Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes

Maged M. Michael
IBM Thomas J. Watson Research Center
P.O. Box 218 Yorktown Heights NY 10598 USA
magedm@us.ibm.com

ABSTRACT

A major obstacle to the wide use of lock-free data structures, despite their many performance and reliability advantages, is the absence of a practical lock-free method for reclaiming the memory of dynamic nodes removed from dynamic lock-free objects for arbitrary reuse.

The only prior lock-free memory reclamation method depends on the DCAS atomic primitive, which is not supported on any current processor architecture. Other memory management methods are blocking, require special operating system support, or do not allow arbitrary memory reuse.

This paper presents the first lock-free memory management method for dynamic lock-free objects that allows arbitrary memory reuse, and does not require special operating system or hardware support. It guarantees an upper bound on the number of removed nodes not yet freed at any time, regardless of thread failures and delays. Furthermore, it is wait-free, it is only logarithmically contention-sensitive, and it uses only atomic reads and writes for its operations. In addition, it can be used to prevent the ABA problem for pointers to dynamic nodes in most algorithms, without requiring extra space per pointer or per node.

1. INTRODUCTION

A shared object is lock-free (also called non-blocking) if it guarantees that in a system with multiple threads attempting to perform operations on the object, some thread will complete an operation successfully in a finite number of system steps even with the possibility of arbitrary thread delays, provided that not all threads are delayed indefinitely [10]. By definition, lock-free objects are immune to deadlock even with thread failures and provide robust performance even when faced with inopportune thread delays. Dynamic lock-free objects have the added advantage of arbitrary size. Many such objects have been developed [1, 5, 7, 8, 10, 12, 15, 16, 17, 20, 22, 24].

However, a major problem associated with dynamic objects is the safe reclamation of memory occupied by removed nodes. In a lock-based dynamic object, when a thread removes a node from the object, it can be guaranteed that no other thread will subsequently access the contents of the removed node while still assuming its retention of type and content semantics. Consequently, it is safe for the removing thread to free the memory occupied by the removed node into the general memory pool for arbitrary future reuse.

This is not the case for a lock-free object. When a thread removes a node, it is possible that one or more contending threads, in the course of their lock-free operations, have earlier read a pointer to the subsequently removed node, and are about to access its contents. A contending thread might corrupt the shared object or another object, if the thread performing the removal were to free the removed node for arbitrary reuse. Furthermore, on some systems, even read access to freed memory may result in fatal access errors.

Lock-free objects generally use universal atomic primitives such as CAS and LL/SC [9]. CAS (compare-and-swap) takes three arguments: the address of a memory location, an expected value and a new value. If the memory location holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. SC (store-conditional) takes two arguments: the address of a memory location and a new value. If no other thread has written the memory location since the current thread last read it using LL (load-linked), it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. All architectures that support LL/SC restrict memory accesses between LL and SC. Associated with most uses of CAS (and restricted LL/SC) is the ABA problem [12]. If a thread reads a value A from a shared location, computes a new value, and then attempts a CAS operation, the CAS may succeed when it should not, if between the read and the CAS other threads change the value of the shared location from A to B and back to A again.

The simplest and most efficient solution to the ABA problem is to include a tag with the target memory location such that both are manipulated atomically and the tag is incremented with updates of the target location [12]. CAS succeeds only if the tag has not changed since the thread last read the location. However, applying this solution or more elaborate tag techniques [18] to pointers contained in dynamic nodes, without means of detecting when threads no longer need the tag values, dictates that the tag fields retain their values indefinitely, thus preventing the arbitrary reuse of deleted dynamic nodes. Once allocated and inserted in a dynamic lock-free object, a dynamic node may be reused but only if it retains its size and the semantics of its tag fields.

Prior memory management methods for lock-free dynamic objects suffer from one or more serious drawbacks: not allowing arbitrary memory reuse, blocking, requiring special operating system support, or using DCAS.

In this paper we present a new method for safe memory reclamation (SMR) that is wait-free[1], is operating system independent, is only logarithmically contention-sensitive, requires no extra space per node but only a small constant space per participating thread, and requires only atomic reads and writes. It can be used to prevent the ABA problem for dynamic pointers without the need for extra space per pointer or per node. SMR applies to the vast majority of known lock-free dynamic algorithms, and in each of the few cases of incompatible algorithms [5, 8], other similar or better algorithms for the same objects were found to be compatible with it [15, 16].

The core idea of SMR is to associate a small constant number K (typically no more than three) of shared pointers, called *hazard pointers*, with each participating thread. Hazard pointers, either have null values or point to nodes that may potentially be accessed by the thread without further verification of the validity of the local references used in their access. Each hazard pointer can be written only by its associated thread, but can be read by all threads.

SMR requires target lock-free algorithms to guarantee that no thread can access a dynamic node at a time when it is possibly deleted, unless its associated hazard pointers have been pointing to the node continuously, from a time when it was not deleted.

For example, Figure 1 shows an SMR-compatible version of a lock-free stack based on the IBM freelist [12] that guarantees that no dynamic node is accessed while free and prevents the ABA problem. The pointer hp is a static private pointer to the hazard pointer associated with the executing thread, and the procedure DeleteNode is part of the SMR algorithm (Section 3). The Push routine need not change, as no dynamic node that is possibly free is accessed, and the CAS is not ABA-prone as observed by Treiber [22]. In the Pop routine the pointer t is used to access a dynamic node tˆ and holds the expected value of an ABA-prone CAS. By setting the hazard pointer to t (line 2) and then checking that tˆ is not deleted (line 3), we guarantee that the hazard pointer is continuously pointing to tˆ from a point when it was not deleted (line 3) until the end of hazards, i.e., accessing tˆ (line 4) and using t to hold the expected value of an ABA-prone CAS (line 5).

