Dynamic-sized Lockfree Data Structures

**Maurice Herlihy, Victor Luchangco,
Paul Martin, and Mark Moir**

Dynamic-sized Lockfree Data Structures

Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir

SMLI TR-2002-110 June 2002

Abstract:

We address the problem of integrating lockfree shared data structures with standard dynamic allocation mechanisms (such as **malloc** and **free**).

We have two main contributions. The first is the design and experimental analysis of two dynamic-sized lockfree FIFO queue implementations, which extend Michael and Scott's previous implementation by allowing unused memory to be freed. We compare our dynamic-sized implementations to the original on 16-processor and 64-processor multiprocessors. Our experimental results indicate that the performance penalty for making the queue dynamic-sized is modest, and is negligible when contention is not too high. These results were achieved by applying a solution to the *Repeat Offender Problem* (ROP), which we recently posed and solved.

Our second contribution is another application of ROP solutions. Specifically, we show how to use any ROP solution to achieve a general methodology for transforming lockfree data structures that rely on garbage collection into ones that use explicit storage reclamation.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email address:
mph@cs.brown.edu
victor.luchangco@sun.com
paul.a.martin@sun.com
mark.moir@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our Website, http://research.sun.com/techrep/.

Dynamic-sized Lockfree Data Structures

Maurice Herlihy
Computer Science Department
Brown University, Box 1910
Providence, RI 02912

Victor Luchangco Paul Martin Mark Moir
Sun Microsystems Laboratories
1 Network Drive, UBUR02-311
Burlington, MA 01803

1 Introduction

A shared data structure is *lockfree* if the failure or delay of some threads does not prevent the other threads from making progress. Lockfree data structures are attractive in multiprocessor environments because they alleviate or eliminate problems associated with locks, including priority inversion, convoying, and deadlock.

Modern programming environments typically provide support for dynamically allocating and freeing memory (for example, using **malloc** and **free**). In this paper, we consider the problem of integrating dynamic memory management with lockfree data structures. We focus on the problem of constructing *dynamic-sized* data structures that permit their underlying memory blocks to be allocated and freed dynamically using standard libraries. Specifically, we present two dynamic-sized lockfree FIFO queue implementations. These implementations are achieved by applying some novel techniques we recently proposed to Michael and Scott's pioneering design [4], which is not dynamic-sized because it cannot free memory for arbitrary reuse. We compare our dynamic-sized implementations to the original on 16-processor and 64-processor multiprocessors. Our experimental results indicate that the performance penalty for making the queue dynamic-sized is modest, and is negligible when contention for the queue is not too high.

Lockfree data structures do not easily lend themselves to dynamic memory management. We may safely free a block of memory (that is, make it available for reuse) only when we can guarantee that no thread will continue to treat that block as if it were still part of the original data structure. Otherwise, storing into that memory could corrupt another data structure that had happened to reuse that memory block; even reading that memory block can be problematic, as the operating system may have removed that page from the thread's address space [6].

By contrast, lock-based data structures typically support dynamic memory management in a straightforward way. For example, consider a linked list in which every thread locks a node before reading the pointer to its successor, and then discards that pointer once it unlocks the predecessor. A thread can free a node after locking its predecessor, simply because no other thread can have a pointer to that node. Without locking, however, some other mechanism is needed to prevent a thread from acquiring a pointer to the node about to be freed.

One alternative is to use a garbage collector (GC) to ensure that memory is freed only after all pointers to it have vanished. However, we want to avoid relying on GC, in part because we are interested in using lockfree data structures in the implementation of garbage collectors.

Several authors [6, 5] have proposed memory management techniques that assign version

numbers (or similar tags) to pointers and memory blocks. To dereference a pointer, an atomic compare-and-swap (CAS) operation is used in a way guaranteed to fail if the memory block has been deallocated since the pointer value was loaded. For such techniques to work, memory block version numbers must be preserved across successive allocation and reallocation. Version-numbered blocks may be kept in shared "memory pools" for reuse [2, 4, 6]. Such an approach has certain advantages: specialized, "handcrafted" memory management is typically faster than generalized library calls, especially for fixed-size blocks. Nevertheless, the approach of maintaining a pool of version-numbered memory blocks has significant limitations. Data structures implemented in this way are not truly dynamic-sized: if an object grows large and subsequently shrinks, the associated pool will contain many memory blocks that can never truly be freed (say, by a call to **free**), because the block's version number must never be overwritten. Similarly, adjacent free blocks cannot be coalesced, and so on.

