High Performance Dynamic Lock-Free Hash Tables and List-Based Sets

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

ABSTRACT

Lock-free (non-blocking) shared data structures promise more robust performance and reliability than conventional lock-based implementations. However, all prior lock-free algorithms for sets and hash tables suffer from serious drawbacks that prevent or limit their use in practice. These drawbacks include size inflexibility, dependence on atomic primitives not supported on any current processor architecture, and dependence on highly-inefficient or blocking memory management techniques.

Building on the results of prior researchers, this paper presents the first CAS-based lock-free list-based set algorithm that is compatible with all lock-free memory management methods. We use it as a building block of an algorithm for lock-free hash tables. In addition to being lock-free, the new algorithm is dynamic, linearizable, and space-efficient.

Our experimental results show that the new algorithm outperforms the best known lock-free as well as lock-based hash table implementations by significant margins, and indicate that it is the algorithm of choice for implementing shared hash tables.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming/multitasking, synchronization*; E.2 [**Data Storage Representations**]: *hash table representations*

General Terms: Algorithms, Performance, Reliability

1. INTRODUCTION

The hash table is a ubiquitous data structure widely used in system programs and applications as a search structure. Its appeal lies in its guarantee of completing each operation in expected constant time, with the proper choice of a hashing function and assuming a constant load factor [3, 12].

To ensure correctness, concurrent access to shared objects must be synchronized. The most common synchronization method is the use of mutual exclusion locks. However, lock-based shared objects suffer significant performance degradation when faced with the inopportune delay of a thread while holding a lock, for instance due to preemption. While the lock holder is delayed, other active threads that need access to the locked shared object are prevented from making progress until the lock is released by the delayed thread.

A lock-free (also called non-blocking) implementation of a shared object guarantees that if there is an active thread trying to perform an operation on the object, some operation, by the same or another thread, will complete within a finite number of steps regardless of other threads' actions [8]. Lock-free objects are inherently immune to priority inversion and deadlock, and offer robust performance, even with indefinite thread delays and failures.

Shared sets (also called dictionaries) are the building blocks of hash table buckets. Several algorithms for lock-free set implementations have been proposed. However, all suffer from serious drawbacks that prevent or limit their use in practice.

Lock-free set algorithms fall into two main categories: array-based and list-based. Known array-based lock-free set algorithms [5, 13] are generally impractical. In addition to restricting maximum set size inherently, they do not provide mechanisms for preventing duplicate keys from occupying multiple array elements, thus limiting the maximum set size even more, and requiring excessive overallocation in order to guarantee lower bounds on maximum set sizes.

Prior list-based lock-free set algorithms involve one or more serious problems: dependence on the DCAS (double-compare-and-swap)[1] atomic primitive that is not supported on any current processor architecture [5, 14], susceptibility to livelock [25], and/or dependence on problematic memory management methods [6, 14, 25] (i.e., memory management methods that are impractical, very inefficient, blocking (not lock-free), and/or dependent on special operating system support).

The use of universal lock-free methodologies [1, 2, 8, 11, 22, 24] for implementing hash tables or sets in general is too inefficient to be practical.

This paper presents a lock-free list-based set algorithm that we use as a building block of a lock-free hash table algorithm. The algorithm is dynamic, allowing the object size and memory use to grow and shrink arbitrarily. It satisfies the linearizability correctness condition [9].

It uses CAS (compare-and-swap) or equivalently restricted LL/SC (load-linked/store-conditional). CAS takes three arguments: the address of a memory location, an expected

[1]DCAS 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.

SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.

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 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, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. All architectures that support LL/SC restrict some or all memory accesses between LL and SC, and allow SC to fail spuriously. All current major processor architectures support one of these two primitives.

This algorithm is the first CAS-based list-based set or hash table algorithm that is compatible with simple and efficient methods [10, 17], as well as all other memory management methods for lock-free objects.

Our experimental results show significant performance advantages of the new algorithm over the best known lock-free as well as lock-based hash table implementations. The new algorithm outperforms the best known lock-free algorithm [6] by a factor of 2.5 or more, in all lock-free cases. It outperforms the best lock-based implementations, under high and low contention, with and without multiprogramming, often by significant margins.

In Section 2 we review prior lock-free algorithms for sets and hash tables and relevant memory management methods. In Section 3 we present the new algorithm. In Section 4 we discuss its correctness. In Section 5 we present performance results relative to other hash table implementations. We conclude with Section 6.

2. BACKGROUND

2.1 The Hash Table Data Structure

A hash table is a space efficient representation of a set object K when the size of the universe of keys U that can belong to K is much larger than the average size of K. The most common method of resolving collisions between multiple distinct keys in K that hash to the same hash value h is to chain nodes containing the keys (and optional data) into a linked list (also called bucket) pointed to by a head pointer in the array element of the hash table array with index h. The load factor α is the ratio of $|K|$ to m, the number of hash buckets [3, 12].

With a well-chosen hash function $h(k)$ and a constant average α, operations on a hash table are guaranteed to complete in constant time on the average. This bound holds for shared hash tables in the absence of contention.

The basic operations on hash tables are: *Insert*, *Delete* and *Search*. Most commonly, they take a key value as an argument and return a Boolean value. *Insert(k)* checks if nodes with key k are in the bucket headed by the hash table array element of index $h(k)$. If found (i.e., $k \in K$), it returns false. Otherwise it inserts a new node with key k in that bucket and returns true.

Delete(k) also checks the bucket with index $h(k)$ for nodes with key k. If found, it removes the nodes from the list and returns true. Otherwise, it returns false. *Search(k)* returns true if the bucket with index $h(k)$ contains a node with key k, and returns false otherwise.

For time and space efficiency most implementations do not allow multiple nodes with the same key to be present concurrently in the hash table. The simplest way to achieve this is to keep the nodes in each bucket ordered by their key values.

Figure 1 shows a list-based hash table representing a set K of positive integer keys. It has seven buckets and the hash function $h(k) = k \bmod 7$.

By definition, a hash *function* maps each key to one and only one hash value. Therefore, operations on different hash buckets are inherently disjoint and are obvious candidates for concurrency. Generally, hash table implementations allow concurrent access to different buckets or groups of buckets to proceed without interference. For example if locks are used, different buckets or groups of buckets can be protected by different locks, and operations on different bucket groups can proceed concurrently. Thus, shared set implementations are obvious building blocks of concurrent hash tables.

