Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects

Maged M. Michael

Abstract—Lock-free objects offer significant performance and reliability advantages over conventional lock-based objects. However, the lack of an efficient portable lock-free method for the reclamation of the memory occupied by dynamic nodes removed from such objects is a major obstacle to their wide use in practice. This paper presents *hazard pointers*, a memory management methodology that allows memory reclamation for arbitrary reuse. It is very efficient, as demonstrated by our experimental results. It is suitable for user-level applications—as well as system programs—without dependence on special kernel or scheduler support. It is wait-free. It requires only single-word reads and writes for memory access in its core operations. It allows reclaimed memory to be returned to the operating system. In addition, it offers a lock-free solution for the ABA problem using only practical single-word instructions. Our experimental results on a multiprocessor system show that the new methodology offers equal and, more often, significantly better performance than other memory management methods, in addition to its qualitative advantages regarding memory reclamation and independence of special hardware support. We also show that lock-free implementations of important object types, using hazard pointers, offer comparable performance to that of efficient lock-based implementations under no contention and no multiprogramming, and outperform them by significant margins under moderate multiprogramming and/or contention, in addition to guaranteeing continuous progress and availability, even in the presence of thread failures and arbitrary delays.

Index Terms—Lock-free, synchronization, concurrent programming, memory management, multiprogramming, dynamic data structures.

1 Introduction

A shared object is *lock-free* (also called nonblocking) if it guarantees that whenever a thread executes some finite number of steps toward an operation on the object, *some* thread (possibly a different one) must have made progress toward completing an operation on the object, during the execution of these steps. Thus, unlike conventional lock-based objects, lock-free objects are immune to deadlock when faced with thread failures, and offer robust performance, even when faced with arbitrary thread delays.

Many algorithms for lock-free dynamic objects have been developed, e.g., [11], [9], [23], [21], [4], [5], [16], [7], [25], [18]. However, a major concern regarding these objects is the reclamation of the memory occupied by removed nodes. In the case of a lock-based object, when a thread removes a node from the object, it is easy to guarantee that no other thread will subsequently access the memory of that node, before it is reused or reallocated. Consequently, it is usually safe for the removing thread to reclaim the memory occupied by the removed node (e.g., using `free`) for arbitrary future reuse by the same or other threads (e.g., using `malloc`).

This is not the case for a typical lock-free dynamic object, when running in programming environments without support for automatic garbage collection. In order to guarantee lock-free progress, each thread must have unrestricted opportunity to operate on the object, at any time. When a thread removes a node, it is possible that some other contending thread—in the course of its lock-free operation—has earlier read a reference to that node, and is about to access its contents. If the removing thread were to reclaim the removed node for arbitrary reuse, the contending thread might corrupt the object or some other object that happens to occupy the space of the freed node, return the wrong result, or suffer an access error by dereferencing an invalid pointer value. Furthermore, if reclaimed memory is returned to the operating system (e.g., using `munmap`), access to such memory locations can result in fatal access violation errors. Simply put, the memory reclamation problem is how to allow the memory of removed nodes to be freed (i.e., reused arbitrarily or returned to the OS), while guaranteeing that no thread accesses free memory, and how to do so in a lock-free manner.

Prior methods for allowing node reuse in dynamic lock-free objects fall into three main categories. 1) The IBM tag (update counter) method [11], which hinders memory reclamation for arbitrary reuse and requires double-width instructions that are not available on 64-bit processors. 2) Lock-free reference counting methods [29], [3], which are inefficient and use unavailable strong multiaddress atomic primitives in order to allow memory reclamation. 3) Methods that depend on aggregate reference counters or per-thread timestamps [13], [4], [5]. Without special scheduler support, these methods are blocking. That is, the failure or delay of even one thread can prevent an aggregate reference counter from reaching zero or a timestamp from advancing and, hence, preventing the reuse of unbounded memory.

This paper presents *hazard pointers*, a methodology for memory reclamation for lock-free dynamic objects. It is efficient; it takes constant expected amortized time per retired node (i.e., a removed node that is no longer needed by the removing thread). It offers an upper bound on the total number of retired nodes that are not yet eligible for reuse, regardless of thread failures or delays. That is, the failure or delay of any number of threads can prevent only a

• *The author is with the IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598. E-mail: magedm@us.ibm.com.*

Manuscript received 15 Apr. 2003; revised 22 Oct. 2003; accepted 2 Jan. 2004. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0058-0403.

bounded number of retired nodes from being reused. The methodology does not require the use of double-width or strong multiaddress atomic primitives. It uses only single-word reads and writes for memory access in its core operations. It is wait-free [8], i.e., progress is guaranteed for active threads individually, not just collectively; thus, it is also applicable to wait-free algorithms without weakening their progress guarantee. It allows reclaimed memory to be returned to the operating system. It does not require any special support from the kernel or the scheduler.

The core idea is to associate a number (typically one or two) of single-writer multireader shared pointers, called *hazard pointers*, with each thread that intends to access lock-free dynamic objects. A hazard pointer either has a null value or points to a node that may be accessed later by that thread without further validation that the reference to the node is still valid. Each hazard pointer can be written only by its owner thread, but can be read by other threads.

The methodology requires lock-free algorithms to guarantee that no thread can access a dynamic node at a time when it is possibly removed from the object, unless at least one of the thread's associated hazard pointers has been pointing to that node continuously, from a time when the node was guaranteed to be reachable from the object's roots. The methodology prevents the freeing of any retired node continuously pointed to by one or more hazard pointers of one or more threads from a point prior to its removal.

Whenever a thread retires a node, it keeps the node in a private list. After accumulating some number R of retired nodes, the thread scans the hazard pointers of other threads for matches for the addresses of the accumulated nodes. If a retired node is not matched by any of the hazard pointers, then it is safe for this node to be reclaimed. Otherwise, the thread keeps the node until its next scan of the hazard pointers.

By organizing a private list of snapshots of nonnull hazard pointers in a hash table that can be searched in constant expected time, and if the value of R is set such that $R = H + \Omega(H)$, where H is the total number of hazard pointers, then the methodology is guaranteed in every scan of the hazard pointers to identify $\Theta(R)$ nodes as eligible for arbitrary reuse, in $O(R)$ expected time. Thus, the expected amortized time complexity of processing each retired node until it is eligible for reuse is constant.

Note that a small number of hazard pointers per thread can be used to support an arbitrary number of objects as long as that number is sufficient for supporting each object individually. For example, in a program where each thread may operate arbitrarily on hundreds of shared objects that each requires up to two hazard pointers per thread (e.g., hash tables [25], FIFO queues [21], LIFO stacks [11], linked lists [16], work queues [7], and priority queues [9]), only a total of two hazard pointers are needed per thread.

Experimental results on an IBM RS/6000 multiprocessor system show that the new methodology, applied to lock-free implementations of important object types, offers equal and, more often, significantly better performance than other memory management methods, in addition to its qualitative advantages regarding memory reclamation and independence of special hardware support. We also show that lock-free implementations of important object types, using hazard pointers, offer comparable performance to that of efficient lock-based implementations under no contention and no multiprogramming, and outperform them by significant margins under moderate multiprogramming and/or contention, in addition to guaranteeing continuous progress and availability even in the presence of thread failures and arbitrary delays.

The rest of this paper is organized as follows: In Section 2, we discuss the computational model for our methodology and memory management issues for lock-free objects. In Section 3, we present the hazard pointer methodology. In Section 4, we discuss applying hazard pointers to lock-free algorithms. In Section 5, we present our experimental performance results. In Section 6, we discuss related work and summarize our results.

2 Preliminaries

2.1 The Model

The basic computational model for our methodology is the asynchronous shared memory model. Formal descriptions of this model appeared in the literature, e.g., [8]. Informally, in this model, a set of threads communicate through primitive memory access operations on a set of shared memory locations. Threads run at arbitrary speeds and are subject to arbitrary delays. A thread makes no assumptions about the speed or status of any other thread. That is, it makes no assumptions about whether another thread is active, delayed, or crashed, and the time or duration of its suspension, resumption, or failure. If a thread crashes, it halts execution instantaneously.

A shared object occupies a set of shared memory locations. An object is an instance of an implementation of an abstract object type, that defines the semantics of allowable operations on the object.