The techniques proposed here provide the best of both worlds by allowing memory pools from which excess memory blocks can be freed for arbitrary reuse (using **free**) if the pool becomes too large. We emphasize that our approach does not impose special requirements on the operating system's memory manager; it works with any standard memory management library.

Valois [7] proposed a memory allocation mechanism that maintains a reference count for each memory block. Valois's API provides a SAFEREAD operation that returns a pointer and increments the associated reference count. His memory allocator has many of the same limitations as the memory pool approach outlined above: blocks cannot be coalesced, and the programmer cannot substitute alternative memory management libraries (which is often desirable or necessary for performance or other reasons).

Interestingly, the prior work whose claims come closest to ours predates the work discussed above by almost a decade. Treiber [6] proposed a technique called *obligation passing*. He describes a lockfree linked list that supports search, insert, and delete operations. While this implementation allows freed nodes to be returned to the operating system through standard interfaces, it has significant limitations: It employs a "use counter" that ensures that memory is reclaimed only by the "last" thread to access the linked list in any period. As a result, a single failed thread can prevent any memory from ever being reclaimed (defeating the main purpose of lockfree implementations). Continual access also inhibits all memory reclamation. Finally, the obligation-passing code is comingled with the linked-list code (all presented in assembly language) in such a way that it remains unclear to us how to apply the techniques to other data structures.

Our lockfree FIFO queue results are achieved using a solution to the *Repeat Offender Problem* (ROP), which we recently posed and solved [3]. We believe this technique will be applicable to a variety of purposes related to memory management in lockfree algorithms. To demonstrate another application, we have also applied our technique to achieve a two-part methodology for designing certain kinds of dynamic-sized lockfree data structures. In the first phase, we design the data structure as if GC were available (that is, we allocate memory, but never free it), which significantly simplifies the design by removing the need to reason about memory management. In the second phase, we apply a new technique called *single-word lockfree reference counting* (SLFRC) to achieve an equivalent GC-independent implementation. This result improves the original *lockfree reference counting* technique (LFRC) [1] by removing the reliance on the double compare-and-swap (DCAS) instruction, which is not widely supported.

In Section 2, we give an overview of ROP. We present our lockfree FIFO queue implementations in Section 3 and our performance experiments on these implementations in Section 4. In Section 5, we present the SLFRC methodology. Concluding remarks are in Section 6.

2 ROP Background

In this section, we describe the relevant aspects of the *Repeat Offender Problem* (ROP) and explain the guarantees made by solutions to this problem; for a more precise specification, see [3].

We consider a set of *values* and a set of *application clients*, which use the values. Each value can be free, injail or escaping. A client must not use a value that is free; the purpose of an ROP solution is to help clients to avoid this. To this end, an ROP solution provides two operations, PostGuard and Liberate, whose behaviour is described below. In addition, a client must be able to verify, under some circumstances, that a value is injail.

For the algorithms in this paper, the clients are threads and the values are pointers. A pointer that is injail has been allocated since it was last freed, and one that is free has not. Pointers that are escaping are due to be freed, but may not be freed yet because some threads may still use them. Threads use an ROP solution to prevent pointers from being freed when they might still be used.

The status (free, injail or escaping) of each value is determined as follows: Initially, all values are free. An application-dependent external Arrest(v) action can cause a free value v to become injail at any time. (In this paper, the Arrest(v) action corresponds to a call to **malloc** returning v.) A client can help a set S of injail values to escape by invoking Liberate(S), which causes each value in S to become escaping. Each invocation of Liberate returns a (possibly empty) set S' of escaping values, each of which becomes free at that time.

If a client wants to use a value v, it can prevent v from becoming free, by "posting a guard" g on v (by invoking PostGuard(g, v)) while v is injail.[1] A guard may be posted on at most one value; when the client subsequently invokes PostGuard(g, v') for some v', we say that g is *reposted on* v', and g is no longer posted on v (unless $v' = v$). The client can also invoke PostGuard(g,**null**)—where **null** is a special value not used elsewhere—to remove the guard from the previously guarded value without reposting the guard on a new value; in this case, we say the guard is *stood down*. If a value v is injail at any time that a guard g is posted on it, then g *traps* v from that time until g is stood down or reposted. An ROP solution must ensure that a value does not become free while it is trapped; that is, trapped values must not be returned by the Liberate operation.

