Bringing Practical Lock-Free Synchronization to 64-Bit Applications

Simon Doherty
School of Mathematical and Computing Sciences
Victoria University
Wellington, New Zealand
simon.doherty@mcs.vuw.ac.nz

Maurice Herlihy
Department of Computer Science
Brown University
Providence, RI 02912, USA
mph@cs.brown.edu

Victor Luchangco
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
victor.luchangco@sun.com

Mark Moir
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803, USA
mark.moir@sun.com

ABSTRACT

Many lock-free data structures in the literature exploit techniques that are possible only because state-of-the-art 64-bit processors are still running 32-bit operating systems and applications. As software catches up to hardware, "64-bit-clean" lock-free data structures, which cannot use such techniques, are needed.

We present several 64-bit-clean lock-free implementations: *load-linked/store-conditional* variables of arbitrary size, a FIFO queue, and a freelist. In addition to being portable to 64-bit software, our implementations also improve on previous ones in that they are space-adaptive and do not require knowledge of the number of threads that will access them.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.2 [**Operating Systems**]: Storage Management; E.1 [**Data**]: Data Structures

General Terms

Algorithms, Theory

Keywords

Multiprocessors, 64-bit architectures, 64-bit-clean software, nonblocking synchronization, lock-free, memory management, population-oblivious, space-adaptive, compare-and-swap (CAS), load-linked/store-conditional (LL/SC), queues, freelists

1. INTRODUCTION

For more than a decade, 64-bit architectures have been available [14, 16, 18, 21]. These architectures support 64-bit addresses, allowing direct access to huge virtual address spaces [3]. They also support atomic access to 64-bit quantities using synchronization primitives such as compare-and-swap (CAS) and the load-linked/store conditional (LL/SC) pair, which provide powerful tools for implementing lock-free data structures.

Predictably, operating systems and application software that exploit these 64-bit capabilities have been slower to emerge. Thus, many important 32-bit operating systems and applications are still in common use, and most 64-bit architectures support them. As a result, for a period of several years, techniques that use 64-bit synchronization primitives to atomically manipulate 32-bit pointers together with other information, such as version numbers, have been widely applicable. Many practical lock-free data structures exploit such techniques (e.g., [13, 19]).

The increasing prevalence of 64-bit operating systems and applications (in which pointers are 64 bits) signals the end of this convenient era: *64-bit-clean* lock-free data structures that do not require synchronization primitives that can atomically manipulate a pointer and a version number are increasingly important.

We present 64-bit-clean[1] implementations of several important lock-free data structures: arbitrary-size variables supporting LL and SC operations, FIFO queues, and freelists.

Our implementations are based on 64-bit CAS, but it is straightforward to modify them for use in architectures that support LL/SC instead of CAS [15]. Our LL/SC implementation is useful even in architectures that provide LL/SC in hardware, because it eliminates numerous restrictions on the size of variables accessed by LL/SC and the way in which they are used. For example, in some architectures, the program must perform only register operations between an LL and the following SC; no such restriction is imposed by our implementation. Our results therefore will help programmers to develop portable code, because they can ignore the different restrictions imposed by different architectures on the use of

[1]Our techniques target any architecture that can perform CAS on a pointer-sized variable (of at least 64 bits); for concreteness, we present our techniques assuming that this size is 64 bits.

PODC'04, July 25–28, 2004, St. John's, Newfoundland, Canada.

ACM 1-58113-802-4/04/0007.

LL/SC. Furthermore, our implementations are portable between 32-bit and 64-bit applications, while many previous lock-free data structure implementations are not.

The only previous 64-bit-clean CAS-based implementation of LL/SC is due to Jayanti and Petrovic [10]. While their implementation is wait-free [5], it requires $O(mn)$ space, where m is the number of variables that support LL/SC and n is the number of threads that can access those variables; ours uses only $O(m+n)$ space. Furthermore, the implementation in [10] requires *a priori* knowledge of a fixed upper bound on the number of threads that will access an LL/SC variable. We call such implementations *population-aware* [8]. In contrast, our implementation has no such requirement; it is *population-oblivious*. However, our implementation guarantees only lock-freedom, a weaker progress guarantee than wait-freedom, but one that is generally considered adequate in practice.

Our lock-free FIFO queue implementation is the first that is 64-bit-clean, population-oblivious and *space-adaptive* (i.e., its space consumption depends only on the number of items in the queue and the number of threads currently accessing the queue). Previous lock-free FIFO queue implementations have at most one of these advantages.

A freelist manages memory resources and can be used to avoid the cost of using a general malloc/free implementation for this purpose. Previous lock-free freelists [19] are not 64-bit-clean, and can be prevented by a single thread failure from ever freeing memory blocks to the system. Ours overcomes both of these problems.

We provide some background in Section 2, and present our LL/SC implementation in detail in Section 3. In Sections 4 and 5, we explain how to adapt the techniques used in the LL/SC implementation to achieve our queue and freelist implementations We conclude in Section 6.