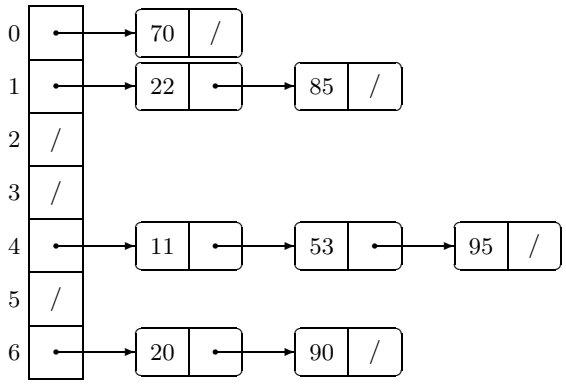


Figure 1: An example of a hash table with 7 buckets and hash function $h(k) = k \bmod 7$.

2.2 Prior Lock-Free Set Algorithms

Array-Based Lock-Free Set Algorithms

Lanin and Shasha [13] presented two array-based set algorithms that can be implemented using CAS. The first algorithm allows Delete and Search operations without locking, but requires Insert operations to use locks, in order to avoid the concurrent insertion of multiple instances of the same key into the hash table. The use of locks precludes the algorithm from being lock-free.

The second algorithm allows duplicate key values to occupy multiple array elements. Delete operations must search for and delete all matching keys in the array. Using arrays to represent sets requires the maximum bucket size to be static. For a hash table the maximum bucket size is too large to allow static allocation, and doing so would eliminate any advantage of hash tables over direct addressing of arrays. Furthermore, even if an upper bound on the bucket set size is known, the array cannot guarantee accommodating that size, without excessive overallocation, as the same key value may occupy multiple array entries. The algorithm is impractical even when a maximum set size can be set.

Greenwald [5] presented an incomplete lock-free array-based set algorithm using DCAS that does not address the issue of handling duplicate copies of the same key in the array.

List-Based Lock-Free Set Algorithms

Massalin and Pu [14] presented a sketch of a lock-free linked list algorithm that uses the Motorola 68040 CAS2 instruction (more commonly known as DCAS in recent years). The algorithm performs deletions from the middle of the list in two steps, it first marks a node as deleted and then removes it from the list. The algorithms deals with memory management of deleted nodes by using reference counting. Many details of the algorithm are lacking.

Also using DCAS, Greenwald [5] presented a lock-free linked list algorithm. The algorithm allows insert and delete operations to complete in one step, in the absence of contention, by using a version number that must be incremented with every change to the list.

DCAS was supported on some generations of the Motorola 68000 processor family (as CAS2) in the 1980s. These implementations were extremely inefficient. Since then no processor architecture supports DCAS. Algorithms using DCAS remain impractical until such an instruction is supported efficiently on processor architectures.

Valois [25] presented CAS-based lock-free linked list algorithms that place one or more auxiliary nodes between every two normal nodes (i.e., nodes containing keys) to allow safe deletions. The algorithms also rely on a shared cursor that has to be positioned using CAS, at the target nodes of operations on the list, before these operations can be attempted. However, the concept of a shared cursor is actually inherently inconsistent with lock-freedom. For example, it is possible that two threads attempting operations (e.g., Insert) on the list at different locations, indefinitely alternate moving the shared cursor to their respective desired positions, with neither of them completing its intended operation. That is, the algorithm is susceptible to livelock, thus violating a basic correctness condition, and by definition is not lock-free. In addition, the algorithm is inefficient due to the extra work manipulating auxiliary nodes, and is not compatible with simple and efficient lock-free memory management methods.

Harris [6] presented the first correct CAS-based lock-free list-based set algorithm. A Delete operation, first marks a node as deleted using CAS to prevent new nodes from being linked to it, and then removes it from the list by swinging the next pointer of the previous node to the next node in the list, also using CAS.

The algorithm allows a thread traversing the list to access the contents of a node or a sequence of nodes after they have been removed from the list. If removed nodes are allowed to be reused immediately, the traversing thread may reach an incorrect result or corrupt the list.

This precludes the algorithm from using simple and efficient lock-free memory management methods, the IBM freelists [10, 23] and the safe memory reclamation method [17]. It forces the algorithm to use problematic memory management methods as discussed in the following subsection.

2.3 Memory Management

Initially, Harris used Valois' [25] reference counting memory management method. The method requires the inclusion of a reference counter in each dynamic node, that reflects the maximum number of references to that node in the object and the registers and local variables of threads operating on the object. A node can be reused only after its reference counter goes to zero. As reported by Harris [6], the method entailed prohibitive degradation in execution time by a factor of 10 to 15 times, relative to experiments without memory management, where removed nodes are not reused. Obviously, prohibiting memory reuse is not a generally practical solution for this problem, since the address space no matter how large is a finite resource.

As an alternative, Harris suggested assuming the use of a tracing garbage collector. However, garbage collectors pose many problems for lock-free algorithms. First, the presence of a garbage collector is not universal. Thus, lock-free object libraries that assume the use of garbage collectors are not portable to systems without such support. Second, even if present, garbage collectors are not lock-free as they either require mutual exclusion or stop-the-world techniques, or require special operating system support to access private stack space and registers. Third, the failure or delay of the garbage collector may prevent threads operating on lock-free objects from making progress indefinitely, thus violating lock-freedom. Fourth, Harris' algorithm prohibits threads from nullifying the pointers of dynamic nodes after their removal, thus the indefinite delay of a single thread is certain to prevent the automatic garbage collector from freeing an unbounded number of nodes, indefinitely. The latter problem applies to Harris' algorithm when using lock-free reference counting methods [25, 4] as well.

As a third memory management option, Harris proposed a sketch of a deferred freeing memory management method. Each node includes an extra field through which it can be linked into a to-be-freed list when it is removed from the set object, without changing its critical contents. Each thread must set a per-thread shared timestamp before it starts an operation on the list. The method uses two to-be-freed lists that alternate taking the roles of the old list and the new list. The nodes in the old to-be-freed list are freed only when the removal time of the latest node in the list precedes the minimum per-thread timestamp. Also, at that time the two to-be-freed lists exchange their labels as old and new lists.

The method has multiple flaws. First, it is prone to deadlock if one of the threads does not perform operations on the set indefinitely (even if the thread itself is active). As the thread's timestamp remains unchanged indefinitely, the nodes in the old to-be-freed list are not freed, indefinitely, and the new to-be-freed list never takes the role of the old to-be-freed list. Second, even if threads are somehow guaranteed to operate on the set infinitely often, the method is actually blocking (i.e., not lock-free), as the delay or failure of a thread prevents it from updating its timestamp, and hence prevents the reuse of an unbounded number of nodes, indefinitely.