SMR prevents the freeing of a node continuously pointed to by one or more hazard pointers of one or more threads from a point prior to its deletion. When a thread deletes a node by calling DeleteNode, it stores it in a private list. After accumulating a certain number R of deleted nodes, the thread scans the hazard pointers for matches for the addresses of the accumulated nodes. If a deleted node is not matched by any of the hazard pointers, then the thread frees that node making its memory available for arbitrary reuse. Otherwise, the thread keeps the node until its next scan of the hazard pointers which it performs after the number of accumulated deleted nodes reaches R again.

```
// j is the thread id for SMR purposes
// HP is the shared array of hazard pointers
// static private pointer hp ← &HP[j]
// initially Top = null

// calling Push
// if node ← AllocateNode() ≠ null
//       { nodeˆ.Data ← data; Push(node);}
Push(node:*NodeType) {
      while true {
          t ← Top;
          nodeˆ.Next ← t;
          if CAS(&Top,t,node) return;
      }
}

// calling Pop
// if node ← Pop() ≠ null
//       { data ← nodeˆ.Data; DeleteNode(node);}
Pop() : *NodeType {
      while true {
1         if t ← Top = null break;
2         *hp ← t;
3         if t ≠ Top continue;
4         next ← tˆ.Next;
5         if CAS(&Top,t,next) break;
      }
      *hp ← null; return t;
}
```

Figure 1: SMR-compatible lock-free stack based on the IBM freelist algorithm [12] (SMR-related code is in a different font).

By setting R to a number such that $R - N = \Omega(N)$ (e.g., $R = 2N$), where $N = KP$ and P is the number of participating threads, and sorting a private list of snapshots of non-null hazard pointers, SMR is guaranteed in every scan of the N hazard pointers to free $\Theta(R)$ nodes in $O(R \log p)$ time, when p threads have non-null hazard pointers. Thus, the amortized time complexity of processing each deleted node until it is freed is only logarithmically contention-sensitive, i.e., constant in the absence of contention and $O(\log p)$ when p threads are operating on the object during the scan of their associated hazard pointers. SMR also guarantees that no more than PR deleted nodes are not yet freed at any time regardless of thread failures and delays.

In Section 2 we review prior approaches to memory management for dynamic lock-free objects. In Section 3 we present the SMR method. In Section 4 we discuss its correctness and performance. In Section 5 we apply SMR to known dynamic lock-free algorithms and we conclude with Section 6.

2. RELATED WORK

Lock-free OS-Independent Methods

The earliest and simplest lock-free memory management method is the use of a freelist implemented as a stack of nodes available for restricted reuse [12, 22]. When a thread

[1]An operation is wait-free if it is guaranteed to complete successfully in a finite number of its own steps regardless of other threads' actions [9].

deletes a node, it pushes it in a singly linked list stack, where the same or a different thread can later reuse the node by popping it from the stack. The operations are similar to those in Figure 1 excluding data and hazard pointer manipulation and node allocation and deallocation. Nodes must retain their size and the semantics of some fields indefinitely (i.e., until application termination). Once allocated and inserted in a lock-free object, a node cannot be freed for arbitrary reuse. Freelists require applying ABA prevention mechanisms such as tags to the anchor of the freelist as well as pointers contained in nodes for almost all lock-free objects except stacks.

Valois [25] presented a lock-free garbage collection method that requires the inclusion of a reference counter in each dynamic node, reflecting the maximum number of references to that node in the object and the registers (and local variables) of threads operating on the object. Every time a new reference to a dynamic node is created/destroyed, the reference counter is incremented/decremented, using fetch-and-add and CAS instructions. Only after its reference counter goes to zero, can a node be pushed in a freelist, where it can be subsequently allocated. However, due to the use of single-address CAS to manipulate pointers and independently located reference counters non-atomically, the resulting timing windows dictate the permanent retention of nodes of their type and field semantics, thus prohibiting memory reclamation as is the case with freelists.

Detlefs *et. al.* [6] presented lock-free reference counting (LFRC), a lock-free garbage collection method that uses DCAS[2] to operate on pointers and reference counters atomically to guarantee that a reference counter is never less than the actual number of references. When the reference counter of a node reaches zero, it becomes safe to reclaim for arbitrary reuse. Both reference counting methods avoid the ABA problems without using a tag per pointer (they use a counter per node instead). LFRC also allows memory reclamation, which is a significant advantage. However, the dependence on DCAS makes the method impractical. For both methods the performance cost of reference counting is prohibitive.

The reference counting methods transform statements involving pointers to dynamic nodes (even reads and register to register assignments) into unbounded loops containing CAS and DCAS operations. This increases the total work time complexity of the target algorithms by a *multiplicative* factor of $O(p)$ where p is contention.

DCAS was supported on some generations of the Motorola 68000 processor family (as CAS2) in the 1980s. These implementations were extremely inefficient, often requiring locking system buses while one CAS2 is in progress. Since then no processor architecture supports DCAS.

As discussed later in Section 5, while examining known lock-free dynamic algorithms, we found only two [5, 8] to be incompatible with SMR and freelists, but compatible with the reference counting methods. However, in each case, we found other similar or better algorithms that are compatible with SMR and freelists for the same objects [15, 16].

[2]DCAS (double-compare-and-swap) takes six arguments: the addresses of two independent memory locations, two expected values and two new values. If both memory locations hold the corresponding expected values, they are assigned the corresponding new values atomically. A Boolean return value indicates whether the replacements occurred.

Blocking and OS-Dependent Methods

Herlihy and Moss [11] presented a lock-free garbage collection method that requires a clean sweep from each participating thread, as a precondition for memory reclamation. The failure of a thread can indefinitely prevent the freeing of unbounded memory (i.e., bounded only by the size of memory). The method also suffers from substantial copying overhead.

McKenney and Slingwine [14] presented read-copy update, a framework where only when each thread is certain to have reached a *quiescence point* after a node is deleted, can the memory of that node be freed. The definition of quiescence points, if any, varies depending on the environment, but mostly depends on timestamps or collective reference counters. Not all environments are suitable for the concept of quiescence points, and the method is blocking as the delay of even one thread prevents freeing an unbounded number of nodes.

Greenwald [7] presented sketches of Type Stable Memory (TSM) implementations in the OS kernel and in user-level. TSM requires deleted nodes to retain their type while references to them are potentially active. The kernel-based TSM implementation relies on the kernel's knowledge of the status of processes and its access to their private memory and stack space. The drawbacks of kernel-dependence are that it limits the portability of dependent algorithms to systems lacking the special kernel support, and precludes their use by most user-level threads.