2. BACKGROUND

A data structure implementation is *linearizable* [9] if for each operation, there is some point—called the *linearization point*—during the execution of that operation at which the operation appears to have taken place atomically. It is *lock-free* [5] if it guarantees that after a finite number of steps of any operation on the data structure, *some* operation completes. It is *population-oblivious* [8] if it does not depend on the number of threads that will access the data structure. Finally, it is *space-adaptive* if at all times, the space used is proportional to the size of the abstract data structure (and the number of threads currently accessing it).[2]

The CAS operation, defined in Figure 1, takes the address of a memory location, an expected value, and a new value. If the location contains the expected value, then the CAS atomically stores the new value into the location and returns *true*. Otherwise, the contents of the location remain unchanged, and the CAS returns *false*. We say that the CAS *succeeds* if it returns *true*, and that it *fails* if it returns *false*.

A typical way to use CAS is to read a value—call it A—from a location, and to then use CAS to attempt to change the location from A to a new value. The intent is often to ensure that the CAS succeeds only if the location's value does not change between the read and the CAS. However, the location might change to a different value B and then back

[2]There are other variants of space adaptivity [8], but this simple definition suffices for this paper.

```
bool CAS(a, e, n) {
    atomically {
        if (*a == e) {
            *a = n;
            return true;
        } else
            return false;
}
```

Figure 1: The CAS operation.

to A again between the read and the CAS, in which case the CAS can succeed. This phenomenon is known as the *ABA problem* [17] and is a common source of bugs in CAS-based algorithms. The problem can be avoided by storing the variable being accessed together with a version number in a CAS-able word: the version number is incremented with each modification of the variable, eliminating the ABA problem (at least in practice; see [15] for more detail). However, if the variable being modified is a 64-bit pointer, then this technique cannot be used in architectures that can perform CAS only on 64-bit variables. A key contribution of this paper is a novel solution to the ABA problem that can be used in such architectures.

The LL and SC operations are used in pairs: An SC operation is *matched* with the preceding LL operation by the same thread to the same variable; there must be such an LL operation for each SC operation, and no LL operation may match more than one SC operation. LL loads the value of a location, and SC conditionally stores a value to a location, succeeding (returning *true*) if and only if no other store to the location has occurred since the matching LL.[3] Thus the ABA problem does not arise when using LL and SC—instead of read and CAS—to modify a variable.

For simplicity, we require every LL to be matched. If a thread decides not to invoke a matching SC for a previous LL, it instead invokes UNLINK, which has no semantic effect on the variable. An LL operation is said to be *outstanding* from its linearization point until its matching SC or UNLINK returns. It is straightforward to eliminate the need for an explicit thread-visible UNLINK operation by having LL invoke UNLINK whenever a previous LL operation by the same thread to the same location is already outstanding.

Related work

Anderson and Moir [2] describe a wait-free implementation of a multiword LL/SC that requires $O(mn^2)$ space, where m is the number of variables and n is the number of threads that may access the variables; Jayanti and Petrovic present another that uses $O(mn)$ space [10]. These algorithms are impractical for applications that require LL/SC on many variables. In addition, they are not population-oblivious; they require n to be known in advance, a significant drawback for applications with dynamic threads. Moir [15] presents a lock-free algorithm that uses only $O(m)$ space, but his algorithm is not 64-bit-clean.

Our queue algorithm is similar to the algorithms of Valois [20] and of Michael and Scott [13]. However, those algorithms are not 64-bit-clean. Furthermore, they cannot free nodes removed from the queue for general reuse (though the

[3]We describe the *ideal* LL/SC semantics here. Hardware LL/SC implementations are usually weaker; in particular, they allow SC to fail even in the absence of an intervening store [15].

nodes can be reused by subsequent enqueue operations), and are therefore not space adaptive.

Herlihy et al. [7] and Michael [11] independently proposed general techniques to enable memory to be freed from lock-free data structures, and they applied these techniques to the Michael-and-Scott queue [6, 11]. However, the resulting algorithms are not population-oblivious. Although they can be made population-oblivious [8], the resulting solutions are still not space adaptive; in the worst case, they require space proportional to the number of all threads that ever access the queue.

Treiber [19] provides two freelist implementations, neither of which is 64-bit-clean. Furthermore, the first does not provide operations for expanding and contracting the freelist. Modifying the implementation to expand the freelist is straightforward, but enabling contraction is not. Treiber's second freelist implementation does allow contraction, but a single delayed thread can prevent all contraction. In contrast, our freelist implementation is 64-bit-clean and does not allow thread delays to prevent other threads from contracting the freelist.

3. OUR LL/SC IMPLEMENTATION

With unbounded memory, it is straightforward to implement a lock-free, population-oblivious, arbitrary-sized LL/SC variable using the standard *pointer-swinging* technique: We store values in contiguous regions of memory called *nodes* and maintain a pointer to the *current node*. LL simply reads the pointer to the current node and returns the contents of the node it points to. SC allocates a new node, initializes it with the value to be stored, and then uses CAS to attempt to "swing" the pointer from the previously current node to the new one; the SC succeeds if and only if the CAS succeeds. If every SC uses a new node, then the CAS in an SC succeeds if and only if there is no change to the pointer between the CAS and the read in the preceding LL. This technique is well-known and used in systems that use garbage collection to provide the illusion of unbounded memory (see the JSR-166 library [1], for example).