Detlefs *et. al.* [4] presented a lock-free reference counting memory management method that uses DCAS. Its performance is expected to be at best similar to that of Valois' reference counting method (i.e., extremely inefficient). Most statements in an algorithm involving pointers to dynamic nodes (even reads and register-to-register instruction) are transformed to functions involving CAS and DCAS operations. Its advantage over Valois' method is allowing arbitrary reuse of the memory of removed nodes.

Other known memory management methods that may accommodate Harris' list algorithm are not without serious problems. They are either blocking or depend on special operating system support [15, 5].

The simplest and most efficient lock-free memory management methods are not compatible with Harris' list algorithm. The IBM freelist [10, 23] is implemented as a lock-free stack. A thread allocates a node by popping it from the freelist in one successful CAS operation, and frees a node for future reuse by pushing it into the freelist, also in one successful CAS operation.

The other efficient lock-free memory management method is the safe memory reclamation method [17] that allows arbitrary reuse of the memory of deleted nodes and provides a solution to the ABA-problem[2] for pointers to dynamic nodes, without the use of extra space per pointer or per node. It guarantees an upper bound on the number of deleted nodes not yet freed, regardless of thread failures and delays. It is wait-free[3], and operating system-independent.

The new algorithm is the first CAS-based lock-free list-based set algorithm that is compatible with all lock-free memory management methods, including the latter two.

[2]The ABA problem [10] is historically associated with CAS. It happens 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 and corrupt the object, 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.

[3]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 [7].

```
// types and structures
structure NodeType {
   Key : KeyType;
   ⟨Mark,Next,Tag⟩ : ⟨boolean,*NodeType,TagType⟩;
}
structure MarkPtrType {
   ⟨Mark,Next,Tag⟩ : ⟨boolean,*NodeType,TagType⟩;
}
// T is the hash array
// M is the number of hash buckets
T[M] : MarkPtrType; // Initially ⟨0,null,dontcare⟩
```

Figure 2: Types and structures.

```
// Hash function
h(key:KeyType):0..M-1 { ... }

// Hash table operations
HashInsert(key:KeyType):boolean {
   // Assuming new node allocations always succeed
   node ← AllocateNode();
   node^.Key ← key;
   if Insert(&T[h(key)],node) return true;
   FreeNode(node); return false;
}

HashDelete(key:KeyType):boolean {
   return Delete(&T[h(key)],key);
}

HashSearch(key:KeyType):boolean {
   return Search(&T[h(key)],key);
}
```

Figure 3: Hash table operations.

3. THE ALGORITHM

Structures and Hash Table Functions

Since compatibility with simple and efficient memory management methods is a central advantage of the new algorithm, we start by presenting a version that is compatible with freelists [10, 23]. We discuss implementations of the new algorithm using other memory management methods later in this section.

The simplest and earliest known ABA-prevention mechanism 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 [10]. CAS (or a validation condition) succeeds only if the tag has not changed since the thread last read the location, assuming that the tag has enough bits to make full wraparound between the read and the CAS or validation condition practically impossible.

Figure 2 shows the data structures and the initial values of shared variables used by the algorithm. The main structure is an array T of size M. Each element in T is basically a pointer to a hash bucket, implemented as a singly linked list. For simplicity, we include a deletion mark with the header pointer, although it is guaranteed to be always clear.

Each dynamic node must contain the following fields: *Key*, *Mark*, *Next*, and *Tag*. The *Key* field holds a key value. The *Mark* field indicates if the key in the node has been deleted from the set. The *Next* field points to the following node in the linked list if any, or has a null value otherwise. The *Tag* field is used for preventing the ABA problem. ⟨*Mark,Next,Tag*⟩ must occupy a contiguous aligned memory block that can be manipulated atomically using CAS or LL/SC.

The *Mark* bit and the *Next* pointer can be placed in one word, since pointers are at least word aligned on all current major systems. The *Mark* bit can occupy a low order bit. The ABA-prevention *Tag* field can be placed in an adjacent word such that both words are aligned on a double word boundary.[4]. Later, in this section, we present an implementation that uses only single-word CAS or restricted LL/SC.

Figure 3 shows the hash table functions that use the new list-based set algorithm. Basically, every hash table operation, maps the input key to a hash bucket and then calls the corresponding list-based set function with the address of the bucket header as an argument.

The List-Based Set Algorithm

Figure 4 shows the Insert, Delete and Search operations of the new list-based set algorithm. The function Find (described later in detail) returns a Boolean value indicating whether a node with a matching key was found in the list. In either case, by its completion, it guarantees that the private variables *prev*, ⟨*cur,ptag*⟩ and ⟨*next,ctag*⟩ have captured a snapshot of a segment of the list including the node (if any) that contains the lowest key value greater than or equal to the input key and its predecessor pointer. Find guarantees that there was a time during its execution when $*prev$ was part of the list, $*prev = \langle 0,cur,ptag\rangle$, and if $cur \neq null$, then also at that time $cur\hat{}.\langle Mark,Next,Tag\rangle = \langle 0,next,ctag\rangle$ and $cur\hat{}.Key$ was the lowest key value that is greater than or equal to the input key. If $cur = null$ then it must be that at that time all the keys in the list were smaller than the input key. Note that, we assume a sequentially consistent memory model. Otherwise, memory barrier instructions need to be inserted in the code between memory accesses whose relative order of execution is critical.

An Insert operation returns false if the key is found to be already in the list. Otherwise, it attempts to insert the new node, containing the new key, before the node $cur\hat{}$, in one atomic step using CAS in line A3 after setting the *Next* pointer of the new node to *cur*, as shown in Figure 5. The success of the CAS in line A3 is the linearization point of an Insert of a new key in the set. The linearization point of an Insert that returns false (i.e., finds the key in the set) is discussed later when presenting Find.

The failure of the CAS in line A3 implies that one or more of three events must have taken place since the snapshot in Find was taken. Either the node containing $*prev$ was deleted (i.e. its *Mark* is set), the node $cur\hat{}$ was deleted and removed (i.e., no longer reachable from *head*), or a new node was inserted immediately before $cur\hat{}$.

A Delete operation returns false if the key is not found in the list, otherwise, $cur\hat{}.Key$ must have been equal to the input key. If the key is found, the thread executing Delete attempts to mark $cur\hat{}$ as deleted, using the CAS in line B2, as shown in Figure 6. If successful, the thread attempts to remove $cur\hat{}$ by swinging $prev\hat{}.Next$ to *next*, while verifying that $prev\hat{}.Mark$ is clear, using the CAS in line B3.

