The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures

Maurice Herlihy[1], Victor Luchangco[2], and Mark Moir[2]

[1] Computer Science Department, Box 1910, Brown University, Providence, RI 02912
[2] Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803

Abstract. We define the *Repeat Offender Problem* (ROP). Elsewhere, we have presented the first dynamic-sized, lock-free data structures that can free memory to any standard memory allocator—even after thread failures—without requiring special support from the operating system, the memory allocator, or the hardware. These results depend on a solution to the ROP problem. Here we present the first solution to the ROP problem and its correctness proof. Our solution is implementable in most modern shared memory multiprocessors.

1 Introduction

A lock-free data structure is *dynamic-sized* if it can grow and shrink over time. Modern programming environments typically provide support for dynamically allocating and freeing memory (for example, the **malloc** and **free** library calls). A data structure is *lock-free* if it guarantees that after a finite number of steps of any operation on the data structure, *some* operation completes. Lock-free data structures avoid many problems associated with the use of locking, including convoying, susceptibility to failures and delays, and, in real-time systems, priority inversion.

This paper presents a new memory management mechanism for dynamic-sized lock-free data structures. Designing such data structures is not easy: a number of papers describe clever and subtle ad-hoc algorithms for relatively mundane data structures such as stacks [14], queues [11], and linked lists [15, 5].

We define an abstract problem, the *Repeat Offender Problem* (ROP), that captures the essence of the memory management problem for dynamic-sized lock-free data structures. Any solution to ROP can be used to permit dynamic-sized lock-free data structure implementations to return unused memory to standard memory allocators. We have formulated this problem to support the use of one or more "worker" threads, perhaps running on spare processors, to do most of the work in parallel with the application's threads.

We present the first solution to the ROP problem, which we call "Pass-the-Buck". In this paper, we focus on the problem statement and a detailed but informal explanation of the algorithm. Elsewhere [6], we show how to apply ROP solutions to achieve the first truly dynamic-sized lock-free data structures,

and we evaluate one such implementation. (As we discuss in Section 3, Maged Michael [10] has concurrently and independently developed a similar technique.)

In the remainder of this section, we discuss why dynamic-sized data structures are challenging to implement in a lock-free manner and then briefly summarize previous related work.

Before freeing an object that is part of a dynamic-sized data structure (say, a node of a linked list), we must ensure that no thread will subsequently modify the object. Otherwise, a thread might corrupt an object allocated later that happens to reuse some of the memory used by the first object. Furthermore, in some systems, even read-only accesses to freed objects can be problematic: the operating system may remove the page containing the object from the thread's address space, causing a subsequent access to crash the program because the address is no longer valid [14].

The use of locks makes it relatively easy to ensure that freed objects are not subsequently accessed because we can prevent access by other threads to (parts of) the data structure while removing objects from it. In contrast, without locks, multiple operations may access the data structure concurrently, and a thread cannot determine whether other threads are already committed to accessing the object that it wishes to free (this can only be ascertained by inspecting the stacks and registers of other threads). This is the root of the problem that our work aims to address.

Below we discuss various previous approaches for dealing with the problem described above.[1] One easy approach is to use garbage collection (GC). GC ensures that an object is not freed while any pointer to it exists, so threads cannot access objects after they are freed. This approach is especially attractive because recent experience (e.g., [2]) shows that GC significantly simplifies the design of lock-free, dynamic-sized data structures. However, GC is not available in all languages and environments, and in particular, we cannot rely on GC to implement GC!

Another common approach is to tag values stored in objects. If we access such values only through compare-and-swap (CAS) operations, we can ensure that a CAS applied to a value after the object has been deallocated will fail [14, 12, 13]. This approach implies that the memory used for tag values can never be used for anything else. One way to ensure that tag memory is never reused is for the application itself to maintain an explicit pool of objects not currently in use [14, 12].