The user level TSM implementation requires threads to increment and decrement a per-type reference counter upon the activation and end of scope, respectively, of local pointers to the node type. A variant of the method uses two reference counters per type: an old counter and a new counter. A thread always increments the new counter and decrements the counter it incremented earlier. The counters switch roles whenever the old counter reaches zero. At such time, deleted nodes are released to the general pool. The method is blocking as the failure of a thread to decrement a per-type reference counter, due to the thread's failure or delay, may cause the indefinite prevention of an unbounded number of deleted nodes from being freed.

Harris [8] presented a brief outline of a deferred freeing method that requires each thread to record a timestamp of the latest time it held no references to dynamic nodes, and it maintains two to-be-freed lists of deleted nodes: an old list and a new list. Deleted nodes are placed on the new list and when the time of the latest insertion in the old list precedes the earliest per-thread timestamp, the nodes of the old list are freed and the old and new lists exchange labels. The method is blocking, as the failure of a thread to update its timestamp causes the indefinite prevention of an unbounded number of deleted nodes from being freed.

One crucial difference between SMR and these methods [7, 8, 14] is that SMR does not use reference counters or timestamps at all. The use of collective reference counters for unbounded numbers of nodes and/or the reliance on per thread timestamps make a memory management method vulnerable to the failure or delay of even one thread.

Tracing garbage collectors do not provide OS-independent non-blocking solutions for the memory reclamation problem. They mostly require mutual exclusion with mutator threads, use stop-the-world techniques, or require special OS support to access private stack space and registers [11]. Furthermore, the failure or indefinite delay of the garbage collector can prevent the reclamation of unbounded number of deleted nodes indefinitely [6].

3. THE METHOD

The core idea of the SMR method is associating K shared pointers, called hazard pointers, with each of the P threads operating on the target object. The value of K depends on the target algorithm and is typically no more than three. Hazard pointers are implemented as a shared array HP of size N, where $N = KP$.

The SMR algorithm communicates with the target algorithms only through a DeleteNode procedure that is part of the SMR algorithm, and the hazard pointer array HP. Each hazard pointer can be written only by its associated thread in the target algorithm. The SMR algorithm itself does not perform any shared writes.

3.1 The Algorithm

Figure 2 shows the structures and operations of the SMR algorithm. We assume a sequentially consistent memory model. Otherwise, memory barriers may be needed in between instructions with critical relative order.

When a thread j deletes a node nˆ by calling DeleteNode(n), it inserts nˆ into a static private list *dlist* of deleted nodes, and increments a static private counter *dcount* that holds the number of deleted nodes accumulated by j that are not freed yet. When *dcount* reaches the value R, j starts a scan by calling Scan(). R is chosen such that $R - N = \Omega(N)$ (e.g., $R = 2N$), in order to achieve amortized execution time that is logarithmic in contention per freed node.

A scan includes four stages. The first stage involves scanning the array HP for non-null values. Whenever a non-null value is encountered, it is inserted in a local pointer list *plist*. The counter *p* holds the size of *plist*, which is proportionally bounded by contention. Only stage 1 accesses shared variables. The following stages operate only on private variables.

The second stage of a scan involves sorting *plist* to allow binary search in the third stage.

The third stage of a scan involves checking each node in *dlist* against the pointers in *plist*. If the binary search yields no match, the node is freed. Otherwise, it is inserted into a local list *new_dlist*.

The forth stage involves copying the local list *new_dlist* of the nodes that could not be freed during the current scan to the private static list *dlist*, where they remain until the next scan, which occurs after $R{-}new\_dcount$ more nodes are deleted by j.

We assume the use of a comparison-based sorting algorithm that takes $\Theta(p \log p)$ time, such as heap sort, to sort *plist* in the second stage. Binary search in the third stage takes $O(\log p)$ time. We omit the code for these algorithms, as they are widely known sequential algorithms [4].

```
// Constants
// P : number of participating threads
// K : number of hazard pointers per thread
// N : total number of hazard pointers = KP
// R : batch size, R-N = Ω(N)

// shared variables
HP[N] = null : *NodeType;

// static private variables per thread
dcount = 0 : 0 .. R;
dlist[R] : *NodeType;

DeleteNode(node:*NodeType) {
  dlist[dcount++] ← node;
  if dcount = R Scan();
}

Scan() {
  i : 0 .. R;
  p = 0,new_dcount = 0 : 0 .. N;
  hptr,plist[N],new_dlist[N] : *NodeType;

  // Stage 1
  for i ← 0 to N-1
    if hptr ← HP[i] ≠ null
      plist[p++] ← hptr;
  // Stage 2
  sort(p,plist);
  // Stage 3
  for i ← 0 to R-1
    if binary_search(dlist[i],p,plist)
      new_dlist[new_dcount++] ← dlist[i];
    else
      FreeNode(dlist[i]);
  // Stage 4
  for i ← 0 to new_dcount-1 { dlist[i] ← new_dlist[i];}
  dcount ← new_dcount;
}
```

Figure 2: The SMR algorithm.

Optimizations

The static space requirements per thread can be reduced to a constant, if the node structures used by the target algorithm contain a word size field that is guaranteed not to be accessed by the target algorithm after a node is deleted. The vast majority of algorithms use nodes that contain such fields (e.g., data fields). Also, nodes often contain extra space for alignment or for cache line padding to avoid false sharing. If so, the chosen field is also defined as a pointer *smrp* to NodeType and is used to link deleted nodes into a linked list instead of using an array of size R per thread to accumulate deleted nodes.

Removing duplicates from *plist* after sorting it can reduce search time when duplicates are frequent, which is very common for constant time algorithms such as those for stacks and queues. Figure 3 shows a version of the SMR algorithm incorporating these optimizations.

In order to reduce the overhead of calling the standard allocation and deallocation procedures (e.g., malloc and free) for every node allocation and deallocation, SMR can allow each thread to maintain a limited size private list of free nodes. When a thread runs out of private free nodes it allocates new nodes when needed, and when a thread accumulates too many private free nodes it deallocates the excess nodes.