The key technique of marking the next pointer of a deleted node in order to prevent a concurrent insert operation from linking another node after the deleted node was used earlier in Harris' lock-free list-based set algorithm [6], and was first used in Prakash, Lee, and Johnson's [20] lock-free FIFO queue algorithm.

DeleteNode prepares the removed node for reuse and its implementation is dependent on the memory management

[4]Most current architectures (32-bit as well as 64-bit) that support CAS (Intel x86, Sun SPARC) or restricted LL/SC (PowerPC, MIPS, Alpha) support their operation on aligned 64-bit blocks, and aligned 128-bit operations are likely to follow on 64-bit architectures

```
// private variables
prev : *MarkPtrType;
⟨pmark,cur,ptag⟩ : MarkPtrType;
⟨cmark,next,ctag⟩ : MarkPtrType;

Insert(head:*MarkPtrType,node:*NodeType):boolean {
    key ← node^.Key;
    while true {
A1:   if Find(head,key) return false;
A2:   node^.⟨Mark,Next⟩ ← ⟨0,cur⟩;
A3:   if CAS(prev,⟨0,cur,ptag⟩,⟨0,node,ptag+1⟩)
          return true;
    }
}

Delete(head:*MarkPtrType,key:KeyType):boolean {
    while true {
B1:   if !Find(head,key) return false;
B2:   if !CAS(&cur^.⟨Mark,Next,Tag⟩,
                          ⟨0,next,ctag⟩,
                          ⟨1,next,ctag+1⟩) continue;
B3:   if CAS(prev,⟨0,cur,ptag⟩,⟨0,next,ptag+1⟩)
          DeleteNode(cur); else Find(head,key);
      return true;
    }
}

Search(head:*MarkPtrType,key:KeyType):boolean {
    return Find(head,key);
}

Find(head:*MarkPtrType;key:KeyType) : boolean {
try_again:
    prev ← head;
D1: ⟨pmark,cur,ptag⟩ ← *prev;
    while true {
D2:   if cur = null return false;
D3:   ⟨cmark,next,ctag⟩ ← cur^.⟨Mark,Next,Tag⟩;
D4:   ckey ← cur^.Key;
D5:   if *prev ≠ ⟨0,cur,ptag⟩ goto try_again;
      if !cmark {
D6:       if ckey≥key return ckey = key;
D7:       prev ← &cur^.⟨Mark,Next,Tag⟩;
      } else {
D8:       if CAS(prev,⟨0,cur,ptag⟩,⟨0,next,ptag+1⟩)
              {DeleteNode(cur); ctag ← ptag+1;}
          else
              goto try_again;
      }
D9:   ⟨pmark,cur,ptag⟩ ← ⟨cmark,next,ctag⟩;
    }
}
```

Figure 4: The list-based set algorithm.

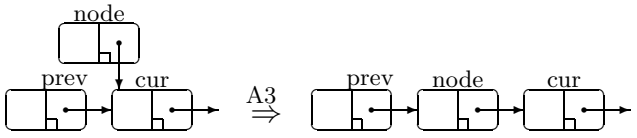


Figure 5: Insertion in the middle of the list.

method. For freelists, DeleteNode pushes the removed node onto the freelist.

The success of the CAS in line B2 is the linearization point of a Delete of a key that was already in the set. The linearization point of a Delete that does not find the input key in the set is discussed later when presenting the Find function.

The failure of the CAS in line B2 implies that one or more of three events must have taken place since the snapshot in Find was taken. Either the node *cur^* was deleted, a new

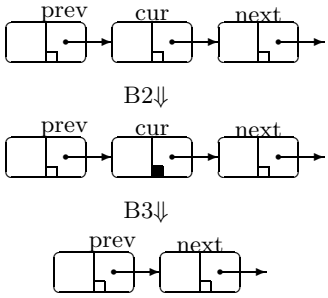


Figure 6: Deletion from the middle of the list.

node was inserted after *cur^*, or the node *next^* was removed from the list. The failure of the CAS in line B3 implies that another thread must have removed the node *cur^* from the list after the success of the CAS in line B2 by the current thread. In such a case, a new Find is invoked in order to guarantee that the number of deleted nodes not yet removed never exceeds the maximum number of concurrent threads operating on the object.

The Search operation simply relays the response of the Find function.

The Find function starts by reading the header of the list **head* in line D1. If the *Next* pointer of the header is null, then the list must be empty, therefore Find returns false after setting *prev* to *head* and *cur* to *null*. The linearization point of finding the list empty is the reading of **head* in line D1. That is, it is the linearization point of all Delete and Search operations that return false after finding the set empty.

If the list is not empty, a thread executing Find traverses the nodes of the list using the private pointers *prev*, *cur*, and *next*. Whenever it detects a change in **prev*, in lines D5 or D8, it starts over from the beginning. As discussed in Section 4, the algorithm is lock-free. A change in **prev* implies that some other threads have made progress in the meantime.

A thread keeps traversing the list until it either finds a node with a key greater than or equal to the input key, or reaches the end of the list without finding such node. If it is the former case, it returns the result of the condition *cur^.Key = key* at the time of its last execution of the read in line D3, with *prev* pointing to *cur^.⟨Mark,Next,Tag⟩* and *cur^.Key* is the lowest key in the set that is greater than or equal the input key, at that point (line D3). If the thread reaches the end of the list without finding a greater or equal key, it returns false, with **prev* pointing to the fields of the last node and *cur = null*.

In all cases of non-empty lists, the linearization point of the snapshot in Find is the last reading of cur^.⟨Mark,Next,Tag⟩ (line D3) by the current thread. That is, it is the linearization point of all Insert operations that return false and all Search operations that return true, as well as all Delete and Search operations that return false after finding the set non-empty.

During the traversal of the list, whenever the thread encounters a marked node, it attempts to remove it from the list, using CAS in line D8. If successful, the removed node is prepared for future reuse in DeleteNode and *cmark* is updated for continued traversal.

Note that, for a snapshot in Find to be valid, *prev^.Mark* and *cur^.mark* must be fount to be clear. If a *Mark* is found to be set the associated node must be removed first before capturing a valid snapshot.

On architectures that support restricted LL/SC but not CAS, implementing *CAS(addr,exp,new)* using the following routine suffices for the purposes of the new algorithm.

while true {*if LL(addr)* $\neq$ *exp return false;*
if SC(addr,new) return true;}

Moir [19] presented a general implementation of CAS using restricted LL/SC that allows *exp* = *new*, infinitely often.