Rather than returning an unused object to the environment's memory management subsystem (say, via the **free** library call), the application places it into its own object pool. An important limitation of application-specific pools is that the application's data structures are not truly dynamic-sized: if the data structures grow large and subsequently shrink, then the application's object pool contains many objects that cannot be coalesced or reused by other applications. In

[1] These approaches are all forms of *type-stable memory* (TSM), defined by Greenwald [4] as follows: "TSM [provides] a guarantee that an object O of type T remains type T as long as a pointer to O exists."

fairness, object pools can provide performance advantages for some applications under some circumstances, bypassing calls to the environment's general-purpose memory allocator.

Elsewhere [6], we show how to eliminate object pools from Michael and Scott's lock-free FIFO queue implementation [12] using the techniques presented in this paper. We also show how to combine the best of both approaches, constructing application-specific object pools that can free excess unused objects to the environment's memory allocator. We also present performance results that show the overhead of making these data structures dynamic-sized is negligible in the absence of contention, and low in all cases. We believe these are the first lock-free, dynamic-sized concurrent data structures that can continue to reclaim memory even after threads fail.

Valois [15] proposed an approach in which the memory allocator maintains reference counts for objects to determine when they can be freed. An object's reference count may be accessed even after the object has been released to the environment's memory allocator. This behavior restricts what the memory allocator can do with released objects (for example, released objects cannot be coalesced). Thus, this approach shares the principal disadvantages of explicit object pools. Valois's approach requires the memory allocator to support certain nonstandard functions, which may make it difficult to port applications to new platforms. Finally, the space overhead for per-object reference counts may be prohibitive. (In [3], we proposed a similar approach that does allow memory allocators to be interchanged, but depends on double compare-and-swap (DCAS), which is not widely supported.)

Our goal is to permit dynamic-sized lock-free data structure implementations to free unneeded memory to the environment's memory allocator through standard interfaces, ensuring that memory allocators can be switched with ease, and that freed memory is not subsequently accessed, permitting the memory allocator to unmap those pages.

Interestingly, the previous work that comes closest to meeting this goal predates the work discussed above by almost a decade. Treiber [14] proposes a technique called "Obligation Passing". The instance of this technique for which Treiber presents specific details is in the implementation of a lock-free linked list supporting search, insert, and delete operations. This implementation allows freed nodes to be returned to the memory allocator through standard interfaces and without requiring any special functionality of the memory allocator. Nevertheless, Obligation Passing employs a "use counter" such that memory is reclaimed only by the "last" thread to access the linked list in any period. As a result, this implementation can be prevented from ever recovering any memory by a failed thread (which defeats one of the main purposes of using lock-free implementations). Another disadvantage of this implementation is that the Obligation Passing code is bundled together with the linked-list maintenance code (all of which is presented in assembly code). Because it is not clear what aspects of the linked-list code are depended upon by the Obligation Passing code, it is difficult to apply this technique to other situations.

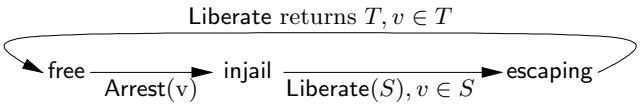

Fig. 1. Tranisition diagram for value v.

2 The Repeat Offender Problem

In this section, we specify the *Repeat Offender Problem* (ROP). Informally, we are given a set of uninterpreted *values*, each of which can be one of three states: free, injail, or escaping. Initially, all values are free. We are given a set of *clients* that interact with values. At any time, a client can Arrest a free value, causing it to become injail, or it can cause an injail value to become escaping. An escaping value can finish escaping and become free again.

Clients *use* values, but must never use a value that is free. A client can attempt to prevent a value v from escaping (becoming free) by "posting a guard" on v. If, however, the guard is posted too late, v may escape anyway. To be safe, a client first posts a guard on v, and then checks whether v is still injail. If so, then a ROP solution must ensure that v does not escape before the guard is removed or redeployed.

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

It is sometimes possible for a client p to determine independently of ROP that a value it wants to use will remain injail until p uses the value (see [6]). In this case, p can use the value without posting a guard.