2.2 Atomic Primitives

In addition to atomic reads and writes, primitive operations on shared memory locations may include stronger atomic primitives such as compare-and-swap (CAS) and the pair load-linked/store-conditional (LL/SC). CAS takes three arguments: the address of a memory location, an expected value, and a new value. If and only if the memory location holds the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. That is, $CAS(addr, exp, new)$ performs the following atomically:

$$\{if\ (*addr \neq exp)\ return\ false; *addr \leftarrow new;\ return\ true;\}.$$

LL takes one argument: the address of a memory location, and returns its contents. SC takes two arguments: the address of a memory location and a new value. Only if no other thread has written the memory location since the current thread last read it using LL, the new value is written to the memory location, atomically. A Boolean return value indicates whether the write occurred. An associated instruction, Validate (VL), takes one argument: the address of a memory location, and returns a Boolean value that indicates whether any other thread has written the memory location since the current thread last read it using LL.

For practical architectural reasons, none of the architectures that support LL/SC (Alpha, MIPS, PowerPC) support VL or the ideal semantics of LL/SC as defined above. None allow nesting or interleaving of LL/SC pairs, and most prohibit any memory access between LL and SC. Also, all such architectures, occasionally—but not infinitely often—allow SC to fail spuriously; i.e., return false even when the

```
// Hazard pointer record
   structure HPRecType
        { HP[K]:*NodeType; Next:*HPRecType; }
// The header of the HPRec list
   HeadHPRec : *HPRecType;
// Per-thread private variables
   rlist : listType; // initially empty
   rcount : integer; // initially 0
```

Fig. 1. Types and structures.

memory location was not written by other threads since it was last read by the current thread using LL. For all the algorithms presented in this paper, $CAS(addr, exp, new)$ can be implemented using restricted LL/SC as follows:

$$\{do\ \{if\ (LL(addr) \neq exp)\ return\ false;\}$$
$$until\ SC(addr, new);\ return\ true;\}.$$

Most current mainstream processor architectures support either CAS or restricted LL/SC on aligned single words. Support for CAS and LL/SC on aligned double-words is available on most 32-bit architectures (i.e., support for 64-bit instructions), but not on 64-bit architecture (i.e., no support for 128-bit instructions).

2.3 The ABA problem

A different but related problem to memory reclamation is the ABA problem. It affects almost all lock-free algorithms. It was first reported in the documentation of CAS on the IBM System 370 [11]. It occurs when a thread reads a value A from a shared location, and then other threads change the location to a different value, say B, and then back to A again. Later, when the original thread checks the location, e.g., using read or CAS, the comparison succeeds, and the thread erroneously proceeds under the assumption that the location has not changed since the thread read it earlier. As a result, the thread may corrupt the object or return a wrong result.

The ABA problem is a fundamental problem that must be prevented regardless of memory reclamation. Its relation to memory reclamation is that solutions of the latter problem, such as automatic garbage collection (GC) and the new methodology, often prevent the ABA problem as a side-effect with little or no additional overhead.

This is true for most lock-free dynamic objects. But, it should be noted that a common misconception is that GC inherently prevents the ABA problem in all cases. However, consider a program that moves dynamic nodes back and forth between two lists (e.g., LIFO stacks [11]). The ABA problem is possible in such a case, even with perfect GC.

The new methodology is as powerful as GC with respect to ABA prevention in lock-free algorithms. That is, if a lock-free algorithm is ABA-safe under GC, then applying hazard pointers to it makes it ABA-safe without GC. As we discuss in a recent report [19], lock-free algorithms can *always* be made ABA-safe under GC, as well as using hazard pointers in the absence of GC. In the rest of this paper, when discussing the use of hazard pointers for ABA prevention in the absence of support for GC, we assume that lock-free algorithms are already ABA-safe under GC.

```
RetireNode(node:*NodeType) {
   rlist.push(node);
   rcount++;
   if (rcount ≥ R)
      Scan(HeadHPRec);
}
```

Fig. 2. The RetireNode routine.

3 The Methodology

The new methodology is primarily based on the observation that, in the vast majority of algorithms for lock-free dynamic objects, a thread holds only a small number of references that may later be used without further validation for accessing the contents of dynamic nodes, or as targets or expected values of ABA-prone atomic comparison operations.

The core idea of the new methodology is associating a number of single-writer multireader shared pointers, called *hazard pointers*, with each thread that may operate on the associated objects. The number of hazard pointers per thread depends on the algorithms for associated objects and may vary among threads depending on the types of objects they intend to access. Typically, this number is one or two. For simplicity of presentation, we assume that each thread has the same number K of hazard pointers.

The methodology communicates with the associated algorithms only through hazard pointers and a procedure $RetireNode$ that is called by threads to pass the addresses of retired nodes. The methodology consists of two main parts: the algorithm for processing retired nodes, and the condition that lock-free algorithms must satisfy in order to guarantee the safety of memory reclamation and ABA prevention.

3.1 The Algorithm

Fig. 1 shows the shared and private structures used by the algorithm. The main shared structure is the list of hazard pointer (HP) records. The list is initialized to contain one HP record for each of the N participating threads. The total number of hazard pointers is $H = NK$.[1] Each thread uses two static private variables, $rlist$ (retired list) and $rcount$ (retired count), to maintain a private list of retired nodes.

Fig. 2 shows the $RetireNode$ routine, where the retired node is inserted into the thread's list of retired nodes and the length of the list is updated. Whenever the size of a thread's list of retired nodes reaches a threshold R, the thread scans the list of hazard pointers using the $Scan$ routine. R can be chosen arbitrarily. However, in order to achieve a constant expected amortized processing time per retired node, R must satisfy $R = H + \Omega(H)$.

Fig. 3 shows the $Scan$ routine. A scan consists of two stages. The first stage involves scanning the HP list for nonnull values. Whenever a nonnull value is encountered, it is inserted in a local list $plist$, which can be implemented as a hash table. The second stage of $Scan$ involves checking each node in $rlist$ against the pointers in $plist$. If the lookup yields no match, the node is identified to be ready for arbitrary reuse. Otherwise, it is retained in $rlist$ until the

1. As discussed in Section 3.2, the algorithm can be extended such that the values of N and H do not need to be known in advance, and threads can join and leave the system dynamically and allocate and deallocate hazard pointers dynamically.

```
Scan(head:*HPRecType) {
  // Stage 1: Scan HP list and insert non-null values in plist
  plist.init();
  hprec ← head;
  while (hprec ≠ null) {
    for (i ← 0 to K-1) {
      hptr ← hprec^.HP[i];
      if (hptr ≠ null)
        plist.insert(hptr);
    }
    hprec ← hprec^.Next;
  }
  // Stage 2: Search plist
  tmplist ← rlist.popAll();
  rcount ← 0;
  node ← tmplist.pop();
  while (node ≠ null) {
    if (plist.lookup(node)) {
      rlist.push(node);
      rcount++;
    } else {
      PrepareForReuse(node);
    }
    node ← tmplist.pop();
  }
  plist.free();
}
```

Fig. 3. The Scan routine.

next scan by the current thread. Insertion and lookup in $plist$ take constant expected time.

Alternatively, if a lower worst-case—instead of average—time complexity is desired, $plist$ can be implemented as a balanced search tree with $O(\log p)$ insertion and lookup time complexities, where p is the number of nonnull hazard pointers encountered in Stage 1 of $Scan$. In such a case, the amortized time complexity per retired node is $O(\log p)$.

In practice, for simplicity and speed, we recommend implementing $plist$ as an array and sorting it at the end of Stage 1 of $Scan$, and then using binary search in Stage 2. We use the latter implementation for our experiments in Section 5. We omit the algorithms for hash tables, balanced search trees, sorting, and binary search, as they are well-known sequential algorithms [2].

The task of the memory reclamation method in the context of this paper is to determine *when* a retired node is eligible for reuse safely while allowing memory reclamation. Thus, the definition of the $PrepareForReuse$ routine is open for several implementation options and is not an integral part of this methodology. An obvious implementation of that routine is to reclaim the node immediately for arbitrary reuse using the standard library call for memory deallocation, e.g., `free`. Another possibility—in order to reduce the overhead of calling `malloc` and `free` for every node allocation and deallocation—is that each thread can maintain a limited size private list of free nodes. When a thread runs out of private free nodes, it allocates new nodes, and when it accumulates too many private free nodes, it deallocates the excess nodes.

The algorithm is wait-free; it takes $O(R)$ expected time—or $O(R \log p)$ worst-case time if a logarithmic search structure is used—to identify $\Theta(R)$ retired nodes as eligible for arbitrary reuse. It only uses single-word reads and writes. It offers an upper bound NR on the number of retired nodes that are not yet eligible for reuse, even if some or all threads are delayed or have crashed.

3.2 Algorithm Extensions

The following are optional extensions to the core algorithm that enhance the methodology's flexibility.

If the maximum number N of participating threads is not known before hand, we can add new HP records to the HP list using a simple push routine [11]. Note that such a routine is wait-free, as the maximum number of threads is finite. This can be useful also, if it is desirable for threads to be able to allocate additional hazard pointers dynamically.