Using Other Memory Management Methods

As mentioned earlier, the algorithm is compatible with all memory management methods for lock-free objects. For example, if lock-free reference counting [4, 25] or automatic garbage collection is used, when a node is removed, the call to DeleteNode only needs to nullify the removed node's fields, just in case their values match the address of some dynamic structure and thus may form a problematic garbage cycle. Since, as discussed in Section 2, all memory management methods other than freelists and the safe memory reclamation method (SMR) [17] are either extremely inefficient, blocking, dependent on special system support, and/or dependent on DCAS, we focus on using SMR with the new algorithm.

SMR's advantages over freelists include allowing the memory of removed dynamic nodes to be reused arbitrarily. That is, it allows the memory use of a dynamic data structure to shrink. Another advantage is that it can prevent the ABA problem without the need for double-width CAS or LL/SC or any extra space per pointer or per node. Thus, it allows lock-free objects to use minimal space, which is an important issue for hash tables with large numbers of buckets.

SMR requires target lock-free algorithms to associate a number of shared pointers, called hazard pointers, (three in the case of this algorithm) with each participating thread. The method guarantees that no deleted[5] dynamic node is freed or reused as long as some thread's hazard pointers have been pointing to it continuously from a time when it was not deleted. It is impossible for Valois' and Harris' list-based set algorithms to comply with this requirement as they allow a thread to traverse a node or a sequence of nodes after these nodes have already been removed from the list, and hence possibly deleted.

Figure 7 shows a version of the new algorithm that is compatible with SMR. Before returning, Insert, Delete, and Search nullify the hazard pointers to guarantee that the amortized time of processing a deleted node until it is freed for reuse is (logarithmically) bounded by contention. That is, whenever a thread is not operating on the object, its hazard pointers are null. This is done after the end of hazards (i.e., accesses to dynamic structure when they are possibly deleted and the use of pointers to dynamic structures as expected values of ABA-prone CAS operations in lines A3, B2 and B3).

In the Find function, there are accesses to dynamic structures in lines D3, D4, and D8, and the addresses of dynamic nodes are used as expected values of ABA-prone validation conditions and CAS operations in lines D5 and D8.

Lines E1 and E2 serve to guarantee that the next time a thread accesses *cur*ˆ in lines D3 and D4 and executes the validation condition in line D5, it must be the case that the hazard pointer *hp1 has been continuously pointing to *cur*ˆ from a time when it was in the list, thus guaranteeing that *cur*ˆ is not free during the execution of lines D3 and D4.

The ABA problem is impossible in the validation condition in line D5 and the CAS in line D8, even if the value of **prev* has changed since last read in line D1 (or line D3 for subsequent loop executions). The removal and reinsertion of *cur*ˆ after D1 and before E2 do not cause the ABA problem in D5 or D8. The hazardous sequence of events that can cause the ABA problem in D5 and D8 is if *cur*ˆ is removed and then reinserted in the list after line D3 and before D5 or D8. The insertion and removal of other nodes between **prev* and *cur*ˆ never causes the ABA problem in D5 and D8. Thus, by preventing *cur*ˆ from being removed and reinserted during the current thread's execution of D3–D5 or D3–D8, SMR makes the ABA problem impossible in lines D5 and D8.

Lines E3, E4, E5, and E6 serve to prevent *cur*ˆ in the next iteration of the loop (if any) from being removed and reinserted during the current thread's execution of D3–D5 or D3–D8, and also to guarantee that if the current thread accesses *cur*ˆ in the next iteration in lines D3 and D4, then *cur*ˆ is not free.

Lines E3 and E4 must be between lines D3 and D5. Lines E5 and E6 can either immediately precede or immediately follow lines D7 and D9, respectively.

The protection of *cur*ˆ in one iteration continues in the next iteration for protecting the node containing **prev*, such that it is guaranteed that when the current thread accesses **prev* in lines D5 and D8, that node is not free. The same protections of **prev*, *cur*ˆ and *next*ˆ continue through the execution of lines A3, B2, and B3.

The three hazard pointers *hp0, *hp1, and *hp2 shadow the movement of the three private pointers *next*, *cur*, and *prev* down the list, respectively (*hp2 holds the address of the node including **prev* not the value *prev*). The movement of the pointers resembles that of a worm with three segments, with *next*, *cur*, and *prev* as the the head, midsection, and tail, respectively. In order to advance through the list, a thread assigns a variant of the value of *cur* (&*cur*ˆ.⟨*Mark,Next*⟩) to *prev*, then assigns *next* to *cur* and finally advances *next*. SMR requires that if any hazard pointer inherits its value from another, then the index of the latter must be less than that of the former in the shared hazard pointer array. This is needed because the SMR algorithm scans the hazard pointer array in ascending index order, non-atomically, i.e., one hazard pointer at a time [17]. Therefore, the indices of *hp0, *hp1, and *hp2 must be in ascending order, respectively.

Finally, it is worth noting that in the new algorithm, the *Key* field of a dynamic node does not need to be preserved after the node's removal. Therefore, that field can be reused by the SMR algorithm to link deleted nodes, thus offering significant space savings.

Also, the bucket headers only need one word per bucket. The ⟨*Mark,Next*⟩ field in dynamic nodes only needs to occupy one word, and no ABA tags are needed. Also, as an additional advantage, SMR allows the new algorithm to be completely dynamic using only single word CAS or restricted LL/SC.

4. CORRECTNESS

For brevity, we provide only informal proof sketches, with lemmas indicating the proof roadmap. First we introduce some definitions.

For all times t, a node is in the list at t, iff at t it is reachable by following the *Next* pointers of reachable nodes starting from *head*ˆ.*Next*.

For all times t, the list is in state $S_{n,m}$ iff the following are all true:

1. $|\{\text{node } x\text{: at } t,\ x \text{ is in the list} \wedge x \text{ is not marked.}\}| = n$.
2. $|\{\text{node } x\text{: at } t,\ x \text{ is in the list} \wedge x \text{ is marked.}\}| = m$.
3. $\forall$ nodes x, y in the list, x.Next$= y \implies x$.Key$< y$.Key.

For example, a list is in state $S_{0,0}$ if it contains no nodes (i.e., its head pointer is null). A list is in state $S_{2,1}$ if it contains exactly three nodes, one is marked as deleted and the other two are not.