To support these interactions, ROP solutions provide the following procedures. A thread *posts* a guard g on a value v by invoking PostGuard(g,v), which removes the guard from any value it previously guarded. (A special **null** value is used to *stand down* the guard, that is, to remove the guard from the previously guarded value without posting the guard on a new value). A thread causes a set of values S to begin *escaping* by invoking Liberate(S); the application must ensure that each value in S is injail before this call, and the call causes each value to become escaping. The Liberate procedure returns a (possibly different) set of escaping values causing them to be *liberated* (each returned value becomes free). These transitions are summarized in Figure 1. Finally, a thread can check whether a value v is injail by invoking IsInJail(v); if this invocation returns *true*, then v was injail at some point during the invocation (the converse is not necessarily true, as explained later). Although the Arrest action is specific to the application, the ROP solution must be aware of arrests in order to detect when a free value becomes injail.

If a guard g is posted on a value v, and v is injail at some time t after g is posted on v and before g is subsequently stood down or reposted on a different

value, then we say that g *traps* v from time t until g is stood down or reposted. The main correctness condition for ROP is that it does not allow a value to escape (i.e., become free) while it is trapped.

We now turn our attention to some additional important but mundane details, together with a formal specification of ROP. In some applications (for example, [6]), a client must guard multiple values at the same time. Clients may *hire* and *fire* guards by invoking the HireGuard and FireGuard procedures. Applications' use of guards is expected to follow obvious well-formedness properties, such as ensuring that a thread posts only those guards it employs.

A formal definition of ROP is given by the I/O automaton ([9]) shown in Figure 2, explained below.

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

The automaton consists of a set of *environment actions* and a set of *ROP output actions*. Each action consists of a *precondition* for performing the action and the *effect* on state variables of performing the action. Most environment actions are invocations of ROP operations, and are paired with matching ROP output actions that represent the system's response to the invocations. For example, the $\mathsf{PostInv}_p(g, v)$ action models client p invoking PostGuard(g,v), and the $\mathsf{PostResp}_p()$ action models the completion of this procedure, and similarly for HireGuard(), FireGuard(), and Liberate(). Finally, the Arrest(v) action models the environment (application) arresting value v.

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

Preconditions on the invocation actions specify assumptions about the circumstances under which the application invokes the corresponding ROP procedures. Most of these preconditions are mundane well-formedness conditions such as the requirement that a client posts only guards that it currently employs. The

actions

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

state variables

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

transitions

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

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

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

$\mathsf{IsInJailInv}_p(v)$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p \leftarrow \mathsf{injail}(v)$

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

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

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

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

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

$\mathsf{IsInJailResp}_p(b)$
Pre: for some v, $pc_p = \mathsf{injail}(v)$
  $b \Rightarrow (status[v] = \mathsf{injail})$
Eff: $pc_p \leftarrow \mathsf{idle}$

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

Fig. 2. I/O Automaton specifying the Repeat Offender Problem.

precondition for LiberateInv captures the assumption that the application passes only injail values to Liberate, and the precondition for the Arrest action captures the assumption that only free values are arrested. The application designer must determine how these guarantees are made.

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

Liveness Properties

As specified so far, a ROP solution in which Liberate always returns the empty set, or simply does not terminate, is correct. Clearly, such solutions are unacceptable because each escaping value represents a resource that will be reclaimed only when the value is liberated (returned by some invocation of Liberate). One might be tempted to specify that every value passed to a Liberate operation is eventually returned by some Liberate operation. However, without special operating system support, it is not possible to guarantee such a strong property if threads can fail. Rather than proposing a single nontriviality condition, we instead discuss a range of alternative conditions.

The state variable *numEscaping* counts the number of values currently escaping (that is, passed to some invocation of Liberate and not subsequently returned from any invocation of Liberate). If we require a solution to ensure that *numEscaping* is bounded by some function of application-specific quantities, we exclude the trivial solution in which Liberate always returns the empty set. However, because this bound necessarily depends on the number of concurrent Liberate operations, and the number of values each Liberate operation is invoked with, it does not exclude the solution in which Liberate never returns.