Our implementation builds on this simple idea, but is complicated by the need to free and reuse nodes in order to bound memory consumption. Reclaiming nodes too late results in excessive space overhead. However, reclaiming them too soon leads to other problems. First, an LL reading the contents of a node might in fact read part or all of a value stored by an SC that is reusing the node. Second, the CAS might succeed despite changes since the previous read because of the recycling of a node: the ABA problem. Our implementation maintains additional information that allows nodes to be reclaimed promptly enough to bound its space complexity, but avoids the problems above by preventing premature reclamation.

We first describe the basic implementation assuming that nodes are never reclaimed prematurely and ignoring those parts related only to node reclamation (Section 3.1). Then we describe how nodes are reclaimed, argue why they are never reclaimed prematurely, and analyze the space complexity of the algorithm (Section 3.2). To simplify the code and discussion, the implementation presented here restricts threads to have at most one outstanding LL operation at a time. However, extending it to allow each thread to have multiple outstanding LL operations is straightforward.

```
typedef struct {
    Node *ptr0, *ptr1;
    EntryTag entry;
} LLSCvar;

typedef struct {
    int ver;
    int count;
} EntryTag;

typedef struct {
    Data d;
    Node *pred;
    ExitTag exit;
} Node;

typedef struct {
    int count;
    bool nlC;
    bool nlP;
} ExitTag;
```

Figure 2: Data types used in the LL/SC algorithm. The EntryTag and ExitTag types fit into 64 bits, so can be atomically accessed using cas.

3.1 The basic implementation

Rather than storing a pointer to the current node in a single location, we alternate between two locations, `ptr0` and `ptr1`. At any time, one of these pointers is the *current pointer*—it points to the current node—and the other is the *noncurrent pointer*. Which pointer is current is determined by a version number `entry.ver`:[4] if `entry.ver` is even, then `ptr0` is the current pointer; otherwise `ptr1` is.

An LL operation determines the current node and returns the data value it contains. An SC operation attempts to change the noncurrent pointer to point to a new node—initialized with the data value to be stored—and then increment `entry.ver`, making this pointer current. If the SC successfully installs the new pointer but is delayed before incrementing the version number, then another thread can "help" by incrementing the version number on its behalf. The successful SC is linearized at the point at which the version number is incremented (either by the thread executing that SC or by a helping thread), causing the newly installed node to become current.

Our algorithm guarantees the following *alternating property*: In any execution, the sequence of events that modify `ptr0` and `ptr1` and `entry.ver` strictly alternates between

- modifying the noncurrent pointer to point to the new node of an SC operation; and
- incrementing `entry.ver`, thereby causing the current pointer to become noncurrent and vice versa.

With this property, it is easy to see that the algorithm described above provides the correct semantics: neither `ptr0` nor `ptr1` ever changes while it is the current pointer; the noncurrent pointer is changed exactly once (by a successful SC operation) between consecutive increments of `entry.ver`; and each time we increment the version number, and therefore linearize a successful SC (the unique SC that changed the noncurrent pointer since the previous time the version number was incremented), the new node installed by the successful SC becomes the current node.

Figure 2 shows the types used in our implementation. An **LLSCvar** structure consists of three fields, `ptr0`, `ptr1` and

[4]In addition to determining which pointer is current, the version number eliminates the ABA problem in practice, provided the version number has enough bits to ensure that it does not repeat a value during the interval in which some thread executes a short code sequence. In our algorithm, we can easily allocate 32 or more bits to the version number, which we believe is sufficient. For further discussion of the number of bits required to avoid the ABA problem, see [15].

ptr0→d = d_0
ptr0→pred = ptr1
ptr0→exit = ⟨0, *false*, *false*⟩
ptr1→exit = ⟨0, *true*, *false*⟩
entry.ver = 0
entry.count = 0

Figure 3: Initial state of an LL/SC location, where d_0 is the initial value of the location.

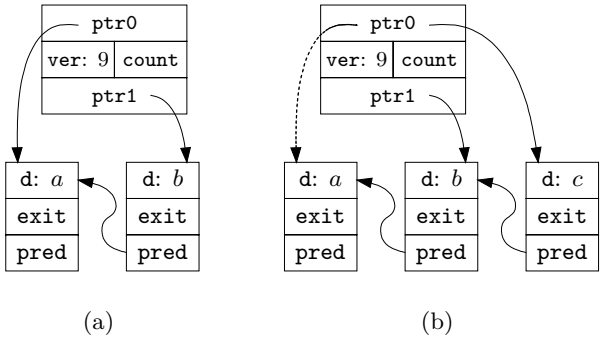

Figure 4: Two states of the LL/SC implementation. Dotted pointer indicates previous value (see text).

entry, each of which is 64 bits (so the CAS operation can be applied to each field, though not to the entire **LLSCvar** structure). In addition to the fields already mentioned, the entry field of an LL/SC variable has a count field, and each node has pred and exit fields. The pred field of each node contains a pointer to the node that was current immediately before this node. The other fields are concerned only with node reclamation, and are discussed later.