[5]In this context, the term deleted refers to calls to DeleteNode and is not related to the HashDelete or list Delete operations. A deleted node is one that was passed as an argument of DeleteNode.

```
// types and structures
structure NodeType {
    Key : KeyType;
    ⟨Mark,Next⟩ : ⟨boolean,*NodeType⟩;}
structure MarkPtrType {
    ⟨Mark,Next⟩ : ⟨boolean,*NodeType⟩;}

// Shared variables
T[M] : MarkPtrType; // Initially ⟨0,null⟩
// private variables
prev : *MarkPtrType;
cur,next : *NodeType;

// No change in HashInsert, HashDelete, HashSearch.

Find(head:*MarkPtrType;key:KeyType) : boolean {
try_again:
    prev ← head;
D1: ⟨pmark,cur⟩ ← *prev;
E1: *hp1 ← cur;
E2: if *prev ≠ ⟨0,cur⟩ goto try_again;
    while true {
D2:     if cur = null return false;
D3:     ⟨cmark,next⟩ ← cur^.⟨Mark,Next⟩;
E3:     *hp0 ← next;
E4:     if cur^.⟨Mark,Next⟩ ≠ ⟨cmark,next⟩ goto try_again;
D4:     ckey ← cur^.Key;
D5:     if *prev ≠ ⟨0,cur⟩ goto try_again;
        if !cmark {
D6:         if ckey≥key return ckey = key;
D7:         prev ← &cur^.⟨Mark,Next⟩;
E5:         *hp2 ← cur;
        } else {
D8:         if CAS(prev,⟨0,cur⟩,⟨0,next⟩)
                DeleteNode(cur); else goto try_again;
        }
D9:     cur ← next;
E6:     *hp1 ← next;
    }
}
```

```
// SMR related variables
// static private variables
hp0,hp1,hp2 : **NodeType;
// HP is the shared array of hazard pointers
// j is thread id for SMR purposes
hp0 = &HP[3*j]
hp1 = &HP[3*j+1]
hp2 = &HP[3*j+2]
// The order is important

Insert(head:*MarkPtrType,node:*NodeType):boolean {
    key ← node^.Key;
    while true {
A1:     if Find(head,key) {result ← false; break;}
A2:     node^.⟨Mark,Next⟩ ← ⟨0,cur⟩;
A3:     if CAS(prev,⟨0,cur⟩,⟨0,node⟩)
            {result ← true; break;}
    }
    *hp0 ← null; *hp1 ← null; *hp2 ← null;
    return result;
}

Delete(head:*MarkPtrType,key:KeyType):boolean {
    while true {
B1:     if !Find(head,key) {result ← false; break;}
B2:     if !CAS(&cur^.⟨Mark,Next⟩,
                ⟨0,next⟩,
                ⟨1,next⟩) continue;
B3:     if CAS(prev,⟨0,cur⟩,⟨0,next⟩)
            DeleteNode(cur); else Find(head,key);
        result ← true; break;
    }
    *hp0 ← null; *hp1 ← null; *hp2 ← null;
    return result;
}

Search(head:*MarkPtrType,key:KeyType):boolean {
    result ← Find(head,key);
    *hp0 ← null; *hp1 ← null; *hp2 ← null;
    return result;
}
```

Figure 7: Version of the new algorithm using the SMR method [17] (SMR-related code is in a different font).

A list in state $S_{n,m}$ corresponds to an abstract set K with n elements, such that for all nodes x in the list that are not marked, x.Key $\in K$.

LEMMA 1. *$\forall$nodes x, y in the list, $x \neq y \implies x.Key \neq y.Key$.*

LEMMA 2. *The Key field of a node never changes while the node is in the list.*

LEMMA 3. *Once set, the Mark field of a node remains set until the node's removal from the list.*

LEMMA 4. *A node is never removed from the list while its Mark field is clear.*

Safety

To prove safety, we use the state definitions and the state transition diagram of Figure 8. The list is in a valid state, iff it matches the definition of some state $S_{n,m}$. The state of the list changes only on the success of the CAS operations in lines A3, B2, B3, and D8. Our goal is to prove that the following claim is always true.

CLAIM 1. *The list is in a valid state, and if a CAS succeeds then a correct transition occurs as shown in the state transition diagram in Figure 8, and the transition is consistent with the abstract set semantics.*

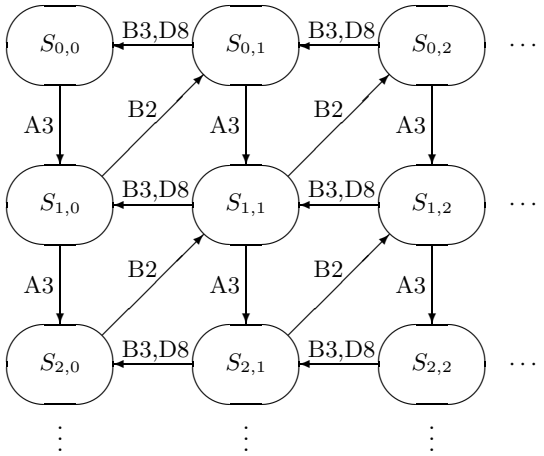


Figure 8: State transition diagram.

Initially, Claim 1 is true, assuming that the list starts in a valid state, e.g., $S_{0,0}$. For a proof by induction, we need to show that whenever a CAS operation succeeds at time t, and

Claim 1 (the induction hypothesis) has been true up to that time, then only a correct transition can take place and the transition is consistent with the abstract set semantics.

All the following theorems and lemmas are predicated on the assumption that Claim 1 has been true for all times before the success of the CAS operation or validation condition in question at time t.

LEMMA 5. *At time t, the validation condition in line D5 succeeds* $\wedge$ *prev*$\neq$*head* $\Longrightarrow$ *at t,* $\exists$*node x in the list :: prev=&x.*$\langle$*Mark,Next*$\rangle$ $\wedge$ *key*$>$ *x.Key.*

Informally. on the success of the validation condition in D5, **prev* is in the list.