A combination of a boundedness requirement and some form of progress requirement on Liberate operations seems to be the most appropriate way to specify the nontriviality requirement. We later prove that the Pass The Buck algorithm provides a bound on *numEscaping* that depends on the number of concurrent Liberate operations. Because the bound (necessarily) depends on the number of concurrent Liberate operations, if an unbounded number of threads fail while executing Liberate, then an unbounded number of values can be escaping. We emphasize, however, that our implementation does *not* allow failed threads to prevent values from being freed in the future, as Treiber's approach does [14].

Our Pass The Buck algorithm has two more desirable properties. First, the Liberate operation is wait-free (that is, it completes after a bounded number of steps, regardless of the timing behaviour of other threads). This is useful because it allows us to calculate an upper bound on the amount of time Liberate will take to execute, which is useful in determining how to schedule Liberate work.

Finally, our algorithm has a property we call *value progress*. Roughly, this property guarantees that a value does not remain escaping forever provided Liberate is invoked "enough" times (unless a thread fails).

Modular Decomposition

A key contribution of this paper is the insight that an effective way to solve ROP in practice is to separate the implementation of the IsInJail operation from the others.

In our experience using ROP solutions to implement dynamic-sized lock-free data structures [6], values are used in a manner that allows threads to determine whether a value is injail *with sufficient accuracy for the particular application*. As a concrete example, when values represent pointers to objects that are part of a concurrent data structure, these values become injail (allocated) before the objects they refer to become part of the data structure, and are removed from the data structure before being passed to Liberate. Thus, simply observing that an object is still part of a data structure is sufficient to conclude that a pointer to it is injail.

Because we intend ROP solutions to be used with application-specific implementations of IsInJail, the specification of this operation is somewhat weak: it permits an implementation of IsInJail that always returns *false*. However, such an implementation would be useless, usually because it would not guarantee the required progress properties of the application that uses it. Because the circumstances under which IsInJail can and should return *true* depend on the application, we retain the weak specification of IsInJail, and leave it to application designers to provide implementations of IsInJail that are sufficiently strong for their applications. (Note that an integrated, application-independent implementation of this operation, while possible, would be expensive: it would have to monitor and synchronize with all actions that potentially affect the status of each value.)

This proposed modular decomposition suggests a methodology for implementing dynamic-sized lock-free objects: use an "off-the-shelf" implementation of an ROP solution for the HireGuard, FireGuard, PostGuard, and Liberate operations, and then exploit specific knowledge of the application to design an optimized implementation of IsInJail. More precisely, we decompose the ROP I/O automaton into two component automata: the *ROPlite* automaton, and the *InJail* automaton. *ROPlite* has the same environment and output actions as ROP, except for IsInJailInv and IsInJailResp. *InJail* has input action IsInJailInv and output action IsInJailResp. In addition, the *InJail* automaton "eavesdrops" on *ROPlite*: all environment and output actions of *ROPlite* are input actions of *InJail* (though in many cases the implementation of the *InJail* automata will ignore these inputs because it can determine whether a value is injail without them, as discussed above).

We present our Pass The Buck algorithm, which implements *ROPlite* in a simple and practical way, in Section 3.

3 One Solution: Pass The Buck

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

Preliminaries The Pass-the-Buck algorithm is presented in pseudocode, which should be self-explanatory. For convenience, we assume a shared-memory multiprocessor with sequentially consistent memory [8].[2] We further assume that the multiprocessor supports a compare-and-swap (CAS) instruction that accepts three parameters: an *address*, an *old* value, and a *new* value. The CAS instruction atomically compares the contents of the address to the old value, and, if they are equal, stores the new value at the address and returns *true*. If the comparison fails, no changes are made to memory, and the CAS instruction returns *false*. □

The Pass-the-Buck algorithm is shown in Figure 3. The GUARDS array allocates guards to threads. The POST array consists of one location per guard, holding the value that guard is currently assigned to guard, if any, and **null** otherwise. The Liberate operation uses the HANDOFF array to "hand off" responsibility for a value to a later Liberate operation if that value has been trapped by a guard. (For ease of exposition, we assume a bound MG on the number of guards simultaneously employed. It is not difficult to eliminate this assumption by replacing the static GUARDS, POST and HANDOFF arrays with a linked list; guards can be added as needed by appending nodes to the end of the list.)