In some applications, threads are created and retired dynamically. In such cases, it is desirable to allow HP records to be reused. Adding a Boolean flag to each HP record can serve as an indicator if the HP record is in use or available for reuse. Before retiring, a thread can clear the flag, and when a new thread is created, it can search the HP list for an available HP record and acquire it using test-and-set (TAS). If no HP records are available, a new one can be added as described above.

Since a thread may have leftover retired nodes not yet identified as eligible for reuse, two fields can be added to the HP record structure so that a retiring thread can pass the values of its $rlist$ and $rcount$ variables to the next thread that inherits the HP record.

Furthermore, it may be desirable to guarantee that every node that is eligible for reuse is eventually freed, barring thread failures. To do so, after executing $Scan$, a thread executes a $HelpScan$, where it checks every HP record. If an HP record is inactive, the thread locks it using TAS and pops nodes from its $rlist$. Whenever, the thread accumulates R nodes, it performs a $Scan$. Therefore, even if a thread retires leaving behind an HP record with a nonempty $rlist$ and its HP record happens not to be reused, the nodes in the $rlist$ will still be processed by other threads performing $HelpScan$.

Fig. 4 shows a version of the algorithm that incorporates the above mentioned extensions. The algorithm is still wait-free, and only single-word instructions are used.

3.3 The Condition

For a correct algorithm for a dynamic lock-free object to use the new methodology for memory reclamation and ABA prevention, it must satisfy a certain condition. When a thread assigns a reference (i.e., a node's address) to one of its hazard pointers, it basically announces to other threads that it may use that reference in a hazardous manner (e.g., access the contents of the node without further validation of the reference), so that other threads will refrain from reclaiming or reusing the node until the reference is no longer hazardous. This announcement (i.e., setting the hazard pointer) must take place before the node is retired and the hazard pointer must continue to hold that reference until the reference is no longer hazardous.

```
structure HPRecType { HP[K]:*NodeType; Next:*HPRecType;
      Active:Boolean; rlist:listType; rcount:integer; }
// Shared variables
   HeadHPRec : *HPRecType; // initially null
   H : integer; // initially 0
// Per-thread private variable
   myhprec : *HPRecType; // initially null

AllocateHPRec() {
   // First try to reuse a retired HP record
   for (hprec ← HeadHPRec; hprec ≠ null; hprec ← hprec^.Next) {
      if (hprec^.Active) continue;
      // TAS(addr) ≡ ¬CAS(addr,false,true)
      if TAS(&hprec^.Active) continue;
      // Succeeded in locking an inactive HP record
      myhprec ← hprec;
      return;
   }
   // No HP records available for reuse
   // Increment H, then allocate a new HP and push it
   do { // wait-free - max. num. of threads is finite
      oldcount ← H;
   } until CAS(&H,oldcount,oldcount+K);
   // Allocate and push a new HP record
   hprec ← NewHPRec();
   Initialize the fields of the new HP record.
   do { // wait-free - max. num. of threads is finite
      oldhead ← HeadHPRec;
      hprec^.Next ← oldhead;
   } until CAS(&HeadHPRec,oldhead,hprec);
   myhprec ← hprec;
}

RetireHPRec() {
   for (i ← 0 to K-1) myhprec^.HP[i] ← null;
   myhprec^.Active ← false;
}

RetireNode(node:*NodeType) {
   myhprec^.rlist.push(node);
   myhprec^.rcount++;
   head ← HeadHPRec;
   if (myhprec^.rcount ≥ R(H)) {  // R(H) = H + Ω(H)
      Scan(head);
      HelpScan();
   }
}

// Scan() is the same as in Figure 3 except that rlist and rcount
// are fields of *myhprec instead of being private variables.

HelpScan() {
   for (hprec ← HeadHPRec; hprec ≠ null; hprec ← hprec^.Next) {
      if (hprec^.Active) continue;
      if TAS(&hprec^.Active) continue;
      while (hprec^.rcount > 0) {
         node ← hprec^.rlist.pop();
         hprec^.rcount--;
         myhprec^.rlist.push(node);
         myhprec^.rcount++;
         head ← HeadHPRec;
         if (myhprec^.rcount ≥ R(H))
            Scan(head);
      }
      hprec^.Active ← false;
   }
}
```

Fig. 4. Algorithm extensions.

For a formal description of the condition, we first define some terms:

Node: We use the term *node* to describe a range of memory locations that at some time may be viewed as a logical entity either through its actual use in an object that uses hazard pointers, or from the point of view of a participating thread. Thus, it is possible for multiple nodes to overlap physically, but still be viewed as distinct logical entities.

At any time t, each *node* n is in one of the following states:

1. *Allocated*: n is allocated by a participating thread, but not yet inserted in an associated object.
2. *Reachable*: n is reachable by following valid pointers starting from the roots of an associated object.
3. *Removed*: n is no longer reachable, but may still be in use by the removing thread.
4. *Retired*: n is already removed and consumed by the removing thread, but not yet free.
5. *Free*: n's memory is available for allocation.
6. *Unavailable*: all or part of n's memory is used by an unrelated object.
7. *Undefined*: n's range of memory locations is not currently viewed as a node.

Own: A thread j *owns* a node n at time t, iff at t, n is *allocated*, *removed*, or *retired* by j. Each node can have at most one *owner*. The *owner* of an *allocated* node is the thread that allocated it (e.g., by calling `malloc`). The *owner* of a *removed* node is the thread that executed the step that removed it from the object (i.e., changed its state from *reachable* to *removed*). The *owner* of a *retired* node is the same one that removed it.

Safe: A node n is *safe* for a thread j at time t, iff at time t, either n is *reachable*, or j owns n.

Possibly unsafe: A node is *possibly unsafe* at time t from the point of view of thread j, if it is impossible solely by examining j's private variables and the semantics of the algorithm to determine definitely in the affirmative that at time t the node is *safe* for j.

Access hazard: A step s in thread j's algorithm is an *access hazard* iff it may result in access to a node that is *possibly unsafe* for j at the time of its execution.

ABA hazard: A step s in thread j's algorithm is an *ABA hazard* iff it includes an ABA-prone comparison that involves a dynamic node that is *possibly unsafe* for j at the time of the execution of s, such that either 1) the node's address—or an arithmetic variation of it—is an expected value of the ABA-prone comparison, or 2) a memory location contained in the dynamic node is the target of the ABA-prone comparison.

Access-hazardous reference: A thread j holds an *access-hazardous reference* to a node n at time t, iff at time t one or more of j's private variables holds n's address or an

arithmetic variation of it, and j is guaranteed—unless it crashes—to reach an *access hazard* s that uses n's address hazardously, i.e., accesses n when n is *possibly unsafe* for j.

ABA-hazardous reference: A thread j holds an *ABA-hazardous reference* to a node n at time t, iff at time t, one or more of j's private variables holds n's address or a mathematical variation of it, and j is guaranteed—unless it crashes—to reach an *ABA hazard* s that uses n's address hazardously.

Hazardous reference: A reference is *hazardous* if it is *access-hazardous* and/or *ABA-hazardous*.

Informally, a hazardous reference is an address that without further validation of safety will be used later in a hazardous manner, i.e., to access possibly unsafe memory and/or as a target address or an expected value of an ABA-prone comparison.

A thread that holds a reference to a node uses hazard pointers to announce to other threads that it may use the reference later without further validation in a hazardous step. However, this announcement is useless if it happens after the reference is already *hazardous*, or in other words, after the node is *possibly unsafe*, since another thread might have already removed the node, and then scanned the HP list and found no match for that node. Therefore, the condition that an associated algorithm must satisfy is that whenever a thread is holding a hazardous reference to a node, it must be the case that at least one of the thread's hazard pointers has been continuously holding that reference from a time when the node was definitely *safe* for the thread. Note that this condition implies that no thread can create a new hazardous reference to a node while it is retired.

Formally, the condition is as follows, where HP_j is the set of thread j's hazard pointers:

$$
\begin{array}{l}
\forall \text{ times } t, \text{ threads } j, \text{ and nodes } n, \\
\quad (\text{at } t,\ j \text{ holds a } \textit{hazardous reference} \text{ to } n) \Rightarrow \\
\quad (\exists hp \in HP_j, t' \leq t :: \\
\qquad (\text{at } t',\ n \text{ is } \textit{safe} \text{ for } j) \wedge \\
\qquad (\forall \text{ times during } [t', t],\ hp = \&n)).
\end{array}
$$

3.4 Correctness

The following lemmas and theorem are contingent on satisfying the condition in Section 3.3.