In order to avoid the adverse performance effect of false sharing on multiprocessor systems, elements of the HP array can be padded such that no two hazard pointers belonging to different threads share the same cache line.

```
// Constants and shared vars are unchanged

// static private variables per thread
dcount = 0: 0 .. R;
dlist = null : *NodeType;

DeleteNode(node:*NodeType) {
  node^.smrp ← dlist; dlist ← node; dcount++;
  if dcount = R  Scan();
}

Scan() {
  i,p = 0,new_dcount = 0 : 0 .. N;
  hptr,plist[N],new_dlist = null,node : *NodeType;

  // Stage 1
  for i ← 0 to N-1
    if hptr ← HP[i] ≠ null
      plist[p++] ← hptr;
  // Stage 2
  sort(p,plist);
  remove_duplicates(&p,plist);
  // Stage 3
  while dlist ≠ null {
    node ← dlist; dlist ← node^.smrp;
    if binary_search(node,p,plist)
      { node^.smrp ← new_dlist;
        new_dlist ← node; new_dcount++;}
    else
      FreeNode(node);
  }
  // Stage 4
  dlist ← new_dlist;
  dcount ← new_dcount;
}
```

Figure 3: The SMR algorithm with optimizations.

In many applications, threads are created and destroyed frequently. In such cases, a static-size (of size P) lock-free freelist based on the IBM freelist [12] can be used for maintaining available thread ids. A thread acquires an SMR thread id by popping it from the freelist, and retires its id by pushing it in the freelist. The freelist elements can include fields for use by retiring threads to pass their list of deleted but not yet freed nodes to their next namesake.

3.2 Compatibility Conditions

For a correct target algorithm to benefit from SMR's guarantees of safe memory reclamation, it must satisfy a set of conditions. First, we define some notation.

Delete(t, j, n): Thread j deletes node $n\hat{}$ at time t: at t, j calls DeleteNode(n).

Allocate(t, j, n): Thread j allocates node $n\hat{}$ at time t: at t, j's call to AllocateNode() returns n.

IsDeleted(t, n): Node n^ is deleted at time t : $\exists t_1 < t, j_1$ :: $Delete(t_1, j_1, n) \wedge \nexists t_2 \in [t_1, t], j_2 :: Allocate(t_2, j_2, n)$.

PossiblyDeleted(t, j, n): A node n^ is possibly deleted from the point of view of thread j at time t : at t, it is impossible solely by examining j's registers (private variables included) and the semantics of the algorithm to establish that $\neg IsDeleted(t, n)$.

CreateRef$(\hat{t}, j, n, \hat{r})$: Thread j creates the reference $\langle \hat{t}, j, n, \hat{r}\rangle$ to node $n\hat{}$ in register $\hat{r}$ at time $\hat{t}$: $\exists$ shared pointer p :: at $\hat{t}$, j reads the value n from p into $\hat{r}$.

AssignRef$(\hat{t}, j, n, \hat{r}, t, r)$: Thread j assigns the reference $\langle \hat{t}, j, n, \hat{r}\rangle$ to register r at time t : $(t = \hat{t} \;\wedge\; r \equiv \hat{r} \;\wedge\; CreateRef(t, j, n, r)) \;\vee\; (\exists$ register r_1, time $t_1 \in [\hat{t}, t]$ :: $AssignRef(\hat{t}, j, n, \hat{r}, t_1, r_1) \wedge$ at t, j assigns the value of r_1 to $r \wedge \nexists t_2 \in (t_1, t]$ :: at t_2, r_1 is the target of an assignment).

HoldRef$(\hat{t}, j, n, \hat{r}.t, r)$: Register r of thread j holds reference $\langle \hat{t}, j, n, \hat{r}\rangle$ at time t : $\exists t_1 \le t :: AssignRef(\hat{t}, j, n, \hat{r}, t_1, r)$ $\wedge$ $\nexists t_2 \in (t_1, t]$ :: at t_2, r is the target of an assignment.

HoldHazRef$(\hat{t}, j, n, \hat{r}, t, r)$: Register r of thread j holds the hazardous reference $\langle \hat{t}, j, n, \hat{r}\rangle$ at time t: $HoldRef(\hat{t}, j, n, \hat{r}, t, r) \;\wedge\; \exists$ a code path c, statement s :: at t, it is possible for j to follow $c \;\wedge\; s$ is the last statement of $c \;\wedge\; \exists$ register r_2 :: s uses r_2 to access $n\hat{}$ when it is $PossiblyDeleted \wedge ((\exists\ t_2 \ge t :: j$ follows $c \wedge j$ executes s at $t_2) \implies ((r \equiv r_2 \wedge \nexists$ statement in c that uses r as the target of an assignment$) \vee (\exists r_1, t_1 \in [t, t2]$ :: at t_1 j assigns the value of r to $r_1 \;\wedge\; HoldHazRef(\hat{t}, j, n, \hat{r}, t_1, r_1))))$.

Informally, A hazardous reference is an address that without further validation can be used to access a node after it has been deleted.

The Conditions

For a lock-free dynamic algorithm to be SMR-compatible, it must satisfy the following main condition:

$$\forall\ t, n, j, r, \hat{t}, \hat{r},\quad PossiblyDeleted(t, j, n) \;\wedge\; HoldHazRef(\hat{t}, j, n, \hat{r}, t, r) \implies \text{at } t, \exists\ 0 \le i < K :: HP[Kj + i] = n \qquad \text{(C1)}$$

Informally, whenever a thread accesses a dynamic node, it must guarantee that if the node was possibly deleted (by another thread) after the creation of the reference to it, then continuously from a time equal to or earlier than the time of its possible deletion, one or more of the thread's hazard pointers is pointing to the node.

When $K > 1$, it is acceptable for a thread to overwrite the address of a node n^ in one of its associated hazard pointers, when n^ is possibly deleted and one of its registers holds a hazardous reference to n^, as long as another associated hazard pointer was already assigned the value n.