The HireGuard and FireGuard procedures are based on a long-lived renaming algorithm [1]. Each guard g has an entry GUARDS[g], initially false. Thread p hires guard g by atomically changing GUARDS[g] from *false* (unemployed) to *true* (employed); p attempts hiring each guard in turn until it succeeds (lines 2 and 3). The FireGuard procedure simply sets the guard back to *false* (line 7).

[2] We have implemented our algorithms for multiprocessor architectures based on SPARC® processors that provide Total Store Ordering (TSO) [16], a memory model weaker than sequential consistency that requires additional memory barrier instructions.

```
// handoff entry (CAS target)
typedef struct { value val; int ver } entry;

const int MG = ... // max num guards

// shared variables
bool GUARDS[MG];    // initially false
value POST[MG];     // initially null
entry HANDOFF[MG]; // initially {null,0}
int MAXG= 0;

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

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

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

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

Fig. 3. Code for Pass The Buck.

The HireGuard procedure also maintains the shared variable MAXG, used by the Liberate procedure to determine how many guards to consider. The Liberate operation considers every guard for which a HireGuard operation has completed. In the loop at lines 4 and 5, each HireGuard operation ensures that MAXG is at least the index of the guard returned.

To make PostGuard as efficient as possible, it is implemented as a single store of the value to be guarded in the specified guard's POST entry (line 9).

The most interesting part of the Pass-the-Buck algorithm lies in the Liberate procedure. Recall that Liberate should return a set of values that have been passed to Liberate and have not since been returned by Liberate (i.e., escaping values), subject to the constraint that Liberate cannot return a value that has been continuously guarded by the same guard since some point when it was injail

(i.e., Liberate must not return trapped values). The Liberate procedure maintains a set of escaping values, initially those values passed to it. It checks each guard, removing any values in the set that may be trapped and leaving them behind for later Liberate operations. It also adds to its set values that were left behind by previous Liberate operations but are no longer trapped. After it has checked all guards, it returns the values that remain in its value set.

Suppose thread p is executing a call to Liberate, value v is in the p's value set, and guard g is guarding v. To ensure that Liberate is wait-free, p must either determine that g is not trapping v, or remove v from its value set. To guarantee value progress, if p removes v from its set, then it must ensure that v will be examined by later calls to Liberate. The interesting aspects of the Pass-the-Buck algorithm concern how threads determine that a value is not trapped, and how they store values while keeping space overhead for stored values low.

The loop at lines 12 through 31 iterates over all guards ever hired. For each guard, if p cannot determine for some value v in its set that v is not trapped by that guard, then p attempts to "hand off" that value (there can be at most one such value per guard). If p succeeds (line 18), it removes v from its set (line 19) and proceeds to the next guard (lines 21 and 31). Also, as explained in more detail below, p might simultaneously add to its set a value handed off previously by another Liberate operation; it can be shown that any such value is not trapped by that guard. If p fails to hand v off, then it retries. If it fails repeatedly, it can be shown that v is not trapped by that guard, so p can move on to the next guard without removing v from its set (lines 23 and 25). When p has examined all guards (see line 12), it can safely return any values remaining in its set (line 32).

The following lemma (proved in the full paper) is helpful for understanding why the algorithm works.

Single Location Lemma: Each escaping value v is stored at a single guard or is in the value set of a single Liberate operation (but not both). Also, no non-escaping value is in any of these locations.

At lines 15 and 16, p determines whether the value currently guarded by g (if any) is in its set. If so, p executes the loop at lines 17 through 26 in order to either determine that the value—call it v—is not trapped, or to remove v from its set. To avoid losing v in the latter case, p "hands off" v by storing it in the HANDOFF array. Each entry of this array consists of a value and a version number. Version numbers are incremented with each modification of the entry for reasons discussed below. We assume version numbers are large enough that they can be considered unique for practical purposes (see [13] for discussion and justification).