LEMMA 6. *At time t, the validation condition in line D5 succeeds* $\wedge$ *cur*$\neq$*null* $\Longrightarrow$ *at t, node cur*$\hat{}$ *is in the list.*

LEMMA 7. *The CAS in line A3 succeeds* $\Longrightarrow$ *for all times since the current thread last executed the validation condition in line D5, prev*$\hat{}$*.Mark is clear.*

LEMMA 8. *The CAS in line A3 succeeds* $\Longrightarrow$ *for all times since the current thread last executed the validation condition in line D5, cur*$\hat{}$ *is in the list* $\wedge$ *cur*$\hat{}$*.Key*$>$*key.*

LEMMA 9. *The CAS in line B2 succeeds* $\Longrightarrow$ *for all times since the current thread last executed the validation condition in line D5, cur*$\hat{}$ *is in the list* $\wedge$ *cur*$\hat{}$*.Key=key.*

LEMMA 10. *The CAS in line B3 succeeds* $\Longrightarrow$ *for all times since the current thread last executed the validation condition in line D5, prev*$\hat{}$*.Mark is clear.*

LEMMA 11. *The CAS in line B3 succeeds* $\Longrightarrow$ *for all times since the current thread last executed the CAS in line B2 successfully, cur*$\hat{}$*.Next=next.*

LEMMA 12. *The CAS in line D8 succeeds* $\Longrightarrow$ *for all times since the current thread last executed the validation condition in line D5, prev*$\hat{}$*.Mark is clear.*

LEMMA 13. *The CAS in line D8 succeeds* $\Longrightarrow$ *for all times since the current thread last read cur*$\hat{}$$\langle$*Mark,Next*$\rangle$ *in line D3, node cur*$\hat{}$ *is in the list* $\wedge$ *cur*$\hat{}$$\langle$*Mark,Next*$\rangle$=$\langle$*1,next*$\rangle$*.*

THEOREM 1. *If successful, the CAS in line A3 takes the list to a valid state and inserts the new key into the set.*

THEOREM 2. *If successful, the CAS in line B2 takes the list to a valid state and removes cur*$\hat{}$*.Key from the set.*

THEOREM 3. *If successful, the CAS in line B3 takes the list to a valid state and does not modify the set.*

THEOREM 4. *If successful, the CAS in line D8 takes the list to a valid state and does not modify the set.*

THEOREM 5. *Claim 1 is true at all times.*

Lock-Freedom

LEMMA 14. *Whenever one of the CAS operations or validation conditions in lines A3, B2, D5, and D8 (also E2 and E4 when SMR is used) fails, then the state of the list must have changed since the current thread last executed line D1.*

LEMMA 15. *If the list is in state* $S_{n,m}$ *and then the state of the list changes* $m+1$ *times, then at least one operation (Insert, Delete, or Search) on the list must have succeeded during that period.*

LEMMA 16. *If a thread starts an operation (Insert, Delete, or Search) on the list when it is in state* $S_{n,m}$ *and then executes line D1* $m+2$ *times, then some operation on the list must have completed successfully since the start of the current operation.*

LEMMA 17. *If a thread starts an operation on the list when it is in state* $S_{n,m}$ *and then executes line D2* $n.m+2$ *times, then some operation on the list must have completed successfully since the start of the current operation.*

THEOREM 6. *The new algorithm is lock-free.*

Linearizability

An implementation of an object is linearizable if it can always give an external observer, observing only the abstract object operations, the illusion that each of these operations takes effect instantaneously at some point between its invocation and its response [9].

The new algorithm is linearizable. since every operation on the list has a specific linearization point, where it takes effect. By the safety properties and the definition of $S_{n,m}$, it can be shown that the responses of the list operations are consistent with the state of the abstract set object at these points. The following are the linearization points:

- Every Search and Delete operation that returns false, after searching an empty list, takes effect on its last reading of **head* in line D1.
- Every Search and Delete operation that returns false, after searching a non-empty list, takes effect on its last reading of *cur*$\hat{}$$\langle$*Mark,Next*$\rangle$ in line D3.
- Every Insert operation that returns false takes effect on its last reading of *cur*$\hat{}$$\langle$*Mark,Next*$\rangle$ in line D3.
- Every Search operation that returns true takes effect on its last reading of *cur*$\hat{}$$\langle$*Mark,Next*$\rangle$ in line D3.
- Every Insert operation that returns true takes effect on its only successful execution of the CAS in line A3.
- Every Delete operation that returns true takes effect on its only successful execution of the CAS in line B2.

THEOREM 7. *The new algorithm is linearizable.*

5. PERFORMANCE

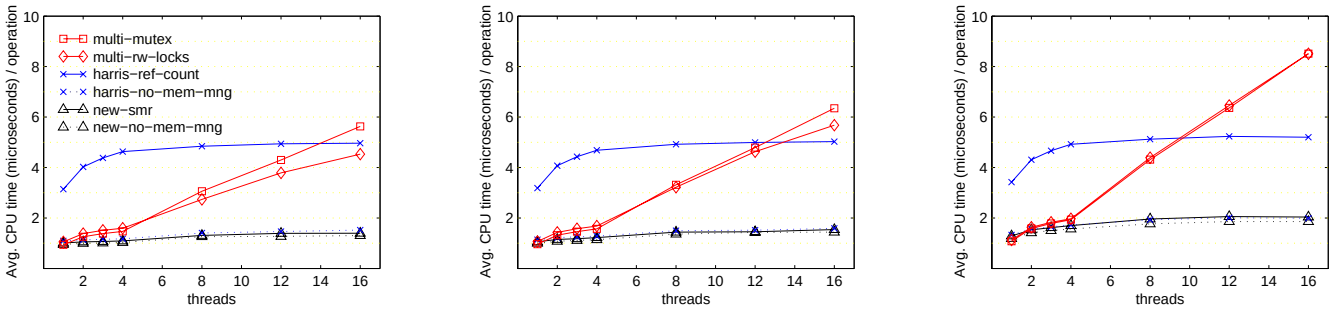
We used a 4-processor (375 MHz 604e) IBM PowerPC multiprocessor to evaluate the performance of the new lock-free hash table algorithm, relative to hash tables that use Harris' lock-free list algorithm and various lock-based implementations.

For the new algorithm, we used two implementations, one with memory management and one without. For memory management, we used the safe memory reclamation (SMR) method [17]. Freelists with ABA tags [10] can also be used efficiently with the new algorithm, but we used SMR as it requires only single-word atomic operations. The implementation without memory management is impractical. We include this implementation only as a means to determine memory management cost for the new algorithm. For this implementation, we preallocated all free nodes before the beginning of each experiment and nodes removed from the hash table were not reused.

For Harris' algorithm, we also used two implementations, with and without memory management. However, for memory management, Harris' algorithm cannot use SMR or freelists. The only lock-free OS-independent memory management methods that can be used with Harris' algorithm are the reference counting methods of Valois'[25] and Detlefs *et. al.* [4]. The latter requires DCAS which renders it impractical, while the former uses only CAS. We used Valois' method with corrections we employed in earlier work [18], and applied it with extreme care to eliminate unnecessary manipulations of reference counters that would otherwise degrade performance even further if automatic transformations were employed.