However, since the SMR algorithm scans the array HP non-atomically (one hazard pointer at a time) in a certain order, passing the responsibility for hazardous references from one hazard pointer to another must strictly follow the same order as scanning HP. This leads to the second condition:

$$\begin{aligned}&\forall j, n, t_1 < t_2, 0 \le i_1 < K, 0 \le i_2 < K, i_1 \ne i_2,\\ &\quad ((\exists \hat{t}, \hat{r}, r :: \forall t \in [t_1, t_2], \text{at } t,\\ &\qquad PossiblyDeleted(t, j, n) \wedge\\ &\qquad HoldHazRef(\hat{t}, j, n, \hat{r}, t, r)) \wedge\\ &\quad (\text{at } t_1, HP[Kj + i_1] = n \wedge\ HP[Kj + i_2] \ne n) \wedge\\ &\quad (\text{at } t_2, HP[Kj + i_1] \ne n \wedge\ HP[Kj + i_2] = n)\\ &\quad) \implies i_1 < i_2 \qquad \text{(C2)}\end{aligned}$$

The last condition is optional. It is needed only to guarantee that the number of non-null hazard pointers (p at the beginning of stage 3 of Scan) is proportionally bounded by contention:

$$\forall\ t, j, 0 \le i < K,\quad \text{at } t, \text{ thread } j \text{ is not operating on the object} \implies \text{at } t,\ HP[Kj + i] = null \qquad \text{(C3)}$$

4. CORRECTNESS AND PERFORMANCE

For brevity, we provide only informal proof sketches. Formal proofs are to be included in the extended version of this paper.

Safety

LEMMA 1. $\forall j, t, n,$
$\exists k < K, t_0 \leq \cdots \leq t_{k-1} \leq t :: \forall\, 0 \leq i < k, \mathit{at}\ t_i, HP[Kj+i] \neq n \ \wedge$
$\mathit{IsDeleted}(t, n) \ \wedge\ \exists r, \hat{r}, \hat{t} :: \mathit{HoldHazRef}(\hat{t}, j, n, \hat{r}, t, r)$
$\implies \exists k \leq m < K :: \mathit{at}\ t, HP[Kj+m] = n.$

If a scan in ascending order of low indexed hazard pointers of a thread finds no match for a deleted node for which the thread holds a hazardous reference, then there must be at least one higher indexed hazard pointer of that thread that points to that node. This follows from (C1) and (C2) and the fact that $\mathit{IsDeleted}(t, n) \implies \forall j, \mathit{PossiblyDeleted}(t, j, n)$.

LEMMA 2. $\forall j, t, n,$
$\exists\ t_0 \leq \cdots \leq t_{K-1} \leq t :: \forall\ 0 \leq i < K,\ \mathit{at}\ t_i, HP[Kj+i] \neq n \ \wedge$
$\mathit{IsDeleted}(t, n) \implies \nexists\ r, \hat{r}, \hat{t} :: \mathit{HoldHazRef}(\hat{t}, j, n, \hat{r}, t, r).$

If a scan in ascending index order of the hazard pointers of a thread finds no match for a deleted node, then it must be the case that the thread holds no hazardous reference for the node. This follows from Lemma 1.

LEMMA 3. $\forall t, n,$ *node* $n\hat{}$ *is freed at* t *in stage 3 of Scan*
$\implies \forall j, \exists t_0 \leq \cdots \leq t_{K-1} \leq t :: \forall\ 0 \leq i < K,\ \mathit{at}\, t_i, HP[Kj+i] \neq n.$

A node is freed in stage 3 of Scan only if a scan of the hazard pointers in ascending index order finds no match. This follows from the fact that stage 1 scans HP in ascending index order and the fact that the list *plist* does not lose distinct pointer values throughout stages 1, 2, and 3. That is, if a pointer value is not in *plist* in stage 3 then it couldn't have been there any time during stage 1.

THEOREM 1. $\forall t, n,$ *node* $n\hat{}$ *is freed at* t *in stage 3 of Scan*
$\implies \nexists\ j, r, \hat{r}, \hat{t} :: \mathit{HoldHazRef}(\hat{t}, j, n, \hat{r}, t, r).$

If SMR frees a node then it must be the case that no thread holds a hazardous reference to it. This follows transitively from Lemmas 2 and 3 and the fact that only deleted nodes are processed in Scan.

From the definition of HoldHazRef, we get the following corollary.

COROLLARY. *SMR guarantees that no thread accesses the contents of a node while the node is free.*

Time Complexity

LEMMA 4. *At the beginning of Scan, the list dlist contains exactly R distinct deleted nodes.*

LEMMA 5. *Throughout Scan, $p \leq N$.*

LEMMA 6. *At most N of the searches of the list plist in stage 3 of Scan find matching pointers.*

LEMMA 7. *Every call to Scan results in freeing at least $R - N$ (i.e., $\Theta(R)$) nodes.*

LEMMA 8. *The execution time of DeleteNode is constant.*

LEMMA 9. *The execution time of Scan (excluding calls to FreeNode) is $O(R \log p)$.*

Stage 1 takes $\Theta(N)$ time. Stage 2 takes $\Theta(p \log p)$ time. Stage 3 takes $O(R \log p)$ time (excluding calls to FreeNode). Stage 4 takes $O(p)$ time (constant time if *dlist* is implemented as a linked list). $p = O(N)$. $N = O(R)$.

THEOREM 2. *The amortized time complexity of processing a deleted node until freeing it is $O(\log p)$.*

Wait-Freedom

THEOREM 3. *The SMR algorithm is wait-free.*

A thread executing the SMR algorithm (at most R calls to DeleteNode and one call to Scan) is guaranteed to complete successfully (i.e., free $\Theta(R)$ nodes) in a finite number of its own steps (i.e., $O(R \log p)$).

Bound on Number of Deleted Nodes not Freed

LEMMA 10. $\forall\ t, j,$
$|\{n \colon \exists t_1 < t :: \mathit{Delete}(t_1, j, n) \ \wedge\ \nexists t_2 \in [t_1, t] :: \mathit{at}\ t_2,\ j\ \mathit{frees}$
$n\hat{}\}|$ *is finite (at most R).*

Each thread holds a finite number of deleted but not yet freed nodes.

THEOREM 4. *At any time, the total number of deleted nodes not freed is finite (at most PR).*

There is a tradeoff between the bound on deleted nodes not freed and the amortized time of processing each freed node. In an earlier version of the Scan algorithm that used single-word CAS, the upper bound on the number of deleted nodes not freed was only $O(P)$ (constant R), but the amortized time complexity per freed node was $\Theta(P)$.

Space Requirements

The space requirements of SMR are N (i.e., $\Theta(P)$) shared words for hazard pointers and $\Theta(R)$ static private words per thread (only a constant when using a linked list to store deleted nodes instead of an array as in Figure 3). The temporary space needed per thread when performing a scan is $\Theta(P)$.

