998. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Paral. Distrib. Comput. 51*, 1, 1–26.

MOIR, M. 1997. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 219–228.

OPLINGER, J., AND LAM, M. S. 2002. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

PRINTEZIS, T. AND DETLEFS, D. 2000. A generational mostly-concurrent garbage collector. In *Proceedings of the 2nd International Symposium on Memory Management*. ACM, New York, 143–154.

RAJWAR, R. AND GOODMAN, J. R. 2002. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

TREIBER, R. 1986. Systems programming: Coping with parallelism. Tech. Rep. RJ5118, IBM Almaden Research Center.

VALOIS, J. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 214–222. (See `http://www.cs.sunysb.edu/~valois` for errata.)

WEAVER, D., AND GERMOND, T. 1994. *The SPARC Architecture Manual Version 9*. Prentice-Hall, Englewood Cliffs, N.J.

Received June 2003; revised April 2004 and August 2004; accepted August 2004