Figure 3 shows how an LL/SC variable is initialized and Figure 4 illustrates two classes of states of our algorithm. In both illustrations, entry.ver is odd, so ptr1 is the current pointer and ptr0 is the noncurrent pointer. In Figure 4(a), the noncurrent pointer points to the current node's predecessor (i.e., the node that was current before the node that is current in the figure). In Figure 4(b), the noncurrent pointer points to a new node whose pred field points to the current node. From a state like the one in Figure 4(a), installing a pointer to a new node whose pred field points to the current node into the noncurrent pointer results in a state like the one in Figure 4(b). Furthermore, from a state like the one in Figure 4(b), incrementing entry.ver results in a state like the one in Figure 4(a), because incrementing entry.ver changes its parity, thereby reversing the roles of ptr0 and ptr1. The key to understanding our algorithm is to notice that it alternates between states like that in Figure 4(a) and states like that in Figure 4(b). This behavior is captured by the alternating property, which is central to the correctness proof for our algorithm.

We now present our algorithm in more detail and explain how it preserves the alternating property; we ignore for now details related to node reclamation. Pseudocode for the LL, SC and UNLINK operations is presented in Figure 5. Each thread has two persistent local variables, mynode and myver, which are set by the LL operation, and retain their values while that LL is outstanding. The CURRENT and NONCURADDR macros determine the current and noncurrent pointers based on the ptr0 or ptr1 fields and the entry.ver fields, as explained above. Specifically, if loc→entry.ver == version, then CURRENT(loc,version) gives the current pointer of loc, and NONCURADDR(loc,version) gives the address of the noncurrent pointer. The RELEASE and TRANSFER procedures, the entry.count field, and the exit field and its initialization value INIT_EXIT are relevant only to node reclamation, as are the effects of UNLINK. We defer further discussion of these procedures and fields until Section 3.2.

Ignoring for now the effect of the CAS at line L5 on the entry.count field, we see that a thread p executing LL records entry.ver in its persistent local myver variable and the current node indicated by this value in its mynode variable. To ensure a consistent view of the current node and version number, LL retries if entry.ver changes while it determines the current node (lines L2 and L5). The LL operation is linearized at the (unique) point at which p successfully executes the CAS at line L5.

To execute an SC operation, p allocates and initializes a new node[5] with the value to be stored, and stores the node observed as current by the previous LL in the node's pred field (lines S1 and S2). Then, p uses CAS to attempt to change the noncurrent pointer to point to the new node (line S4). We do not simply read the contents of the noncurrent pointer in order to determine the expected value for this CAS. If we did, two different SC operations could install new nodes in the noncurrent pointer, without the noncurrent pointer becoming current as the result of an increment of entry.ver. Such behavior would violate the alternating property.

To avoid this problem, we instead determine the expected value for the CAS by reading the pred field of the node observed as current (line S3). Recall that when a node becomes current, its pred field points to the node that was current immediately before it. Thus, the pred field of the current node is the same as the noncurrent pointer before a new node is installed. Once an SC has successfully changed the noncurrent pointer to point to a new node, no other SC can do so again before entry.ver is incremented. This could happen only if some thread previously saw the newly installed node as the predecessor of some node. As we explain later, our node reclamation technique precludes this possibility.

After the execution of line S4, either p's SC has succeeded in changing the noncurrent pointer, or some other SC has. In either case, the entry.ver field should now be incremented in order to make the successful SC that installed a new node take effect. The CAS at line S7 ensures that the version number is incremented. (The loop at lines S6 through S8 is necessary because the CAS at line S7 may fail for reasons other than another thread having incremented entry.ver; this possibility is explained below.)

3.2 Memory reclamation

If nodes are never reclaimed, then values stored to ptr0 and ptr1 are all distinct, and it is easy to see the correctness of the algorithm as described. We now explain how our implementation reclaims and reuses nodes and why the algorithm is correct despite this. For simplicity, we defer consideration of UNLINK until later in this section; for now, we assume that every LL is matched by an SC.

After an LL successfully executes the CAS at line L5, it reads the contents of the node it determined to be current at lines L6 and S3. We ensure that the node is not reclaimed before this happens. Specifically, after a thread successfully executes the CAS at line L5, we ensure that the node is not