5. APPLYING SMR

In this section, we apply the SMR method to the best known dynamic lock-free algorithms. We examined dozens of dynamic lock-free algorithms, many of which are not mentioned here for brevity. Only two correct algorithms [5, 8] were found to be inherently incompatible with SMR as well as freelists. In each case a similar or better SMR-compatible algorithm is found for the same object.

ABA Prevention

In all of the cases of applying SMR to dynamic algorithms discussed in this section, it not only solved the memory reclamation problem, but was also used to prevent the ABA problem for all or some pointers without using any extra space per pointer or per node. This was done by expanding the definition of hazardous references (defined in Section 3) to include references to dynamic nodes that may be used as expected values of ABA-prone instructions. However, we do not claim SMR to be a general solution for the ABA problem for pointers to dynamic nodes, as we can construct hypothetical cases where a reference to a dynamic node can be the expected value of an ABA-prone instruction even when that node is never removed and reallocated. Also, in some SMR-compatible algorithms (e.g., [15]), a pointer may point to nodes that are already deleted and still be the target of ABA-prone instructions. In such cases, it may be preferable to use ABA-prevention tags than to reconstruct the algorithm to prevent the deletion of nodes pointed to by pointers that are possible targets of ABA-prone instructions. Of course, in such cases, the SMR algorithm will still be able to detect when the dynamic node containing such a pointer can be freed safely.

We present and discuss SMR-compatible versions of the best known dynamic lock-free algorithms for FIFO queues, stacks, double-ended queues, list-based sets, priority queue skew heaps, and universal methodologies. We also discuss the cases of algorithms that are inherently incompatible with SMR.

FIFO Queues

We were able to create SMR-compatible versions of all three dynamic lock-free FIFO queue algorithms [20, 24, 17] that are correct, completely defined, fully functional, CAS-based, and have constant time operations in the absence of contention.

Figure 4 shows an SMR-compatible version of Michael and Scott's [17] algorithm. In the enqueue routine, we notice that register t holds hazardous references. It is used to access dynamic nodes (lines 4 and 7). It holds the expected value of ABA-prone CAS operations (lines 6 and 8). It holds the expected value of an ABA-prone validation condition (line 5). Therefore we dedicate a hazard pointer (*hp0) for pointing to t^ whenever t holds a hazardous reference.

The pattern of lines 1–3 is ubiquitous in SMR-compatible algorithms. We observe that after executing line 1 and before executing line 3, it is possible that t^ is removed and then reinserted in the object. But this window poses no problem. During that period the reference held in t is not hazardous, as line 3 verifies the reachability of t^, and hence precludes the possibility of it being deleted at that point. The reference only becomes hazardous immediately after passing the condition in line 3. If t^ is removed before line 2 and not reinserted before line 3, the condition in line 3 forces the thread to retry. If t^ is reinserted before line 3 and the condition in line 3 allows the thread to proceed then at that point it is guaranteed that *hp0 points to t^ from a point when it was not deleted and continues to do so until the reference is no longer hazardous, thus complying with condition (C1).

We employed the technique of lines 1–3 in creating SMR-compatible transformations of most known algorithms. We present additional and alternative techniques when discussing list-based sets and universal methodologies.

The dequeue routine employs a similar technique for dealing with register h. For the register next, no equality check (*if next ≠ t^.Next continue;*) is needed as the validation condition in line 15 (from the original algorithm) guarantees that *hp1 points to next^ from a point when it was not deleted (next^ cannot be removed unless h^ is removed first) and continues to do so until the reference in next is no longer hazardous (i.e. after line 18). Only when register t is equal to h, is it used in CAS. So it is always covered by *hp0. No other registers in the algorithm can hold hazardous reference, and so the algorithm satisfies condition (C1).

It is easy to show that the algorithm satisfies condition (C2) since no hazard pointer inherits its value from another hazard pointer. As for condition (C3), the algorithm guarantees that whenever a thread exits Enq or Deq each of its hazard pointers is equal to null.

```
// j is thread id for SMR purposes
// hp0 = &HP[2*j]
// hp1 = &HP[2*j+1]

// initially both Head and Tail point to a dummy node

Enq(data:DataType) : boolean {
     if node ← AllocateNode() = null return false;
     node^.Data ← data; node^.Next ← null;
     while true {
 1      t ← Tail;
 2      *hp0 ← t;
 3      if t ≠ Tail continue;
 4      next ← t^.Next;
 5      if t ≠ Tail continue;
 6      if next ≠ null { CAS(&Tail,t,next); continue;}
 7      if CAS(&t^.Next,null,node) break;
     }
 8   CAS(&Tail,t,node);
     *hp0 ← null; return true;
}

Deq() : DataType {
     while true {
 9      h ← Head;
10      *hp0 ← h;
11      if h ≠ Head continue;
12      t ← Tail;
13      next ← h^.Next;
14      *hp1 ← next;
15      if h ≠ Head continue;
16      if next = null{ *hp0 ← null; return EMPTY;}
17      if h = t { CAS(&Tail,t,next); continue;}
18      data ← next^.Data;
19      if CAS(&Head,h,next) break;
     }
     *hp0 ← null; *hp1 ← null;
     DeleteNode(h); return data;
}
```

Figure 4: SMR-compatible version of Michael and Scott's [17] lock-free queue algorithm.

Stacks

In Figure 1, we presented an SMR-compatible lock-free stack based on the IBM freelist [12]. The techniques employed in transforming it are covered by those employed in lines 1–3 of Figure 4.

Double-Ended Queues (Deques)

Agesen *et. al.* [1] presented a DCAS-based dynamic lock-free deque algorithm. We were successful in making that algorithm SMR-compatible. Detlefs *et. al.* [5] presented a simpler algorithm that also uses DCAS. However, the algorithms assumes automatic garbage collection, since after popping a node, it is not always possible for a thread, solely based on its registers, to determine if it had removed nodes from the deque's doubly-linked list or not. The algorithm is inherently incompatible with SMR and freelists. It is impossible to determine when to call DeleteNode, which is an integral part of explicit memory management methods.

Recently, we developed the first CAS-based lock-free deque algorithm [15]. The algorithm is compatible with SMR and freelists (as well as all other memory management methods). As part of presenting that algorithms, we demonstrate its compatibility with SMR. The techniques employed in generating its SMR-compatible version are covered by those employed in Figure 4.