Lemma 1. *$\forall$ times t, threads j, and nodes n, ($\forall hp \in HP_j$, $t' \leq t$, ($\forall$ times during $[t', t]$, n is not safe for j) $\wedge$ ($\exists t'' \in [t', t] ::$ at $t'', hp \neq \&n$)) $\Rightarrow$ (at t, j does not hold a hazardous reference to n).*

Informally, if a scan of the hazard pointers of a thread j finds no match for a retired node n, then it must be the case that j holds no hazardous reference to n at the end of the scan.

Proof sketch: For a proof by contradiction, assume that the lemma is false, i.e., the antecedent of the implication is true and the consequent is false. Then, at t, j holds a hazardous reference to n. Then, by the condition in Section 3.3, there must be some time t_0 when n was safe for j, but t_0 must be before t' because we already assume that n is not safe for j during $[t', t]$. Also by the condition in Section 3.3, there must be at least one of j's hazard pointers that is pointing to n continuously during $[t_0, t]$. But, this contradicts the initial assumption that for each of j's hazard pointers, there is some time during $[t', t]$ (and, hence, during $[t_0, t]$) when the hazard pointer does not point to n. Therefore, the initial assumption must be false and the lemma is true. □

Lemma 2. *$\forall$ times t, threads j, and nodes n, (at t, n is identified in stage 2 of* Scan *as eligible for reuse) $\Rightarrow$ ($\forall hp \in HP_j, \exists t' \in [t_0, t] ::$ at $t', hp \neq \&n$), where t_0 is the start time of the current execution of* Scan.

Informally, a retired node is identified as eligible for reuse in stage 2 of *Scan* only after a scan of the hazard pointers of participating threads finds no match.

Proof sketch: For a proof by contradiction, assume that the lemma is false. Then, at least one of j's hazard pointers was continuously pointing to n since t_0. Then, by the flow of control (of stage 1), at the end of stage 1, *plist* must contain a pointer to n. Then, by the flow of control (of stage 2), n is not identified as eligible for reuse. A contradiction to the initial assumption. □

Theorem 1. *$\forall$ times t, threads j, and nodes n, (at t, n is identified in stage 2 of* Scan *as eligible for reuse) $\Rightarrow$ (at t, j does not hold a hazardous reference to n).*

Informally, if *Scan* identifies a node as eligible for reuse, then it must be the case that no thread holds a hazardous reference to it.

Proof sketch: If j is the thread executing *Scan*, the theorem is trivially true. Consider the case where j is a different thread. Assume that at t, n is identified in stage 2 of *Scan* as eligible for reuse. Then, by the definition of *safe*, n is not safe for j since the beginning of *Scan*, and by Lemma 2, for each hazard pointer, there was a time during the current execution of *Scan* when the hazard pointer did not point to n. Then, by Lemma 1, at t, j does not hold a hazardous reference to n. □

By the definition of *access-hazardous reference* and Theorem 1, it follows that the hazard pointer methodology (i.e., algorithm and condition) guarantees that while a node is free or unavailable, no thread accesses its contents, i.e., the hazard pointer methodology guarantees safe memory reclamation.

By the definition of *ABA-hazardous reference* and Theorem 1, it follows that the hazard pointer methodology guarantees that, while a node is free or unavailable, no thread can hold a reference to it without further validation with the intention to use that reference as a target or an expected value of an ABA-prone comparison. This is the same guarantee offered by GC with respect to the ABA problem.

4 Applying Hazard Pointers

This section discusses the methodology for adapting existing lock-free algorithms to the condition in Section 3.3. The following is an outline:

1. Examine the target algorithm as follows:
   a. Identify *hazards* and the *hazardous references* they use.
   b. For each distinct *hazardous reference*, determine the point where it is created and the last *hazard* that uses it. The period between these two

```
structure NodeType { Data:DataType; Next:*NodeType; }
// Shared variables
Head,Tail:*NodeType;
// Initially both Head and Tail point to a dummy node

Enqueue(data:DataType) {
 1:  node ← NewNode();
 2:  node^.Data ← data;
 3:  node^.Next ← null;
     while true {
 4:    t ← Tail;
 5:    next ← t^.Next;
 6:    if (Tail ≠ t) continue;
 7:    if (next ≠ null) { CAS(&Tail,t,next); continue; }
 8:    if CAS(&t^.Next,null,node) break;
     }
 9:  CAS(&Tail,t,node);
}

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

Fig. 5. A memory-management-oblivious lock-free queue algorithm.

points is when a hazard pointer needs to be dedicated to that reference.

c. Compare the periods determined in the previous step for all *hazardous references*, and determine the maximum number of distinct references that can be hazardous—for the same thread—at the same time. This is the maximum number of hazard pointers needed per thread.

2. For each hazardous reference, insert the following steps in the target algorithm after the creation of the reference and before any of the hazards that use it:

   a. Write the address of the node that is the target of the reference to an available hazard pointer.
   b. Validate that the node is *safe*. If the validation succeeds, follow the normal flow of control of the target algorithm. Otherwise, skip over the hazards and follow the path of the target algorithm when conflict is detected, i.e., try again, backoff, exit loop, etc. This step is needed, as the node might have been already removed before the previous step was executed.

We applied hazard pointers to many algorithms, e.g., [11], [9], [23], [28], [21], [26], [4], [16], [7], [6], [18], for the purposes of allowing memory reclamation and ABA prevention. We use several algorithms for important object types to demonstrate the application of the steps described above.

Note that, while applying these steps is easy for algorithm designers, these steps are not readily applicable automatically (e.g., by a compiler). For example, it is not clear if a compiler can determine if a node is no longer *reachable*. Identifying ABA hazards is even more challenging for a compiler, as the ABA problem is a subtle problem that involves the implicit intentions of the algorithm designer.

4.1 FIFO Queues

Fig. 5 shows a version of Michael and Scott's [21] lock-free FIFO queue algorithm, stripped of memory management code. The algorithm demonstrates typical use of hazard pointers. We use it as the main case study.

Briefly, the algorithm represents the queue as a singly linked list with a dummy node at its head. If an enqueue operation finds the *Tail* pointer pointing to the last node, it links the new node at the end of the list and then updates *Tail*. Otherwise, it updates *Tail* first, and then tries to enqueue the new node. A dequeue operation swings the *Head* pointer after reading the data from the second node in the list, while ensuring that *Tail* will not lag behind *Head*. As it is the case with any lock-free object, if a thread is delayed at any point while operating on the object and leaves it in an unstable state, any other thread can take the object to a stable state and then proceed with its own operation.

First, we examine Fig. 5 to identify *hazards* and *hazardous references*, starting with the enqueue routine:

1. The node accesses in lines 2 and 3 are not hazardous because, at that time, the node **node* is guaranteed to be *allocated*—and, hence, *owned*—by the current thread.
2. The node access in line 5 is an *access hazard* because the node **t* may have been removed and reclaimed by another thread after the current thread executed line 4.
3. The function of what we call a *validation condition* in line 6 is to guarantee that the thread proceeds to line 7, only if at the time of reading $t\text{.}Next$ in line 5, *Tail* was equal to t. Without this guarantee, the queue may be corrupted. It is ABA-prone and, hence, it is an *ABA hazard*.
4. The CAS in line 7 is an *ABA hazard*.
5. The CAS in line 8 is both an *access hazard* and an *ABA hazard*.
6. The CAS in line 9 is an *ABA hazard*.

Therefore, the reference to t is hazardous between lines 4 and 9. Only one hazard pointer is sufficient since there is only one hazardous reference at any time in this routine.

Fig. 6 shows the enqueue routine augmented with a hazard pointer and code guaranteeing safe memory reclamation and ABA prevention. The technique in lines 4a and 4b is the fundamental mechanism for applying hazard pointers to target algorithm, as shown in the rest of this section.

After creating a reference (line 4) that is identified as hazardous in the memory-management-oblivious algorithm, the thread takes the following steps: 1) It assigns the address of the referenced node to a hazard pointer