If guard g is posted on value v too late—that is, if v is not injail when g is posted—g may fail to prevent v from becoming free. Therefore, to ensure that v does not become free while it is being used,[2] a client should first post a guard on v and then check that v is injail. If so, then it is safe to use the value until the guard is stood down or reposted.

In [3], we present an algorithm that solves ROP, except that it does not provide a method to verify that a value is injail. As discussed in that paper, it is often possible for a client to efficiently verify that a value it wants to use is injail by exploiting application-specific information. The details of how this is done in the algorithms we present in this paper are discussed with the algorithms themselves.

[1] In this paper, we ignore details of how guards are assigned to threads, simply assuming that a sufficient number of guards are statically assigned to each thread. For details of how guards are managed, see [3].

[2] In some cases, a thread p can determine independently of the ROP solution that a value it wants to use will remain injail while p uses the value. In this case, p can use the value without invoking ROP operations. An example of such an optimization is presented in Section 3.

3 Dynamic-sized Lockfree Queues

In this section, we present two dynamic-sized lockfree queue implementations based on a widely used lockfree queue algorithm by Michael and Scott [4]. 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 memory pool (Michael and Scott call it a freelist). Nodes in the memory pool may be reused by subsequent Enqueue operations, but they cannot be freed to the system. Thus, if the queue grows and then shrinks, the queue and its associated freelist together continue to consume as much space as it did at its maximum size. We show how to modify their algorithm to make truly dynamic-sized queues using standard dynamic allocation mechanisms and a solution to ROP, described in Section 2.

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, but 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 1). 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.

Michael and Scott's algorithm

The generic code in Figure 1 invokes four procedures, shown in italics in the figure, that are not specified in the generic code. We get the original M&S algorithm by instantiating these procedures with those shown in Figure 2. 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. Because M&S does not use an ROP solution, it has no notion of "guards", so *GuardedLoad* is an ordinary load and *Unguard* is a no-op.

We do not describe M&S in detail, nor argue that it is correct; see [4] for such details. Instead, we discuss below the aspects that are relevant for our purposes. 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 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 [4], 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 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

```
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(&Q→Tail, &tail, 0);
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(&Q→Head, &head, 0);
22      pointer_t tail = Q→Tail;
23      pointer_t next;
24      GuardedLoad(&head.ptr→next, &next, 1);
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;
}
```

Figure 1: Generic code for M&S.

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

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

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

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

Figure 2: 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(pointer_t *s, pointer_t *t, int g) {
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);
}
```

Figure 3: Auxiliary procedures for Algorithm 1. Code is shown for thread p.

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.

Algorithm 1

As mentioned earlier, Algorithm 1 eliminates the memory pool, and uses **malloc** and **free** directly for memory allocation. As discussed below, Algorithm 1 also eliminates the ABA problem, and thus, the need for version numbers. This allows Algorithm 1 to be used on systems that support CAS only on pointer-sized values.

Algorithm 1 is achieved by instantiating the generic code with the procedures shown in Figure 3. As explained below, these procedures use an ROP solution. In preparation for this use, it is assumed that two guards have been assigned to each thread p, and identifiers for these guards have been stored in `guards`[p][0] and `guards`[p][1], respectively. (See [3] for details of guard management.) 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. We later argue that the properties of ROP ensure that a node is never returned by an invocation of Liberate while some thread might still access that node.

The *GuardedLoad* procedure loads a value from the address specified by its first argument and stores the value loaded in the address specified by its second argument. The purpose of this procedure is to ensure that the value loaded is guarded by the guard specified by the third argument *before* the value is loaded. This is accomplished by a lockfree 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 stands down the specified guard.

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,[3] that, at the time of the access, the value has been continuously guarded since some point at which it was injail. As discussed earlier, if there is a path from either `Head` or `Tail` to a node, then the node is in the queue, and thus, it is injail. As shown below, we can exploit code already included in M&S, together with the specialization code in Figure 3, to detect the existence of such paths.

We first consider the access at line 9 of Figure 1. 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 3 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 access 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 1) 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 injail at some point since the guard was posted in the call to *GuardedLoad* at line 21, and this guard is not stood down 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 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.[4] This concludes our argument that Algorithm 1 never accesses freed memory.

Next, we argue that the version numbers for the node pointers are unnecessary in Algorithm 1. Apart from the overhead involved with managing these version numbers, the requirement that they are updated atomically with pointers renders algorithms that use them inapplicable in systems that support CAS only on pointer-sized values.

Eliminating version numbers in Algorithm 1

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

[3] As stated earlier, it is sometimes possible to determine that a node will not be freed before certain accesses without using an ROP solution. The accesses in lines 4 and 5 of Figure 1 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.

[4] 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.

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 some other value 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 3; 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 in the appendix). 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 injail at some point since the guard was posted when the comparison is performed (see Lemma 2 in the appendix). 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 stood down or reposted.

It remains to consider the comparison in line 9 of Figure 3, 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 second argument such that v was in the location pointed to by *GuardedLoad*'s first 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 stood down. 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.

Algorithm 2

One drawback of Algorithm 1 is that every Enqueue and Dequeue operation involves a call to the **malloc** or **free** library routine,[5] 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.

[5]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.