List-Based Sets

Valois [25] presented CAS-based lock-free algorithms for link-based sets that use shared cursors, such that whenever a thread operates on the object it first moves the cursor (using CAS) to the desired location and then attempts its intended operation, such as inserting, deleting, and searching for a key. Actually, the algorithms are not even livelock-free, as two threads may indefinitely alternate moving the cursor to their respective desired locations, while neither succeeds in performing its intended operation on the object.

Harris [8] presented a lock-free list-based set algorithm using CAS. The algorithm cannot use freelists for memory management. Harris implemented the algorithm using Valois' [25] memory management method, and in order to avoid the high overhead of reference counting he also introduced his own memory management method that is actually blocking, as discussed in Section 2. Harris' set algorithm is not compatible with SMR as a thread may traverse a sequence of nodes after they have already been removed from the object, and hence possibly deleted. Thus, it is impossible for the algorithm to comply with SMR's condition (C1).

Recently, we developed a simple CAS-based lock-free list-based set algorithm (as part of a lock-free hash table algorithm [16]) that is compatible with all memory management methods. It yields better performance than Harris' algorithm, as it is compatible with simpler and better performing memory management methods. As part of presenting that algorithm we demonstrate its compatibility with SMR.

The algorithm involves the traversal of the linked list using three registers, we use three hazard pointers per thread to track the registers. As the registers move down the list, the third register assumes the value of the second, then the second assumes the value of the first, and then the first moves to the next node. The hazard pointers mirror the same movement. The first hazard pointer uses a technique similar to that used in lines 1–3 of Figure 4. The third and second hazard pointers inherit their values from the second and the first hazard pointers, respectively. So, in order to comply with SMR's condition (C2), the indices in the array HP of the first, second and third hazard pointers associated with each thread must be in ascending order, respectively.

Universal Methodologies

Herlihy [10] presented a universal methodology for transforming sequential dynamic object into lock-free ones. The methodology uses a single anchor for the target object. The value of the anchor must change with every modification of the object. A thread operating on the object starts by reading the anchor then as it traverses the linked object it makes a private copy of each node it needs to access and uses the copy instead. At the end of the operation the thread uses CAS (or SC) to change the anchor to a new value. If successful the private nodes (if any) become part of the object, and the thread puts the removed nodes in a private pool of nodes. If the CAS is not successful, the thread takes the allocated private nodes back into its private pool and starts over by reading the anchor again. Herlihy provides a recoverable set algorithm for maintaining the pool of private nodes. In order to establish the validity of copied nodes, Herlihy uses two check bits written and read in reverse order.

We demonstrate an SMR-compatible version of Herlihy's methodology, using a priority queue skew heap as an example in Figure 5. We also provide memory management routines compatible with SMR and similar in functionality to Herlihy's implementation of private node pools. The sequential operations skew_deq and skew_meld need no fundamental modification. Only two parts involve hazard pointers: the outer routines End and Deq, and the *copy* function. Only two hazard pointers are needed. The first continuously points to the anchor node. The setting of the first hazard pointer uses the same technique as that of lines 1–3 of Figure 4.

In the *copy* routine, a thread sets its second hazard pointers to the address of the source node (of the copy operation) then it verifies that the anchor has not been removed. Since no node that was reachable from an anchor node is removed without the anchor node being removed. Since the anchor node is already covered by the first hazard pointer and the source node is already pointed to by the second hazard pointer, the thread can proceed to perform the copy safely. When the copy is done the second hazard pointer can be safely nullified and reused in subsequent calls to the *copy* function. SMR eliminates the need for the two check bits for verifying the consistency of a copied node.

Turek, Shasha, and Prakash [23] and Barnes [3] presented universal methodologies for transforming lock-based implementation (including those using mutiple locks) into lock-free ones. The methodologies translate each atomic statements (one instruction involving shared locations and any number of instructions involving only registers and private variables) and lock operation of of the original algorithm into a continuation. When a thread operates on the object it tries to establish its sequence of continuations as the current operation. If it finds another operation in progress, it tries to help it complete by reading the current step and attempting to execute it. These methodologies apply to both static and dynamic objects. To be SMR-compatible, the programmer must indicate in each continuation if a certain argument is a dynamic node. In such a case, a technique similar to that in the *copy* routine of Figure 5 can be used but instead of verifying the value of the anchor Q, the address of the continuation is verified.

Other universal methodologies provide multi-word or multi-address CAS or LL/SC implementations from single address CAS (or LL/SC) [2, 7, 13, 19, 21]. When applicable to dynamic objects, these techniques are orthogonal to SMR-compatibility, as we have seen that CAS and DCAS can be used to develop algorithms that are compatible with SMR as well as those that are not.

```
// static private variables for methodology
r : result_type;  old_q : *skew_type;
// static private variables for memory management
alloc_ptr,avail_ptr = 0,removed_ptrc : 0 .. MAX_SIZE;
alloc[MAX_SIZE],removed[MAX_SIZE] : *skew_type;

Enq(v:valueType) {
    node ← allocate_node(); node^.value ← v;
    node^.Child[0] ← null; node^.Child[1] ← null;
    while true {
        removed_ptr ← 0; alloc_ptr ← 0;
        old_q ← Q;
        *hp0 ← old_q;
        if old_q ≠ Q continue;
        if !skew_meld(node,old_q,&r.version) continue;
        if CAS(&Q,old_q,r.version) break;
    }
    *hp0 ← null; reset_memory_management();
}

Deq() {
    while true {
        removed_ptr ← 0; alloc_ptr ← 0;
        old_q ← Q;
        *hp0 ← old_q;
        if old_q ≠ Q continue;
        if !skew_deq(old_q) continue;
        if r.value = SKEW_EMPTY break;
        if CAS(&Q,old_q,r.version) break;
    }
    *hp0 ← null; reset_memory_management();
}

copy(q,p:*NodeType) : boolean {
    *hp1 ← q;
    if old_q ≠ Q return false;
    memcpy(p,q,sizeof(NodeType));
    *hp1 ← null;
    if removed_ptr = MAX_SIZE
        {error("exceeded size"); exit;}
    removed[removed_ptr++] ← q;
    return true;
}
```

```
skew_deq(q:*skew_type) : boolean {
    *left,*new_left,*right,buffer : skew_type;

    if q = null{r.value ← SKEW_EMPTY; return true;}
    if !copy(q,&buffer) return false;
    r.value ← buffer.value;
    left ← buffer.Child[0]; right ← buffer.Child[1];
    if left = null{r.version ← right; return true;}
    new_left ← allocate_node();
    if !copy(left,new_left) return false;
    return skew_meld(new_left,right,&r.version);
}

skew_meld(q,qq,*res:*skew_type) : boolean {
    if q = null{ *res ← qq; return true;}
    if qq = null{ *res ← q; return true;}
    p ← allocate_node();
    if !copy(qq,p) return false;
    if q^.Value < p^.Value
        { p1 ← q; p2 ← p;} else { p1 ← p; p2 ← q;}
    toggle ← p1^.toggle;
    p1^.toggle ← !toggle;
    *res ← p1;
    return skew_meld(p2,p1^.Child[toggle],&p1^.Child[toggle]);
}

allocate_node() : *skew_type {
    if alloc_ptr<avail_ptr return alloc[alloc_ptr++];
    if alloc_ptr = MAX_SIZE
        {error("exceeded size"); exit;}
    if alloc[avail_ptr++] ← AllocateNode() = null
        {error("out of memory"); exit;}
    return alloc[alloc_ptr++];
}

reset_memory_management() {
    // free unused allocated nodes
    for i ← alloc_ptr to avail_ptr-1 FreeNode(alloc[i]);
    avail_ptr ← 0;
    // delete removed nodes
    for i ← 0 to removed_ptr-1 DeleteNode(removed[i]);
}
```