[5]If no suitable lock-free memory allocator is available, then nodes can be allocated from a freelist. The implications of this approach are described in Section 3.3.

```
Macros:
  CURRENT(loc, ver) ≡ (ver%2 == 0 ? loc→ptr0 : loc→ptr1)
  NONCURADDR(loc, ver) ≡ (ver%2 == 0 ? &loc→ptr1 : &loc→ptr0)
  INIT_EXIT ≡ ⟨0, false, false⟩

    Data LL(LLSCvar *loc) {
L1.   do {
L2.       EntryTag e = loc→entry;
L3.       myver = e.ver;
L4.       mynode = CURRENT(loc, e.ver);
L5.   } while (!CAS(&loc→entry, e, ⟨e.ver, e.count + 1⟩));
L6.   return mynode→d;
    }

    void UNLINK(LLSCvar *loc) {
U1.   while ((e = loc→entry).ver == myver)
U2.       if (CAS(&loc→entry, e, ⟨e.ver, e.count − 1⟩)) return;
U3.   RELEASE(mynode);
    }

    bool SC(LLSCvar *loc, Data newd) {
S1.   Node *new_nd = alloc(Node);
S2.   new_nd→d = newd;
      new_nd→pred = mynode;
      new_nd→exit = INIT_EXIT;
S3.   Node *pred_nd = mynode→pred;
S4.   success = CAS(NONCURADDR(loc, myver), pred_nd, new_nd);
S5.   if (!success) free(new_nd);
S6.   while ((e = loc→entry).ver == myver)
S7.       if (CAS(&loc→entry, e, ⟨e.ver + 1, 0⟩))
S8.           TRANSFER(mynode, e.count);
S9.   RELEASE(mynode);
S10.  return success;
    }
```

Figure 5: The LL, SC, and UNLINK operations.

```
Macros:
  CLEAN(exit) ≡ (exit.count == 0 ∧ pre.nlC)
  FREEABLE(exit) ≡ (CLEAN(exit) ∧ exit.nlP)

    void RELEASE(Node *nd) {
R1.   Node *pred_nd = nd→pred;
R2.   do {
R3.       ExitTag pre = nd→exit;
R4.       ExitTag post = ⟨pre.count − 1, pre.nlC, pre.nlP⟩;
R5.   } while (!CAS(&nd→exit, pre, post));
R6.   if (CLEAN(post)) SETNLPRED(pred_nd);
R7.   if (FREEABLE(post)) free(nd);
    }

    void TRANSFER(Node *nd, int count) {
T1.   do {
T2.       ExitTag pre = nd→exit;
T3.       ExitTag post = ⟨pre.count + count, true, pre.nlP⟩;
T4.   } while (!CAS(&nd→exit, pre, post));
    }

    void SETNLPRED(Node *pred_nd) {
P1.   do {
P2.       ExitTag pre = pred_nd→exit;
P3.       ExitTag post = ⟨pre.count, pre.nlC, true⟩;
P4.   } while (!CAS(&pred_nd→exit, pre, post));
P5.   if (FREEABLE(post)) free(pred_nd);
    }
```

Figure 6: Helper procedures for the LL/SC implementation.

reclaimed before that thread invokes RELEASE on that node at line S9. Also, to avoid the ABA problem, we ensure that a node is not reclaimed if some thread might still see it as the predecessor of another node (at line S3), and therefore use it as the expected value for the CAS at line S4.

We avoid both premature reclamation scenarios by recording information in `entry.count` and the `exit` field of each node that allows us to determine when it is safe to reclaim a node. First, we use `entry.count` to count the number of threads that successfully execute the CAS at line L5 while `entry.ver` contains a particular value. (Note that `entry.count` is reset to zero whenever `entry.ver` is incremented at line S7.) When a thread increments `entry.count`, we say the thread *pins* the node that is current at that time.

One might think that we could maintain an accurate count of the number of threads that have pinned a node and not subsequently released it by simply decrementing `entry.count` in RELEASE. However, this approach does not work because by the time a thread invokes RELEASE for a particular node, that node is no longer current, so `entry.count` is being used for a different node—the one that is now current. Therefore, we instead use a node's `exit.count` field to count the number of threads that have released the node; this counter starts at zero and is decremented by each releasing thread (see lines R4 and R5 in Figure 6).

We use the TRANSFER procedure to reconcile the number of threads that pinned the node with the number that have since released it. TRANSFER adds the value of `entry.count` when a node is replaced as the current node to that node's `exit.count` field (lines S7, S8, and T1 through T4). When `exit.count` contains zero after this transfer has happened, all threads that pinned this node have since released it.

To distinguish the initial zero state of the `exit.count` field from the state in which `entry.count` has been transferred and all threads have executed RELEASE, we use a flag `nlC` in the node's `exit` field; TRANSFER sets `exit.nlC` (see line T3) to indicate that the transfer has occurred (`nlC` stands for "no longer current"; TRANSFER is invoked by the thread that makes the node noncurrent). We say that a node with `exit.nlC` set and `exit.count` == 0 is *clean* (as captured by the `CLEAN` macro).

For the UNLINK operation, a thread could simply invoke RELEASE, as on line U3. However, if `entry.ver` has not changed since the thread pinned a node, we can instead decrement `entry.count` (see lines U1 and U2); it is still being used to keep track of the number of threads that pinned the node pinned by the thread that invoked UNLINK.

In our algorithm as described so far, no thread accesses a clean node. However, it is not always safe to free a clean node: recall that we must also prevent a node from being reclaimed while a thread might still determine it to be the predecessor of another node. For this purpose, we use one more flag in the `exit` field called `nlP` (for "no longer predecessor"). At any time, each node is the predecessor of only one node, so we simply need to determine when that node's `pred` field will no longer be accessed by any thread, that is, when that node is clean. A thread that makes a node clean invokes the SETNLPRED procedure to set the `nlP` flag of the

node's predecessor (line R6). When a node is clean *and* has its `exit.nlP` flag set, as expressed by the `FREEABLE` macro, it is safe to free the node (lines R7 and P5).