```
// hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs

Enqueue(data:DataType) {
   1: node ← NewNode();
   2: node^.Data ← data;
   3: node^.Next ← null;
      while true {
   4:    t ← Tail;
  4a:    *hp0 ← t;
  4b:    if (Tail ≠ t) continue;
   5:    next ← t^.Next;
   6:    if (Tail ≠ t) continue;
   7:    if (next ≠ null) { CAS(&Tail,t,next); continue;}
   8:    if CAS(&t^.Next,null,node) break;
      }
   9: CAS(&Tail,t,node);
}
```

Fig. 6. Enqueue routine with hazard pointers.

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

Fig. 7. Dequeue routine with hazard pointers.

(line 4a). 2) Then, it validates that that node is safe (line 4b). If not, it skips over the hazards and tries again.

The second step is needed, since it is possible that after line 4 and before line 4a, some other thread had removed the node *t and checked ***hp0** and concluded that the current thread does not hold hazardous references to *t. Line 4b serves to guarantee that at that point, which is before any hazards, ***hp0** already points to *t, and that *t is safe.

It is possible that the node *t was removed and then reinserted by other threads between the current thread's execution of lines 4 and 4b. However, this is acceptable, as it does not violate the condition in Section 3.3. After executing line 4 and before line 4b, the reference t is not hazardous as the thread is not guaranteed to reach a hazard (lines 5, 6, 7, 8, or 9). It starts to be hazardous only upon the success of the validation condition in line 4b. But, at that point, ***hp0** already covers t and continues to do so until t ceases to be hazardous (after line 9).

Next, we examine the dequeue routine in Fig. 5:

1. The node access in lines 13 is an *access hazard* using the reference h.
2. The validation condition in line 14 is an *ABA hazard* using h.
3. The CAS in line 16 is an *ABA hazard* using t. However, since t is guaranteed to be equal to h at that point, then it is covered if h is covered.
4. The node access in line 17 is an *access hazard* using *next*.
5. The CAS in line 18 is an *ABA hazard* using h.

Therefore, h is hazardous between lines 11 and 18, and *next* is hazardous between lines 13 and 17. Since the two periods overlap, two hazard pointers are needed.

Fig. 7 shows the dequeue routine augmented with hazard pointers. For the reference h, lines 11a and 11b employ the same technique as lines 4a and 4b in Fig. 6. For the reference *next*, no extra validation is needed. The validation condition in line 14 (from the original algorithm) guarantees that ***hp1** equals *next* from a point when **next* was safe. The semantics of the algorithm guarantee that the node **next* cannot be *removed* at line 14 unless its predecessor **h* has been removed first and then reinserted after line 13 and before line 14. But, this is impossible since ***hp0** covers h from before line 11b until after line 18.

Note that hazard pointers allow some optimizations to the original algorithm [21] that are not safe when only the IBM tag method [11] is used for ABA prevention. Line 6 can be removed from the enqueue routine and line 17 can be moved out of the main loop in the dequeue routine.

If only safe memory reclamation is desired—and the ABA problem is not a concern, e.g., assuming support for ideal LL/SC/VL—then only one hazard pointer is sufficient for protecting both **h* and **next* in the *Dequeue* routine. We leave this as an exercise for the reader.

4.2 LIFO Stacks

Fig. 8 shows a lock-free stack based on the IBM freelist algorithm [11] augmented with hazard pointers. In the push routine, the node accesses in lines 2 and 4 are not *hazardous* (as **node* is *owned* by the thread at the time), and the CAS in line 5 is not ABA-prone (as the change of *Top* between lines 3 and 5 can never lead to corrupting the stack or any other object). Thus, no hazard pointers are needed for the push routine.

In the pop routine, the node access in line 10 is hazardous and the CAS in line 11 is ABA-prone. All hazards use the reference t. The technique employed in transforming the pop routine is the same as that used in the enqueue routine of Fig. 6.

4.3 List-Based Sets and Hash Tables

Fig. 9 shows an improved version of the lock-free list-based set implementation in [16], using hazard pointers. The algorithm in [16] improves on that of Harris [5] by allowing efficient memory management. It can be used as a building block for implementing lock-free chaining hash tables.

Node insertion is straightforward. Deletion of a node involves first marking the low bit of its *Next* pointer and then removing it from the list, to prevent other threads from linking newly inserted nodes to removed nodes [23], [5]. Whenever a traversing thread encounters a node marked for deletion, it removes the node before proceeding, to avoid creating references to nodes after their removal.

The traversal of the list requires the protection of at most two nodes at any time: a current node **cur*, if any, and its predecessor, if any. Two hazard pointers, ***hp0** and ***hp1**, are used for protecting these two nodes, respectively. The traversing thread starts the main loop (lines 12-25) with **prev* already protected. In the first iteration, **prev* is the root and, hence, it is safe. The hazard pointer ***hp0** is set by

```
structure NodeType { Data:DataType; Next:*NodeType; }
// Shared variables
Top:*NodeType;  // Initially null
// hp is a private ptr to one of the thread's hazard ptrs.

Push(data:DataType) {
 1:  node ← NewNode();
 2:  node^.Data ← data;
     while true {
 3:    t ← Top;
 4:    node^.Next ← t;
 5:    if CAS(&Top,t,node) return;
     }
}

Pop() : DataType {
     while true {
 6:    t ← Top;
 7:    if (t = null) return EMPTY;
 8:    *hp ← t;
 9:    if (Top ≠ t) continue;
10:    next ← t^.Next;
11:    if CAS(&Top,t,next) break;
     }
12:  data ← t^.Data;
13:  RetireNode(t); return data;
}
```

Fig. 8. Lock-free stack using hazard pointers.

assigning it the hazardous reference *cur* (line 13) and then validating that the node **cur* is in the list at a point subsequent to the setting of the hazard pointer. The validation is done by validating that the pointer **prev* still has the value *cur*. This validation suffices as the semantics of the algorithm guarantee that the value of **prev* must change, if either the node that includes **prev* or its successor, the node **cur*, is removed. The protection of **prev* itself is guaranteed, either by being the root (in the first iteration), or by guaranteeing (as described below) that the node that contains it is covered by ***hp1**.

For protecting the pointer **prev* in subsequent iterations, as *prev* takes an arithmetic variation of the value of *cur* (line 23), the algorithm exchanges the private labels of ***hp0** and ***hp1** (line 24). There are no windows of vulnerability since the steps after line 21 to the end of the loop do not contain any hazards. Also, since ***hp0** already protects the reference *cur* at line 11a, then there is no need for further validation that the node formerly pointed to by *cur* and now contains **prev* is in the list.

Therefore, all the hazards (lines 4, 6, 7, 15, 17, 20, and 21) are covered by hazard pointers according to the condition in Section 3.3.

This algorithm demonstrates an interesting case where hazard pointers are used to protect nodes that are not adjacent to the roots of the object, unlike the cases of the queue and stack algorithms.

4.4 Single-Writer Multireader Dynamic Structures

Tang et al. [27] used lock-free single-writer multireader doubly-linked lists for implementing point-to-point send and receive queues, to improve the performance of threaded MPI

```
structure NodeType { Key:KeyType; Next:*NodeType;};
// Per-thread private variables
prev: **NodeType; cur,next: *NodeType;
// hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs.
// Integer arithmetic in lines 6, 17, and 19.

Insert(head:**NodeType,node:*NodeType):Boolean {
1:  key ← node^.Key;
    while true {
2:    if Find(head,key) return false;
3:    node^.Next ← cur;
4:    if CAS(prev,cur,node) return true;
    }
}

Delete(head:**NodeType,key:KeyType):Boolean {
    while true {
5:    if ¬Find(head,key) return false;
6:    if ¬CAS(&cur^.Next,next,next+1) continue;
7:    if CAS(prev,cur,next) RetireNode(cur); else Find(head,key);
8:    return true;
    }
}

Search(head:**NodeType,key:KeyType):Boolean {
9:  return Find(head,key);
}

Find(head:**NodeType;key:KeyType) : Boolean {
try_again:
10:   prev ← head;
11:   cur ← *prev;
12:   while (cur ≠ null) {
13:     *hp0 ← cur;
14:     if (*prev ≠ cur) goto try_again;
15:     next ← cur^.Next;
16:     if (next & 1) { // bitwise AND
17:       if ¬CAS(prev,cur,next-1) goto try_again;
18:       RetireNode(cur);
19:       cur ← next-1;
        } else {
20:       ckey ← cur^.Key;
21:       if (*prev ≠ cur) goto try_again;
22:       if (ckey≥key) return (ckey = key);
23:       prev ← &cur^.Next;
24:       tmp ← hp0;  hp0 ← hp1;  hp1 ← tmp; // all private
25:       cur ← next;
        }
      }
26:   return false;
}
```

Fig. 9. Lock-free list-based set with hazard pointers.

```
structure NodeType { Key:KeyType; Data:DataType;
        Prev:**NodeType; Next:*NodeType; };
// hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs.