Figure 5: SMR-compatible version of Herlihy's methodology's [10] implementation of a lock-free skew heap.

6. CONCLUSIONS

The problem of arbitrary memory reuse for dynamic lock-free object has long obstructed the wide adoption of lock-free synchronization in multiprocessor applications, despite their clear inherent advantages of resilience when faced with thread failures and robust performance when faced with thread delays in comparison to lock-based synchronization.

Prior memory management methods for dynamic lock-free objects fall into four categories. First, those that restrict the reuse of deleted nodes to the same type and require the retention of the semantics and values of some fields indefinitely [12, 25]. Second, those that use the DCAS atomic operation which is not supported on any current system, and use reference counting that results in significant performance overhead [6]. Third, those that require special operating system support for inspecting thread registers and private stack space [7]. Fourth, blocking methods that depend on actions by each thread, such as decrementing an aggregate counter or setting a timestamp, for allowing the reuse of potentially unbounded numbers of deleted nodes. thus allowing the failure or delay of one thread to cause an unbounded number of nodes to be not freed indefinitely [7, 8, 14].

In this paper we presented SMR, a practical and efficient solution for safe memory reclamation for dynamic lock-free objects. It allows arbitrary memory reuse, does not require special hardware as it uses only atomic reads and writes, does not require special operating system support, guarantees an upper bound on the number of deleted nodes not yet freed at all times, and is wait-free. Furthermore, it is only logarithmically contention-sensitive. It guarantees constant amortized time for processing each deleted node until it is freed in the absence of contention, and only $O(\log p)$ time in the case of contention by p threads. Recent performance results [16] indicate that, in practice, the time overhead of using SMR is negligible. Also, it does not require any extra space per pointer or per node.

We demonstrated the ease of complying with the SMR compatibility conditions as we examined most known dynamic lock-free algorithms and methodologies and presented examples for the best known algorithms in each category. We found only two correct algorithms [5, 8] to be incompatible with SMR and freelists, and in each case we found similar or better algorithms for the same object that are compatible with all memory management methods [15, 16]. The performance and qualitative advantages of SMR and freelists make compatibility with them a crucial factor in evaluating currently known and future lock-free algorithms.

Even in cases where memory reclamation is not crucial, SMR still offers a lock-free solution for the ABA problem for pointers to dynamic nodes without the need for any extra space per pointer or per node, applicable to most dynamic lock-free algorithms. It is the only lock-free method that allows these algorithm to avoid the use of extra-width CAS and high-overhead reference counting.

Finally, this paper demonstrates that DCAS is not needed for lock-free memory reclamation.

7. REFERENCES

[1] Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele Jr. DCAS-Based Concurrent Deques. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146, July 2000.

[2] James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems* 10(12): 1317–1332, December 1999.

[3] Greg Barnes. A Method for Implementing Lock-Free Shared-Data Structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June–July 1993.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[5] David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Martin, Nir N. Shavit, and Guy L. Steele Jr. Even Better DCAS-Based Concurrent Deques. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 59–73, October 2000.

[6] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-Free Reference Counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, August 2001.

[7] Michael B. Greenwald. *Non-Blocking Synchronization and System Design.* Ph.D. thesis, Stanford University Technical Report STAN-CS-TR-99-1624, August 1999.

[8] Timothy L. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 300–314, October 2001.

[9] Maurice P. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13(1): 124–149, January 1991.

[10] Maurice P. Herlihy. A Methodology for Implementing Highly Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 15(5): 745–770, November 1993.

[11] Maurice P. Herlihy and J. Eliot B. Moss. Lock-Free garbage Collection for Multiprocessors. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 229–236, July 1991.

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

[13] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.

[14] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems*, October 1998.

[15] Maged M. Michael. Dynamic Lock-Free Deque Algorithm Using Single-Address Double-Word CAS. Research Report RC 22401, IBM T. J. Watson Research Center, April 2002.

[16] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, August 2002.

[17] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

[18] Mark Moir. Practical Implementations of Non-Blocking Synchronization Primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, August 1997.

[19] Mark Moir. Laziness pays! Using lazy Synchronization Mechanisms to Improve Non-Blocking Constructions. *Distributed Computing* 14(4): 193–204, 2001.

[20] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *IEEE Transactions on Computers* 43(5): 548–559, May 1994.

[21] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing* 10(2): 99–116, 1997.

[22] R. Kent Treiber. Systems Programing: Coping with Parallelism. Research Report RJ 5118, IBM Almaden Research Center, April 1986.

[23] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, pages 212–222, June 1992.

[24] John D. Valois. Implementing Lock-Free Queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 64–69, October 1994.

[25] John D. Valois. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.