Let us analyze the space requirements for an application using our implementation for LL/SC variables. Each variable requires $O(1)$ space for its **LLSCvar** structure, and has two nodes that cannot be reclaimed (the nodes pointed to by its `ptr0` and `ptr1` fields). In addition, each LL/SC sequence in progress can prevent the reclamation of three nodes: the node pinned by the thread between an LL operation and its matching SC or UNLINK, the predecessor of the pinned node, and the new node used by an SC operation. Thus, in an application with m LL/SC variables, the space used by our algorithm at any time is $O(m + k)$, where k is the number of outstanding LL operations *at that time*. In the worst case, when all n threads have outstanding LL operations, the space used is $O(m + n)$. Note that this space complexity is asymptotically optimal, and that the space used adapts to the number of threads actually accessing the LL/SC variables at any time. In particular, only $O(m)$ space is needed when no threads are accessing these variables. The only previous 64-bit-clean implementation [10] always uses $O(mn)$ space, a clear limitation in practice. Furthermore, it requires *a priori* knowledge of n; our algorithm does not.

3.3 Optimizations and Extensions

Our LL/SC implementation can be made more efficient by observing that if `FREEABLE(post)` holds before the CAS on line R5 or line P4, then the CAS does not need to be executed; `mynode` can simply be freed because there are no threads that still have to RELEASE this node. Similarly, a thread that calls TRANSFER at line S8 will always subsequently call RELEASE at line S9. Therefore, we can combine the effect of the two CASes in those two procedures into a single CAS.

It is easy to extend our implementation to allow threads to have multiple outstanding LL operations: each thread simply maintains separate `mynode` and `myver` local variables for each outstanding LL. In the resulting extension, a thread may pin several nodes simultaneously (one for each outstanding LL). The space complexity of this extension is still $O(m + k)$, but now there may be more outstanding LL operations than threads (i.e., we may have $k > n$). In the unlikely case that all n threads *simultaneously* have outstanding LL operations on all m variables, then $O(mn)$ space is used. However, this much space is used only while $O(mn)$ LL operations are outstanding. As before, if no threads are accessing the LL/SC variables, then the space consumed is $O(m)$.

We can also extend our implementation to provide an operation that "validates" the previous LL, that is, determines whether its future matching SC can still succeed. A `validate` operation simply determines whether the noncurrent pointer still points to the predecessor of the node stored in `mynode` by the LL operation. If so, a future SC can replace it with a new node, thereby ensuring its success.

If our algorithm is used with a memory allocator that is not lock-free, then neither is our LL/SC implementation. While lock-free allocators exist [4, 12], most standard allocators are not lock-free. An alternative means for achieving a lock-free implementation is to use a lock-free freelist to manage nodes. (We present a suitable freelist implementation in Section 5.) The idea is to populate the freelist with enough nodes that one is always available for an SC operation to use. The number of nodes needed depends on the number

```
     void ENQUEUE(Value v) {
E1.      Node *nd = alloc(Node);
E2.      nd→v = v;
         nd→next = null;
         nd→exit = INIT_EXIT;
E3.      while (true) {
E4.          Node *tail = LL(&Tail);
E5.          nd→pred = tail;
E6.          if (CAS(&tail→next, null, nd)) {
E7.              SC(&Tail, nd);
E8.              return;
E9.          } else
E10.             SC(&Tail, tail→next);
         }
     }

     Value DEQUEUE() {
D1.      while (true) {
D2.          Node *head = LL(&Head);
D3.          Node *next = head→next;
D4.          if (next == null) {
D5.              UNLINK(&Head);
D6.              return null;
             }
D7.          if (SC(&Head, next)) {
D8.              Value v = next→v;
D9.              SETTOBEFREED(next);
D10.             return v;
             }
         }
     }
```

Figure 7: Queue operations.

of threads that simultaneously access the implemented variable. If we cannot bound this number in advance, we can resort to the standard memory allocator to increase the size of the freelist upon thread creation, and remove nodes from the freelist and free them upon thread destruction. While this approach involves locking when creating or destroying a thread, we avoid locking during the lifetime of each thread.

4. QUEUE

In this section, we describe a 64-bit-clean lock-free FIFO queue implementation that is population-oblivious and consumes space proportional only to the number of items in the queue (and the number of threads currently accessing the queue). Our queue implementation is similar in structure to that of Michael and Scott [13], but overcomes two important drawbacks. First, the implementation of [13] uses version numbers on its `Head` and `Tail` pointers, and is therefore not 64-bit-clean. Second, it cannot free nodes that have been dequeued; instead it stores them in a freelist for subsequent reuse, resulting in space consumption proportional to the historical maximum size of the queue.