```
pointer_t Pool;

node_t *allocNode() {
1   pointer_t oldPool, newPool;
2   while (TRUE) {
3       GuardedLoad(&Pool, &oldPool, 0);
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;
    }
}
```

Figure 4: Revised *allocNode* and *deallocNode* procedures for Algorithm 2 (*GuardedLoad* and *Unguard* are unchanged from Figure 3).

Algorithm 2 is achieved by instantiating the generic code in Figure 1 with the same *GuardedLoad* and *Unguard* procedures used for Algorithm 1 (see Figure 3) and the *allocNode* and *deallocNode* procedures shown in Figure 4. 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 4 use a linked list representation of a stack for a memory pool. This implementation extends Treiber's straightforward implementation [6] 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 lockfree 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 an ROP solution 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.

4 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 lockfree FIFO queue is negligible, provided contention for the queue is not too high, and that it is modest even under high contention. 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 400MHz UltraSPARC® II processors and 16GB of memory, and a Sun E10000 with 64 466MHz UltraSPARC® II processors and 64GB 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 (PTB) [3] to solve the Repeat Offender Problem. One configuration was achieved by instantiating the generic algorithm in Figure 1 with the procedures shown in Figure 3; 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 4. 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 (for example, 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,[6] 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

[6] To facilitate this check, we modified the code of Figure 4 to include a count field in the header node of the memory pool.

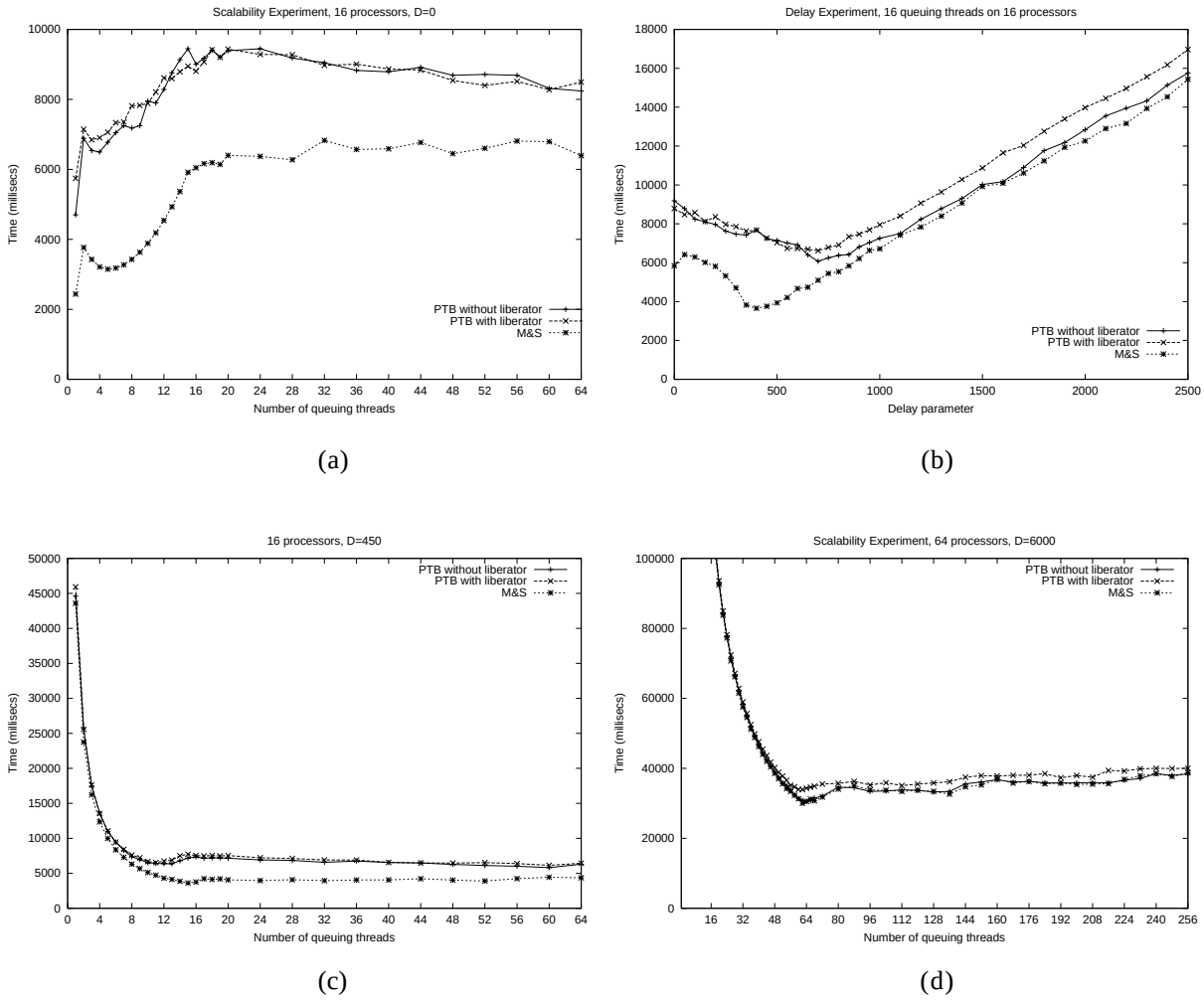

Figure 5: Performance experiments

in Figure 5(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 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.

The results discussed above compare the algorithms under maximum contention, but it is more realistic to model some non-queue activity between each queue operation. Figure 5(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[7] 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 non-zero delay parameters. We chose $D = 450$ (where Figure 5(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 5(c). These results are more in line with simple intution 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 5(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 5(d). (The counterpart experiment on the 16-processor machine with $D = 900$ yielded qualitatively similar results, with the curve bottoming out at about 6000ms.) 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 a particular queue should usually be low because typical applications will do other work between queue operations.

5 Single-word Lockfree Reference Counting (SLFRC)

In Section 3, we showed how to use an ROP solution to make a lockfree 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 lockfree reference counting* (SLFRC), a technique that allows us to transform many lockfree data structure implementations that assume garbage collec-

[7] Anomolous 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 dequeing and for choosing the number of delay loop iterations.

tion (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 *lockfree reference counting* (LFRC) [1]. 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 [1] first.

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 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.

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. 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 6, are discussed below.

SLFRC avoids the need for DCAS by using an ROP solution to coordinate access to an object's reference count. To accommodate this change, the SLFRCDestroy operation must be modified