SingleWriterInsertAfter(prev:**NodeType,node:*NodeType) {
 1:     next ← *prev;
 2:     node^.Prev ← prev;
 3:     node^.Next ← next;
 4:     if (next ≠ null) next^.Prev ← &node^.Next;
 5:     *prev ← node; // Inserted
}

SingleWriterDelete(node:*NodeType) {
 6:     prev ← node^.Prev;
 7:     next ← node^.Next;
 8:     if (next ≠ null) next^.Prev ← prev;
 9:     *prev ← next; // Deleted
10:     node^.Next ← null; // To alert readers not to proceed
11:     RetireNode(node);
}

ReaderSearch(head:**NodeType;key:KeyType) : DataType {
try_again:
12:     prev ← head;
13:     cur ← *prev;
14:     while (cur ≠ null) {
15:         *hp0 ← cur;
16:         if (*prev ≠ cur) goto try_again;
17:         next ← cur^.Next;
18:         ckey ← cur^.Key;
19:         if (cur^.Key = key) {
20:             data ← cur^.Data;
21:             if (*prev ≠ cur) goto try_again;
22:             return data;
            }
23:         if (*prev ≠ cur) goto try_again;
24:         prev ← &cur^.Next;
25:         tmp ← hp0;  hp0 ← hp1;  hp1 ← tmp;
26:         cur ← next;
        }
27:     return NOTFOUND;
}
```

Fig. 10. Lock-free single-writer multiple-reader doubly-linked list with hazard pointers.

on multiprogrammed shared memory systems. The main challenge for that implementation was memory management. That is, how does the owner (i.e., the single writer) guarantee that no readers still hold references to a removed node before reusing or freeing it? In order to avoid inefficient per-node reference counting, they use instead aggregate (per-list) reference counting. However, the delay or failure of a reader thread can prevent the owner indefinitely from reusing an unbounded number of removed nodes. As discussed in Section 6.1, this type of aggregate methods is not lock-free as it is sensitive to thread delays and failures. Furthermore, even without any thread delays, in that implementation, if readers keep accessing the list, the aggregate reference counter may never reach zero, and the owner remains indefinitely unable to reuse removed nodes. Also, the manipulation of the reference counters requires atomic operations such as CAS or atomic add.

We use hazard pointers to provide an efficient single-writer multireader doubly-linked list implementation that allows lock-free memory reclamation. Fig. 10 shows the code for the readers' search routine, and the owner's insertion and deletion routines. Unlike using locks or reference counting, reader threads do not write to any shared variables other than their own hazard pointers, which are rarely accessed by the writer thread, thus decreasing cache coherence traffic. Also, instead of a quadratic number of per-list locks or reference counters, only two hazard pointers are needed per reader thread for supporting an arbitrary number of lists.

The algorithm uses only single-word reads and writes for memory access. Thus, it demonstrates the importance of the feasibility of the hazard pointer methodology on systems with hardware support for memory access limited to these instructions.

The algorithm is mostly straightforward. However, it is worth noting that changing *node^.Next* in line 10 in the *SingleWriterDelete* routine is necessary if **node* is not the last node in the list. Otherwise, it is possible that the validation condition in line 23 of a concurrent execution of the *ReaderSearch* routine by a reader thread may succeed even after the owner has retired the node containing **prev*. If so, and if the owner also removes the following node, the reader might continue to set a hazard pointer for the following node, but too late after it has already been removed and reused.

5 Experimental Performance Results

This section presents experimental performance results comparing the performance of the new methodology to that of other lock-free memory management methods. We implemented lock-free algorithms for FIFO queues [21], LIFO stacks [11], and chaining hash tables [16], using hazard pointers, ABA-prevention tags [11], and lock-free reference counting [29]. Details of the latter memory management methods are discussed in Section 6.1.

Also, we included in the experiments efficient commonly-used lock-based implementations of these object types. For FIFO queues and LIFO stacks, we used the ubiquitous test-and-test-and-set [24] lock with bounded exponential backoff. For the hash tables, we used implementations with 100 separate locks protecting 100 disjoint groups of buckets, thus allowing complete concurrency between operations on different bucket groups. For these locks, we used Mellor-Crummey and Scott's [15] simple fair reader-writer lock to allow intrabucket group concurrency among read-only operations.

Experiments were performed on an IBM RS/6000 multiprocessor with four 375 MHz POWER3-II processors. We ran the experiments when no other users used the system. Data structures of all implementations were aligned to cache line boundaries and padded where appropriate to eliminate false sharing. In all experiments, all needed memory fit in physical memory. Fence instructions were inserted in the code of all implementations wherever memory ordering is required. Locks and single-word and double-word CAS were implemented using the single-word and double-word LL/SC instructions supported on the POWER3 architecture in 32-bit mode. All implementations were compiled at the highest optimization level.

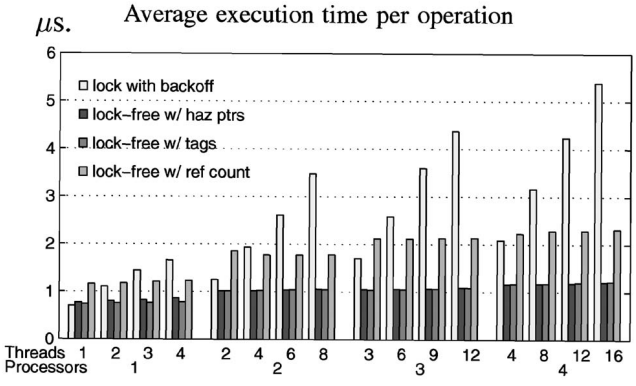

Fig. 11. Performance of FIFO queues.

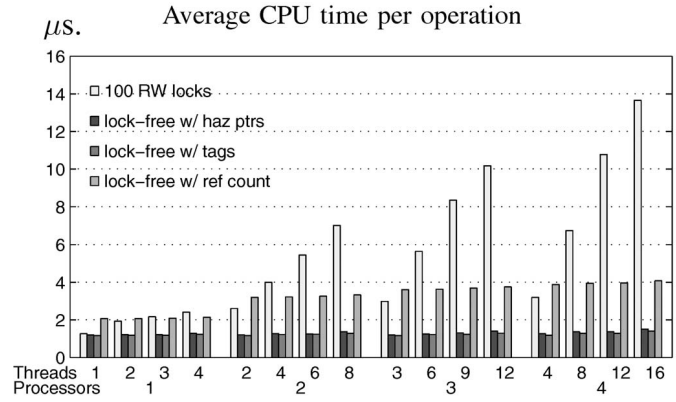

Fig. 13. Performance of hash tables with load factor 1.

We ran each experiment five times and reported the average of the median three. Variance was negligible in all experiments. Reported times exclude initialization. For each implementation, we varied the number of processors used from one to four, and the number of threads per processor from one to four. At initialization, each thread was bound to a specific processor. All needed nodes were allocated at initialization. During time measurement, removed nodes were made ready for reuse, but were not deallocated. The pseudorandom sequences for generating keys and operations for different threads were nonoverlapping in each experiment, but were repeatable in every experiment for fairness in comparing different implementations.

For the hazard pointer implementations, we set the number of threads N conservatively to 64, although smaller numbers of threads were used. For implementations with reference counting, we carefully took into account the semantics of the target algorithms to minimize the number of updates to reference counters.

Figs. 11 and 12 show the average execution time per operation on shared FIFO queue and LIFO stack implementations, respectively. In each experiment, each thread executed 1,000,000 operations, i.e., enqueue and dequeue operations for queues and push and pop operations for stacks. The abbreviated label **"haz ptrs"** refers to hazard pointers, **"tags"** refers to ABA-prevention tags, and **"ref count"** refers to lock-free reference counting.

Figs. 13 and 14 show the average CPU time (product of execution time and the number of processors used) per operation on shared chaining hash tables with 100 buckets and load factors of 1 and 5, respectively. The load factor of a hash table is the average number of items per bucket. In each experiment, each thread executed 2,000,000 operations (Insert, Delete, and Search).

A common observation in all the graphs is that lock-free implementations with hazard pointers perform as well as and often significantly better than the other implementations in virtually all cases. Being lock-free, the performance of lock-free objects is unaffected by preemption, while locks perform poorly under preemption. For example, the lock-free hash table with hazard pointers achieves throughput that is 251 percent, 496 percent, 792 percent, and 905 percent that of the lock-based implementation, with 4, 8, 12, and 16 threads, respectively, running on four processors (Fig. 13).

In all experiments, the performance of hazard pointers is comparable to that of ABA-prevention tags. However, unlike hazard pointers, tags require double-width instructions and hinder rather than assist memory reclamation.

Lock-free objects with hazard pointers handle contention (Figs. 11 and 12) significantly better than locks even in the absence of preemption. For example, under contention by four processors, and even without preemption, they achieve throughputs that are 178 percent that of locks when operating on queues. Note that experiments using locks without backoff (not shown) resulted in more than doubling the execution time for locks under contention by four processors. Using backoff in the lock-free implementations with hazard pointers resulted in moderate improvements in performance under contention by four processors (25 percent on queues and 44 percent on stacks). However, we conservatively report the results without backoff.

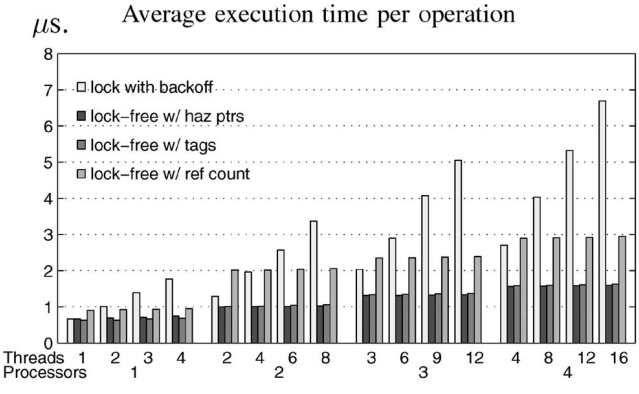

Fig. 12. Performance of LIFO stacks.

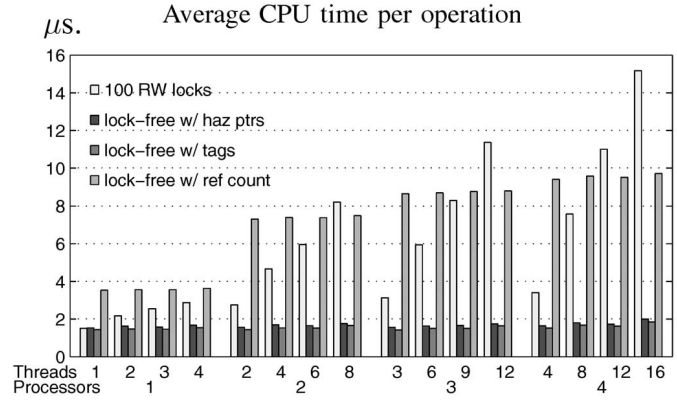

Fig. 14. Performance of hash tables with load factor 5.