Figure 7 presents our queue code. Rather than modifying the `Head` and `Tail` pointers with CAS and using version numbers to avoid the ABA problem (as in [13]), we use LL and SC. If we ignore memory management issues for a moment, and assume that the LL and SC operations used are the standard hardware-supported ones, then this implementation is essentially the one in [13]. To facilitate the memory management required to achieve a 64-bit-clean space-adaptive implementation, we use LL and SC operations similar to those presented in the previous section in place of the standard operations.

```
typedef struct {
    Value v;
    Node *next;
    Node *pred;
    ExitTag exit;
} Node;

typedef struct {
    int count;
    int transfersLeft;
    bool nlP;
    bool toBeFreed;
} ExitTag;
```

INIT_EXIT $\equiv \langle 0, 2, \mathit{false}, \mathit{false} \rangle$
CLEAN(exit) $\equiv$ (exit.count $== 0 \wedge$ exit.transfersLeft $== 0$)
FREEABLE(exit) $\equiv$ (CLEAN(exit) $\wedge$ exit.nlP $\wedge$ exit.toBeFreed)

Figure 8: Modified datatypes and macros for queue algorithm.

The LL and SC operations used here differ from those in the previous section in several ways. First, because the values stored in `Head` and `Tail` are just pointers, the level of indirection used to support variables of arbitrary size in the previous section is unnecessary: we deal with node pointers directly. Thus, we embed the `exit` and `pred` fields in the queue node structure, as shown in Figure 8.

Second, SC does not allocate and initialize a new node, but rather uses the node passed to it by ENQUEUE or DEQUEUE. Nodes are allocated and initialized by ENQUEUE.

Third, we modify **ExitTag** to support node reclamation appropriate for the queue. In the queue implementation, a node should not be reclaimed until it has been replaced as the `Tail` node *and* it has been replaced as the `Head` node. Each of the SC operations that effect these changes must transfer a count of pinning threads to the node. To detect when both of these transfers have occurred, we replace the boolean flag `nlC` of the **ExitTag** structure in the previous section with a counter `transfersLeft`. This counter is initialized to 2 and decremented by each transfer: when the counter is zero, both transfers have occurred. The `CLEAN` macro is also modified to check whether `transfersLeft` is zero rather than whether `nlC` is set, as shown in Figure 8.

Finally, as before, we use the `exit.nlP` field to avoid the ABA problem when changing the noncurrent pointer to point to a new node on line S4. However, observe that line D8 reads a value from a node that may not be pinned by any LL operation. We must also ensure that this node is not reclaimed before this read occurs. Because only one thread (the one that changes `Head` to point to this node) reads this value, a single additional flag `toBeFreed` suffices (set on line D9 by invoking SETTOBEFREED). As shown in Figure 8, the `FREEABLE` macro is modified to check that the `toBeFreed` flag is also set.

These changes to the LL and SC operations necessitate modifications to the other helper procedures used in the implementation; these modifications are straightforward, and the full code for the LL, SC and other helper procedures can be found in Figure 10.

As with the LL/SC implementation in the previous section, we can avoid the overhead of a general-purpose allocator by using a freelist to store dequeued nodes for future reuse. If we know a bound on the maximum size of the queue, we can populate the freelist in advance and avoid using the general-purpose allocator at all. Otherwise, enough ENQUEUE operations will inevitably require us to allocate new nodes.

5. FREELIST

In this section, we describe how to adapt the queue algorithm of the previous section to implement a lock-free freelist. Our freelist implementation provides four operations: GET, which removes a memory block from the freelist and returns a pointer to that block (or *null* if no block is available); PUT, which takes a pointer to a memory block and puts the block on the freelist;[6] EXPAND, which allocates a new memory block and puts it on the freelist; and CONTRACT, which removes a block from the freelist (if one is available) and frees it. An application using the freelist must guarantee that it will not access the memory block pointed to by a pointer passed to PUT until it is subsequently returned by GET, and that any pointer passed to PUT was returned by some previous invocation of GET.

```
Value * GET() {
    Node *nd = DEQUEUE();
    return (Value *)nd;
}

void PUT(Value *v) {
    SETTOBEENQUEUED((Node *)v);
}

void EXPAND() {
    Node *nd = alloc(Node);
    ENQUEUE(nd);
}

void CONTRACT() {
    Node *nd = DEQUEUE();
    if (nd ≠ null) SETTOBEFREED(nd);
}
```

Figure 9: Freelist operations.

The freelist is basically a queue of nodes, and the `v` field of each node contains a memory block managed by the freelist. The GET operation returns a pointer to the `v` field of a node; applications should be oblivious to the presence of the other fields of the node. The PUT operation takes a pointer to the `v` field; from which we assume it can derive a pointer to the node containing the field.[7]

The freelist code appears in Figure 9. This code invokes ENQUEUE and DEQUEUE operations, which are similar to the corresponding operations of the previous section except that these operations take and return pointers to the `v` field of nodes rather than the values to be stored in those nodes. Because the ENQUEUE operation takes a node, it no longer allocates a new node.