```
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);
}
```

Figure 6: Code for SLFRCDestroy and SLFRCLoad. Code for add_to_rc is repeated from [1] for convenience. Code is shown for thread p; g_p is a guard owned by p (see [3] for details of guard management).

slightly from the LFRCDestroy operation used in [1]. 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 pointer values to be passed to Liberate, rather than freeing them directly, as was the case in LFRC. When a value has been returned by Liberate, it can be freed. One way to achieve this, which we adopt in Figure 6, is to have SLFRCDestroy invoke Liberate directly, passing as a parameter the singleton set containing the pointer to be freed, and to then free 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 an ROP solution. Specifically, at line 15, we post a guard on the value read previously. As explained in Section 2, this is not sufficient to prevent the object from being freed before its reference count is accessed: we must ensure that the value being guarded is still (or again) injail *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 lockfreedom 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 after all pointers to them have been destroyed). This is an example of using application-specific information to efficiently verify that a value is injail (the load at line 16 implements this operation).

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 lockfree 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 stand down 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.

SLFRC can easily be applied to the widely used lockfree FIFO queue algorithm of Michael and Scott [4] to remove its dependence on an explicit object pool of queue nodes that can never be freed.[8]

6 Concluding Remarks

We recently posed and solved the Repeat Offender Problem. In this paper, we have demonstrated the utility of solutions to the ROP through example applications concerning lockfree data structures. In this context, ROP solutions allow application threads to ensure that objects are not deallocated while pointers to them still exist, a central difficulty in designing dynamic-sized lockfree data structures.

Specifically, we have presented several variations on Michael and Scott's widely used lockfree FIFO queue algorithm, each of which has the ability to release storage no longer required for the queue. As far as we are aware, these are the first dynamic-sized lockfree data structures that can continue to reclaim memory even if some threads fail. We have also applied our new techniques to achieve SLFRC (single-word lockfree reference counting)—a methodology for transforming GC-dependent applications to equivalent ones that do not depend on GC. This result improves upon the LFRC methodology we presented recently by removing its dependence on the double-compare-and-swap instruction, which is not widely supported.

When specifying the ROP, we paid particular attention to allowing much of the work to be scheduled separately from application work. Future research includes experimenting with methods of doing this scheduling in various application and system configurations, as well as applying ROP solutions to the design of other lockfree data structures.

The ideas in this paper came directly from insights gained and questions raised in our work on LFRC [1]. This demonstrates the value of research that assumes stronger synchronization primitives than are currently widely supported.

Maged Michael has recently presented a solution to a problem very similar to ROP and has also shown how to implement various dynamic-sized lockfree data structures using his approach. In [3], we compare the approaches.

[8] Because the Michael and Scott 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.

References

[1] D. Detlefs, P. Martin, M. Moir, and G. Steele. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.

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

[3] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized lockfree data structures. Technical Report TR-2002-112, Sun Microsystems Laboratories, 2002.

[4] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[5] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.

[6] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.

[7] J. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–22, 1995. See `http://www.cs.sunysb.edu/~valois` for errata.

A Proof Sketches for Lemmas of Section 3

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 `Head` or `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 12) and the initialization to **null** (line 1). 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 3, 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 stood down before the comparison. Furthermore, this value is injail at some time at or after t and before the comparison occurs (which implies that the value is trapped when the comparison occurs).

Proof sketch: First, we observe that M&S has the property that a node is never deallocated while a path from either `Head` or `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, which implies that the value is injail. 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 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 `Tail` and `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. □

About the Authors

Maurice Herlihy is a professor at the Brown University Computer Science Department. Before joining Brown in 1993, he was a member of research staff at Digital's Cambridge Research Laboratory, and before that a faculty member at Carnegie Mellon University's School of Computer Science. His interests include practical and theoretical aspects of distributed systems. He received a Ph.D. in Computer Science from MIT, and an A.B. in Mathematics from Harvard.

Victor Luchangco works in the Scalable Synchronization Group of Sun Microsystems Laboratories. His research focuses on algorithms and mechanisms to support concurrent programming on large-scale distributed systems. Although he is a theoretician by disposition and training, he is also interested in practical aspects of computing; he would like to design mechanisms that people will actually use. He also wants to explore how to make proofs for concurrent systems easier, both by changing how people design these systems and by using tools to aid in formal verification. He received an Sc.D. in Computer Science from MIT in 2001, with a dissertation on models for weakly consistent memories.

Paul Martin joined Sun Microsystems Laboratories in 1993 as founding PI and later co-PI of the Speech Applications project. He is currently a member of the Knowledge Technology Group, adapting and extending natural language and knowledge representation tools. Additionally, Paul works with the Scalable Synchronization Group, exploring lock-free interaction for concurrent processes. Before joining Sun, Paul pursued his research interest in using human language to communicate with computers in IBM's Austin Information Retrieval Tools group, in the Human Interface and Natural Language groups at MCC, and at SRI's Artificial Intelligence Center. He received his Ph.D. from Stanford University for research conducted at Stanford's AI Lab and at Xerox PARC, and his B.S. from North Carolina State in CS and EE.

Mark Moir received the B.Sc. (Hons.) degree in Computer Science from Victoria University of Wellington, New Zealand in 1988, and the Ph.D. degree in Computer Science from the University of North Carolina at Chapel Hill, USA in 1996. From August 1996 until June 2000, he was an assistant professor in the Department of Computer Science at the University of Pittsburgh. In June 2000, he joined Sun Microsystems Laboratories, where he is now the Principal Investigator of the Scalable Synchronization Research Group.

Dr. Moir's main research interests concern practical and theoretical aspects of concurrent, distributed, and real-time computing. His current research focuses on mechanisms for non-blocking synchronization in shared-memory multiprocessors.