Lock-free implementations with hazard pointers also outperform lock-based implementations as well as those with reference counting in the case of no or negligible contention, but under sharing, as is the case with hash tables (Figs. 13 and 14). This is because they do not write to any shared locations other than hazard pointers, during read-only Search operation as well as traversal, thus minimizing cache coherence traffic.[2] On the other hand, lock acquisition and release, even for read-only transactions, result in writes to lock variables. As for reference counting, the situation is even worse, the reference counter of each traversed node needs to be incremented and then decremented, even during read-only transactions. The effect is most evident in the case of hash tables with a load factor of 5 (Fig. 14). The throughput of the implementation using hazard pointers is 573 percent that of the one using reference counting, on four processors.

For hash tables, we ran experiments (not shown) using a single lock for the whole hash table. As expected, the performance of these implementations was extremely poor (more than 10 fold increase in execution time on four processors over the highly concurrent implementations with 100 disjoint locks). We also ran experiments using 100 test-and-test-and-set mutual exclusion locks instead of 100 reader-writer locks. The performance of the former (not shown) is slightly worse than the latter (shown), as they do not allow intrabucket concurrency of read-only operations. Hence, we believe that we have chosen very efficient lock-based implementations as the baseline for evaluating lock-free implementations and their use of hazard pointers.

Also, we conservatively opted to focus on low levels of contention and multiprogramming, that tend to limit the performance advantages of lock-free synchronization. A common criticism of unconventional synchronization techniques is that they achieve their advantages only under uncommonly high levels of contention, and that they often perform very poorly in comparison to simple mutual exclusion in the common case of low contention. Our experimental results show that lock-free objects, with hazard pointers, offer comparable performance to that of the simplest and most efficient lock-based implementations, under no contention and no preemption, in addition to their substantial performance advantages under higher levels of contention and preemption.

The superior performance of lock-free implementations using hazard pointers to that of lock-based implementations is attributed to several factors. Unlike locks,

1. they operate directly on the shared objects without the need for managing additional lock variables,
2. read-only operations do not result in any writes to shared variables (other than the mostly private hazard pointers),
3. there is no useless spinning, as each attempt has a chance of success when it starts, which makes them more tolerant to contention, and
4. progress is guaranteed under preemption.

Note that the effects of the first two factors are applicable even under no contention and no preemption.

2. Typically, a write by a processor to a location that is cached in its cache with a read-only permission results in invalidating all cached copies of the location in other processors' caches, and results in additional traffic if the other processors later need to access the same cache line. However, a read to a cached copy with read-only permission does not result in any coherence traffic.

It is worth noting that, while scalable queue-based locks [14], [15] outperform the simple locks we used in this study under high levels of contention, they underperform them in the common case of no or low contention. Furthermore, these locks are extremely sensitive to even low levels of preemption, as the preemption of any thread waiting in the lock queue—not just the lock holder—can result in blocking.

Overcoming the effects of preemption on locks may be partially achieved, but only by using more complicated and expensive techniques such as handshaking, timeout, and communication with the kernel [1], [12], [22], [30] that further degrade the performance of the common case. On the other hand, as shown in the graphs, lock-free implementations with hazard pointers are inherently immune to thread delays, at virtually no loss of performance in the case of no contention and no preemption, and outperform lock-based implementations under preemption and contention.

6 Discussion

6.1 Related Work

6.1.1 IBM ABA-Prevention Tags

The earliest and simplest lock-free method for node reuse is the tag (update counter) method introduced with the documentation of CAS on the IBM System 370 [11]. It requires associating a tag with each location that is the target of ABA-prone comparison operations. By incrementing the tag when the value of the associated location is written, comparison operations (e.g., CAS) can determine if the location was written since it was last accessed by the same thread, thus preventing the ABA problem. The method requires that the tag contains enough bits to make full wraparound impossible during the execution of any single lock-free attempt. This method is very efficient and allows the immediate reuse of retired nodes.

On the downside, when applied to arbitrary pointers as it is the case with dynamic sized objects, it requires double-width instructions to allow the atomic manipulation of the pointer along with its associated tag. These instructions are not supported on 64-bit architectures. Also, in most cases, the semantics of the tag field must be preserved indefinitely. Thus, if the tag is part of a dynamic node structure, these nodes can never be reclaimed. Their memory cannot be divided or coalesced, as this may lead to changing the semantics of the tag fields. That is, once a range of memory locations are used for a certain node type, they cannot be reused for other node types that do not preserve the semantics of the tag fields.

6.1.2 Lock-Free Reference Counting

Valois [29] presented a lock-free reference counting 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 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. Only after its reference counter goes to zero, can a node be reused. However, due to the use of single-address CAS to manipulate pointers and independently located reference counters nonatomically, the resulting timing windows dictate the permanent retention of nodes of their

type and reference counter field semantics, thus hindering memory reclamation.

Detlefs et al. [3] presented a lock-free reference counting method that uses the mostly unsupported DCAS primitive (i.e., CAS on two independent locations) to operate on both 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 reuse. However, free memory cannot be returned to the operating system.

The most important disadvantage of per-node reference counting is the prohibitive performance cost due to the—otherwise, unnecessary—updates of the reference counters of referenced nodes, even for read-only access, thus causing significant increase in cache coherence traffic.

6.1.3 Scheduler-Dependent and Blocking Methods

Methods in this category are either sensitive to thread failures and delays, i.e., the delay of a single thread can—and most likely will—prevent the reuse of unbounded memory indefinitely; or dependent on special kernel or scheduler support for recovery from such delays and failures.

McKenney and Slingwine [13] presented read-copy update, a framework where a retired node can be reclaimed only after ascertaining that each of the other threads has reached a *quiescence point* after the node was removed. The definition of quiescence points, varies depending on the programming environment. Typically, implementations of read-copy update use timestamps or collective reference counters. Not all environments are suitable for the concept of quiescence points, and without special scheduler support the method is blocking, as the delay of even one thread prevents the reuse of unbounded memory.

Greenwald [4] presented brief outlines of *type stable memory* implementations that depend on special support from the kernel for accessing the private variables of threads and detecting thread delays and failures. The core idea is that threads set timestamps whenever they reach *safe points* such as the kernel's top level loop, where processes are guaranteed not to be holding stale references. The earliest timestamp represents a *high water mark* where all nodes retired before that time can be reclaimed safely.

Harris [5] 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 retired nodes: an old list and a new list. Retired 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 retired nodes from being reused. This can happen even without thread delays. If a thread simply does not operate on the target object, then the thread's timestamp will remain unupdated.