Because at most one value is trapped by guard g at any time, a single location HANDOFF[g] for each guard g is sufficient. To see why, observe that p attempts to hand off v only if v is in p's value set. If a value w was previously handed off (i.e., it is in HANDOFF[g]), then the Single Location Lemma implies that $v \neq w$, so w is not trapped by g. Thus, p can add w to its value set.

To hand v off, p uses a CAS operation to attempt to replace the value previously stored in HANDOFF[g] with v (line 18). If the CAS succeeds, p adds the replaced value to its set (line 20). We explain below why it is safe to do so. If the CAS fails, then p rereads HANDOFF[g] (line 24) and retries the hand-off. The algorithm is wait-free because the loop completes after at most three CAS operations (lines 13, 22, and 23).

As described so far, p picks up a value from HANDOFF[g] only if its value set contains a value that is guarded by guard g. To ensure that a value does not remain in HANDOFF[g] forever (violating the value progress property), if p does not need to remove a value from its set, it still picks up any previously handed off value and replaces it with **null** (lines 28 through 30).

We now consider each of the ways p can break out of the loop at lines 17 through 26, and explain why it is safe to do so. Suppose p exits the loop after a successful CAS at line 18. As described earlier, p removes v from its set (line 19), adds the previous value in HANDOFF[g] to its set (line 20), and moves on to the next guard (lines 21 and 31). Why is it safe to take the previous value w of HANDOFF[g] to the next guard? The reason is that we read POST[g] (line 15 or 26) between reading HANDOFF[g] (line 14 or 24) and attempting the CAS at line 18. Because each modification to HANDOFF[g] increments its version number field, it follows that w was in HANDOFF[g] when p read POST[g]. Also, recall that $w \neq v$ in this case. Therefore, when p read POST[g], w was not guarded by g. Furthermore, because w remained in HANDOFF[g] from that moment until the CAS, w cannot become trapped in this interval (because a value can become trapped only while it is injail, and all values in the HANDOFF array and in the sets of Liberate operations are escaping). The same argument explains why it is safe to pick up the value replaced by **null** at line 29.

It remains to consider how p can break out of the loop *without* performing a successful CAS. In each case, p can infer that v is not trapped by g, so it can give up on its attempt to hand off v. If p breaks out of the loop at line 26, then v is not trapped by g at that moment simply because it is not guarded by g. The other two places where p may break out of the loop (lines 23 and 25) occur only after the thread has executed several failed CAS operations. The full paper [7] contains a detailed case analysis showing that in each case, v is not trapped.

Discussion

The Pass-the-Buck algorithm satisfies the value progress property because a value cannot remain handed off at a particular guard forever if Liberate is executed enough times. If a value v is handed off at guard g, then the first Liberate operation to begin processing g after v is not trapped by g will ensure that v is picked up and taken to the next guard (or returned from Liberate if g is the last guard), either by that Liberate operation or by a concurrent Liberate operation.

As noted earlier, Michael [10] has independently and concurrently developed a solution to a very similar problem. Michael's algorithm buffers to-be-freed values so that it can control the number of values passed to *Scan* (his equivalent of the Liberate operation) at a time. This has the disadvantage that there are

usually $O(GP)$ values that could potentially be freed, but are not (where G is the number of "hazard pointers"—the equivalent of guards—and P is the number of participating threads). However, this technique allows him to achieve a nice amortized bound on time spent per value freed. He also has weaker requirements for *Scan* than we have for Liberate; in particular, *Scan* can return some values to the buffer if they cannot yet be freed. This admits a very simple solution that uses only read and write primitives (recall that ours requires CAS) and allows several optimizations. However, it also means that if a thread terminates while values remain in its buffer, then those values will never be freed, so his algorithm does not satisfy the *value progress* property. (Michael alludes to possible methods for handing off values to other threads but does not explain how this can be achieved efficiently and wait-free using only reads and writes.) This is undesirable because a single value might represent a large amount of resources, which would never be reclaimed in this case. The number of such values is bounded by $O(GP)$. We can perform the same optimizations and achieve the same amortized bound under normal operation, while still retaining the *value progress* property (although we would require threads to invoke a special wait-free operation before terminating to achieve this). In this case, our algorithm would perform almost identically to Michael's (with a slight increase in overhead upon thread termination), but would of course share the disadvantages discussed above, except for the lack of value progress.