The principal difference between the queue and freelist implementations is that the **ExitTag** type has yet another flag, `toBeEnqueued`. This extra field is necessary because when a node becomes `FREEABLE`, there are two possible actions: If the node was most recently dequeued by the CONTRACT operation then it should be freed, but if it was dequeued by a GET operation and has subsequently been passed back to a PUT operation, then it should be enqueued into the freelist again. Using separate `toBeFreed` and `toBeEnqueued` fields allows us to distinguish the two cases.

[6] We allow the actual placement of memory blocks on the freelist to be delayed. That is, a memory block passed to PUT may not actually be put on the freelist until some time after the PUT operation has returned. The user of the freelist may notice this discrepancy if a GET operation returns *null* after some PUT operation completes.

[7] In our code, we assume that the `v` field is placed at the beginning of the node, so we can use type casting to convert between pointers to nodes and the values they contain.

```
Node LL(LLSCvar *loc) {
    do {
        EntryTag e = loc→entry;
        myver = e.ver;
        mynode = CURRENT(loc, e.ver);
    } while (!CAS(&loc→entry, e, ⟨e.ver, e.count + 1⟩));
    return mynode;
}

bool SC(LLSCvar *loc, Node nd) {
    Node *pred_nd = mynode→pred;
    success = CAS(NONCURADDR(loc, myver), pred_nd, nd);
    if (!success) free(new_nd);
    while ((e = loc→entry).ver == myver)
        if (CAS(&loc→entry, e, ⟨e.ver + 1, 0⟩))
            TRANSFER(mynode, e.count);
    RELEASE(mynode);
    return success;
}

void UNLINK(LLSCvar *loc) {
    while ((e = loc→entry).ver == myver)
        if (CAS(&loc→entry, e, ⟨e.ver, e.count − 1⟩)) return;
    RELEASE(mynode);
}

void TRANSFER(Node *nd, int count) {
    do {
        ExitTag pre = nd→exit;
        ExitTag post = ⟨pre.count + count, pre.transfersLeft − 1,
                        pre.nlP, pre.toBeFreed⟩;
    } while (!CAS(&nd→exit, pre, post));
}

void RELEASE(Node *nd) {
    Node *pred_nd = nd→pred;
    do {
        ExitTag pre = nd→exit;
        ExitTag post = ⟨pre.count − 1, pre.transfersLeft,
                        pre.nlP, pre.toBeFreed⟩;
    } while (!CAS(&nd→exit, pre, post));
    if (CLEAN(post)) SETNLPRED(pred_nd);
    if (FREEABLE(post)) free(nd);
}

void SETNLPRED(Node *pred_nd) {
    do {
        ExitTag pre = pred_nd→exit;
        ExitTag post = ⟨pre.count, pre.transfersLeft,
                        true, pre.toBeFreed⟩;
    } while (!CAS(&pred_nd→exit, pre, post));
    if (FREEABLE(post)) free(pred_nd);
}

void SETTOBEFREED(Node *pred_nd) {
    do {
        ExitTag pre = pred_nd→exit;
        ExitTag post = ⟨pre.count, pre.transfersLeft,
                        pre.nlP, true⟩;
    } while (!CAS(&pred_nd→exit, pre, post));
    if (FREEABLE(post)) free(pred_nd);
}
```

Figure 10: Helper procedures for queue

6. CONCLUDING REMARKS

We have presented a lock-free, CAS-based, 64-bit-clean LL/SC implementation that improves on the only previous one by substantially reducing space requirements, as well as eliminating the need for advance knowledge of the number of threads that will access it. We have also presented the first lock-free 64-bit-clean FIFO queue and freelist implementations that do not require advance knowledge of the number of threads or impose a maximum size on the data structure: their space usage adapts to current requirements. All of these factors are important for portability and practicality.

The difficulty of achieving lock-free 64-bit-clean implementations of such mundane data structures strongly suggests that improved hardware support is necessary before practical lock-free data structures will be widely available. However, we do not believe that 128-bit synchronization primitives are sufficient to achieve this goal; we need synchronization primitives that allow atomic access to multiple, independent memory locations.

7. REFERENCES

[1] *Java Specification Request for Concurrent Utilities (JSR166)*. `http://jcp.org`.

[2] J. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999.

[3] J. Chase, M. Baker-Harvey, H. Levy, and E. Lazowska. Opal: A single address space system for 64-bit architectures (abstract). *Operating Systems Review*, 26(2):9, 1992.

[4] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, 2002.

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

[6] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, page 131, July 2002.

[7] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing*, Oct. 2002.

[8] M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. In *Proceedings of Computing: The Australasian Theory Symposium*, 2003.

[9] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, Nov. 1990.

[10] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Distributed Computing*, July 2003.

[11] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), Aug. 2004.

A preliminary version appeared in *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, 2002.

[12] M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, June 2004.

[13] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(10):1–26, 1998.

[14] *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems, Inc., 1991.

[15] 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.

[16] *PowerPC 601 RISC Microprocessor User's Manual*. Motorola, Inc., 1993.

[17] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[18] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press and Prentice-Hall, 1992.

[19] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[20] J. Valois. Implementing lock-free queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, Oct. 1994.

[21] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.