One crucial difference between hazard pointers and these methods [13], [4], [5] is that the former does not use reference counters or timestamps. The use of aggregate reference counters for unbounded numbers of nodes and/or the reliance on per-thread timestamps makes a memory management method inherently vulnerable to the failure or delay of even one thread.

6.1.4 Recent Work

A preliminary version of this work [17] was published in January 2002. Independently, Herlihy et al. [10] developed a memory reclamation methodology. The basic idea of their methodology is the same as that of ours. The difference is in the *Liberate* routine that corresponds to our *Scan* routine. *Liberate* is more complex than *Scan* and uses double-word CAS. Our methodology retains important advantages over theirs, regarding performance as demonstrated experimentally and regarding independence of special hardware support. Our methodology—even with the algorithm extensions in Section 3.2—uses only single-word instructions, while theirs requires double-word CAS—which is not supported on 64-bit processor architecture—in its core operations. Also, our methodology uses only reads and writes in its core operations, while theirs uses CAS in its core operations. This prevents their methodology from supporting algorithms that require only reads and writes (e.g., Fig. 10)—which are otherwise feasible—on systems without hardware support for CAS or LL/SC.

6.2 Conclusions

The problem of memory reclamation for dynamic lock-free objects has long discouraged the wide use of lock-free objects, despite their inherent performance and reliability advantages over conventional lock-based synchronization.

In this paper, we presented the *hazard pointer methodology*, a practical and efficient solution for memory reclamation for dynamic lock-free objects. It allows unrestricted memory reclamation, it takes only constant expected amortized time per retired node, it does not require special hardware support, it uses only single-word instructions, it does not require special kernel support, it guarantees an upper bound on the number of retired nodes not yet ready for reuse at any time, it is wait-free, it offers a lock-free solution for the ABA problem, and it does not require any extra shared space per pointer, per node, or per object.

Our experimental results demonstrate the overall excellent performance of hazard pointers. They show that the hazard pointer methodology offers equal and more often significantly better performance than other memory management methods, in addition to its qualitative advantages regarding memory reclamation and independence of special hardware support.

Our results also show that lock-free implementations of important object types, using hazard pointers, offer comparable performance to that of efficient lock-based implementations under no contention and no multiprogramming, and outperform them by significant margins under moderate multiprogramming and/or contention, in addition to guaranteeing continuous progress and availability even in the presence of thread failures and arbitrary delays.

A growing number of efficient algorithms for lock-free dynamic objects are available. The hazard pointer methodology enhances their practicality by enabling memory reclamation, and at the same time allowing them to achieve excellent robust performance. In combination with a recent completely lock-free algorithm for dynamic memory allocation [20], the hazard pointer methodology enables these objects at last to be completely dynamic and at the same time completely lock-free, regardless of support for automatic garbage collection.

ACKNOWLEDGMENTS

The author thanks the editor and the reviewers for valuable comments on this paper, and Marc Auslander, Maurice Herlihy, Victor Luchangco, Paul McKenney, Mark Moir, Bryan Rosenburg, Michael Scott, Ori Shalev, Nir Shavit, Yefim Shuf, and Robert Wisniewski for useful discussions and comments on various stages of this work, and Victor Luchangco for suggesting pointer renaming as an alternative to pointer ordering.

REFERENCES

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *ACM Trans. Computer Systems,* vol. 10, no. 1, pp. 53-79, Feb. 1992.

[2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* MIT Press, 1990.

[3] D.L. Detlefs, P.A. Martin, M. Moir, and G.L. Steele Jr., "Lock-Free Reference Counting," *Proc. 20th Ann. ACM Symp. Principles of Distributed Computing,* pp. 190-199, Aug. 2001.

[4] M.B. Greenwald, "Non-Blocking Synchronization and System Design," PhD thesis, Stanford Univ., Aug. 1999.

[5] T.L. Harris, "A Pragmatic Implementation of Non-Blocking Linked Lists," *Proc. 15th Int'l Symp. Distributed Computing,* pp.*nbsp;300-314, Oct. 2001.

[6] T.L. Harris, K. Fraser, and I.A. Pratt, "A Practical Multi-Word Compare-and-Swap Operation," *Proc. 16th Int'l Symp. Distributed Computing,* pp. 265-279, Oct. 2002.

[7] D. Hendler and N. Shavit, "Work Dealing," *Proc. 14th Ann. ACM Symp. Parallel Algorithms and Architectures,* pp. 164-172, Aug. 2002.

[8] M.P. Herlihy, "Wait-Free Synchronization," *ACM Trans. Programming Languages and Systems,* vol. 13, no. 1, pp. 124-149, Jan. 1991.

[9] M.P. Herlihy, "A Methodology for Implementing Highly Concurrent Objects," *ACM Trans. Programming Languages and Systems,* vol. 15, no. 5, pp. 745-770, Nov. 1993.

[10] M.P. Herlihy, V. Luchangco, and M. Moir, "The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized Lock-Free Data Structures," *Proc. 16th Int'l Symp. Distributed Computing,* pp. 339-353, Oct. 2002.

[11] IBM, IBM System/370 Extended Architecture, Principles of Operation, publication no. SA22-7085, 1983.

[12] L.I. Kontothanassis, R.W. Wisniewski, and M.L. Scott, "Scheduler-Conscious Synchronization," *ACM Trans. Computer Systems,* vol. 15, no. 1, pp. 3-40, Feb. 1997.

[13] P.E. McKenney and J.D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," *Proc. 10th IASTED Int'l Conf. Parallel and Distributed Computing and Systems,* Oct. 1998.

[14] J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Computer Systems,* vol. 9, no. 1, pp. 21-65, Feb. 1991.

[15] J.M. Mellor-Crummey and M.L. Scott, "Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors," *Proc. Third ACM Symp. Principles and Practice of Parallel Programming,* pp. 106-113, Apr. 1991.

[16] M.M. Michael, "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets," *Proc. 14th Ann. ACM Symp. Parallel Algorithms and Architectures,* pp. 73-82, Aug. 2002.

[17] M.M. Michael, "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes," *Proc. 21st Ann. ACM Symp. Principles of Distributed Computing,* pp. 21-30, July 2002. earlier version in Research Report RC 22317, IBM T.J. Watson Research Center, Jan. 2002.

[18] M.M. Michael, "CAS-Based Lock-Free Algorithm for Shared Deques," *Proc. Ninth Euro-Par Conf. Parallel Processing,* pp. 651-660, Aug. 2003.

[19] M.M. Michael, "ABA Prevention Using Single-Word Instructions," Technical Report RC 23089, IBM T.J. Watson Research Center, Jan. 2004.

[20] M.M. Michael, "Scalable Lock-Free Dynamic Memory Allocation," *Proc. 2004 ACM SIGPLAN Conf. Programming Language Design and Implementation,* June 2004.

[21] M.M. Michael and M.L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," *Proc. 15th Ann. ACM Symp. Principles of Distributed Computing,* pp. 267-275, May 1996.

[22] M.M. Michael and M.L. Scott, "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors," *J. Parallel and Distributed Computing,* vol. 51, no. 1, pp. 1-26, May 1998.

[23] S. Prakash, Y.-H. Lee, and T. Johnson, "A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap," *IEEE Trans. Computers,* vol. 43, no. 5, pp. 548-559, May 1994.

[24] L. Rudolph and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proc. 11th Int'l Symp. Computer Architecture,* pp. 340-347, June 1984.

[25] O. Shalev and N. Shavit, "Split-Ordered Lists: Lock-Free Extensible Hash Tables," *Proc. 22nd Ann. ACM Symp. Principles of Distributed Computing,* pp. 102-111, July 2003.

[26] N. Shavit and D. Touitou, "Software Transactional Memory," *Distributed Computing,* vol. 10, no. 2, pp. 99-116, 1997.

[27] H. Tang, K. Shen, and T. Yang, "Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines," *ACM Trans. Programming Languages and Systems,* vol. 22, no. 4, pp. 673-700, July 2000.

[28] J. Turek, D. Shasha, and S. Prakash, "Locking Without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking," *Proc. 11th ACM Symp. Principles of Database Systems,* pp. 212-222, June 1992.

[29] J.D. Valois, "Lock-Free Linked Lists Using Compare-and-Swap," *Proc. 14th Ann. ACM Symp. Principles of Distributed Computing,* pp. 214-222, Aug. 1995.

[30] J. Zahorjan, E.D. Lazowska, and D.L. Eager, "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 2, no. 2, pp. 180-198, Apr. 1991.

Maged M. Michael received the PhD degree in computer science from the University of Rochester in 1997. Since then, he has been a research staff member at the IBM Thomas J. Watson Research Center at Yorktown Heights, New York. Dr. Michael's research interests include concurrent programming, memory management, distributed and fault-tolerant computing, parallel computer architecture, cache coherence protocol design, and multiprocessor simulation methodology.