Elsewhere [6], we present a dynamic-sized lock-free FIFO queue constructed by applying our ROP solution to the non-dynamic-sized implementation of Michael and Scott [12] together with the non-dynamic-sized freelist of Treiber [14]. Experiments show that the overhead of the dynamic-sized FIFO queue over the non-dynamic-sized one of [12] is negligible in the absence of contention, and low in all cases.

Michael has shown how to apply his technique to achieve dynamic-sized implementations of a number of different data structures, including queues, double-ended queues, list-based sets, and hash tables (see [10] for references). Because the interfaces and safety properties of our approaches are almost identical, those results can all be achieved using any ROP solution too. In addition, using Pass-the-Buck would allow us to achieve value progress in those implementations. Michael also identified a small number of implementations to which his method is *not* applicable. In some cases, this may be because Michael's approach is restricted to use a fixed number of hazard pointers per thread; in contrast, ROP solutions provide for dynamic allocation of guards. Furthermore, we have presented [6] a general methodology based on any ROP solution that can be applied to achieve dynamic-sized versions of these data structures too. This methodology is based on reference counts, and therefore has disadvantages such as space and time overhead, and inability to reclaim cyclic garbage.

4 Concluding Remarks

We have defined the Repeat Offenders Problem (ROP), and presented one solution to this problem. Such solutions provide a mechanism for supporting memory management in lock-free, dynamic-sized data structures. The utility of this mechanism has been demonstrated elsewhere [6], where we present what we believe are the first dynamic-sized, lock-free data structures that can continue to reclaim memory even if some threads fail (although Maged Michael [10] has independently and concurrently achieved such implementations, as discussed in Section 3).

By specifying the ROP as an abstract and general problem, we allow for the possibility of using different solutions for different applications and settings, without the need to redesign or reverify the data structure implementations that employ ROP solutions. We have paid particular attention to allowing much of the work of managing dynamically allocated memory to be done concurrently with the application, using additional processors if they are available.

The ideas in this paper came directly from insights gained and questions raised in our work on lock-free reference counting [3]. This further demonstrates the value of research that assumes stronger synchronization primitives than are currently widely supported.

Future work includes exploring other ROP solutions, and applying ROP solutions to the design of other lock-free data structures. It would be particularly interesting to explore the various ways for scheduling Liberate work.

Acknowledgments: We thank Steve Heller, Paul Martin, and Maged Michael for useful feedback, suggestions, and discussions. In particular, Steve Heller suggested formulating ROP to allow the use of "spare" processors for memory management work.

References

1. J. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11:1–20, 1997. A preliminary version appeared in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 141-150.
2. D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, 2000.
3. 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.
4. M. Greenwald. *Non-Blocking Synchronization and System Design.* PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999.
5. T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, 2001. To appear.

6. M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. Technical Report TR-2002-110, Sun Microsystems Laboratories, 2002.
7. M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. Technical Report TR-2002-112, Sun Microsystems Laboratories, 2002.
8. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.
9. N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report CWI-Quarterly, 2(3), Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1989.
10. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Distributed Computing*, 2002.
11. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 267–276, 1996.
12. 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.
13. 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.
14. R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.
15. J. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Dsitributed Computing*, pages 214–22, 1995. See `http://www.cs.sunysb.edu/~valois` for errata.
16. D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, Englewood Cliffs, NJ 07632, USA, 1994.