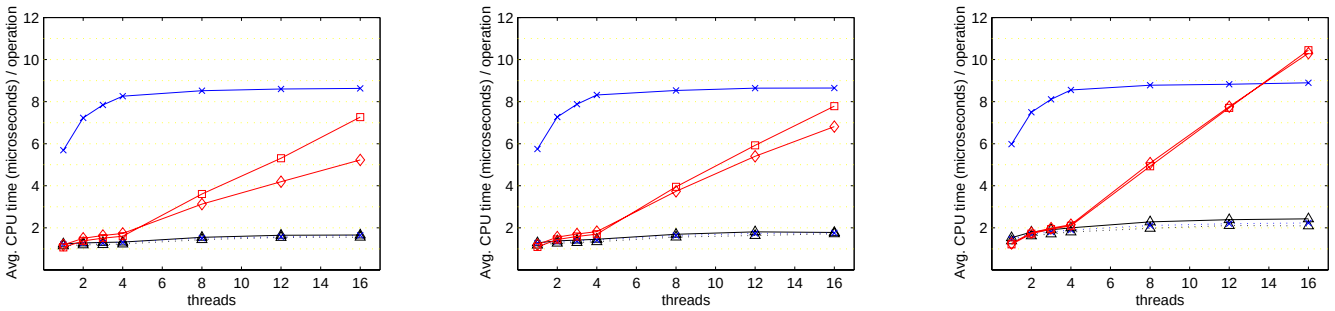
We used four lock-based implementations. Two implementations with global locks for the whole hash table object. The other two implementations associate a lock with every hash bucket, in order to increase concurrency. In both cases, one implementation used mutual exclusion locks and the other used reader-writer locks for allowing concurrent read-only accesses (i.e., Search operations). For mutual exclusion locks, we used the Test-and-Test-and-Set lock [21]. For reader-writer locks, we used a variant of a simple centralized algorithm by Mellor-Crummey and Scott [16].

All lock and CAS operations were implemented in short assembly language routines using LL/SC with minimal function



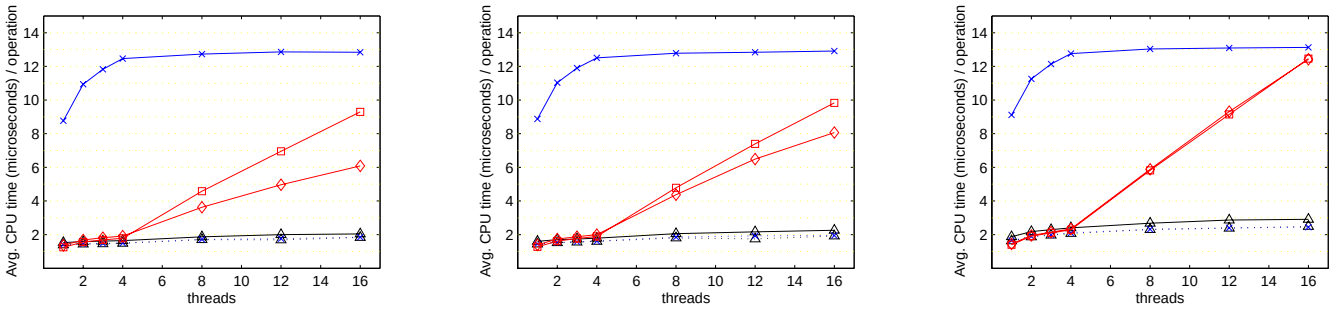
(a) 5% Insert, 5% Delete, 90% Search (b) 10% Insert, 10% Delete, 80% Search (c) $\frac{1}{3}$ Insert, $\frac{1}{3}$ Delete, $\frac{1}{3}$ Search

Figure 9: Average execution time of operations on a hash table with 100 buckets and average load factor 1.



(a) 5% Insert, 5% Delete, 90% Search (b) 10% Insert, 10% Delete, 80% Search (c) $\frac{1}{3}$ Insert, $\frac{1}{3}$ Delete, $\frac{1}{3}$ Search

Figure 10: Average execution time of operations on a hash table with 100 buckets and average load factor 5.



(a) 5% Insert, 5% Delete, 90% Search (b) 10% Insert, 10% Delete, 80% Search (c) $\frac{1}{3}$ Insert, $\frac{1}{3}$ Delete, $\frac{1}{3}$ Search

Figure 11: Average execution time of operations on a hash table with 100 buckets and average load factor 10.

call overheads. For all algorithms, when appropriate cache line padding was used to eliminate false sharing. The algorithms were compiled using the highest optimization level. In all experiments, all memory needs fit in physical memory.

We generated workloads of operations (Insert, Delete and Search) by choosing random keys uniformly from a range of integers. We varied several parameters: the number of hash buckets m, the mix of operations, and the range of keys U. We controlled the average load factor α (average number of keys per hash bucket) indirectly by initializing the hash table to include αm keys and by selecting the size of U to be equal to $2\alpha m$.

In all experiments we varied the number of threads operating concurrently on the hash table. In all cases, each thread performed 1,000,000 operations. We measured the total CPU time used by all threads to execute these operations. The pseudo-random sequences for different threads were non-overlapping in each experiment, but were repeatable for each thread for fairness in comparing different algorithms.

We conducted many experiments with various parameters. All showed the same general trends. For brevity and clarity, we include only results of representative experiments using a hash table with 100 buckets.

Figures 9, 10 and 11 show the average execution time per operation for a hash table with various average load factors and operation mixes. By initializing the hash table to include 100, 500, and 1000 keys, and setting $|U|$ to 200, 1000 and 2000, the hash table had average load factors α of 1, 5 and 10, respectively.

For clarity, we omit the results for the single lock implementation, since as expected they exhibit significantly inferior performance in comparison to other implementations, including those using fine-grain locks.

As expected, with higher percentages of Search operations,

reader-writer locks outperform mutual exclusion locks. The significant effect of multiprogramming (more than 4 threads) on the performance of all lock-based implementations is clear, with varying degrees depending on the frequency of update operations and the level of multiprogramming.

Being lock-free, Harris' algorithm is immune to the performance degradation that affected the lock-based implementations under multiprogramming. However, its performance with memory management is substantially inferior to other practical alternatives, due to the high overhead of reference counting. Combined with the fact that its memory requirements are unbounded even with bounded object size using any memory management method, it is clear that Harris' algorithm is impractical with respect to both performance and robustness.

The new algorithm provides the best overall performance, with and without multiprogramming, with various operation mixes, and under high and low contention. It outperforms Harris' algorithm by a factor of 2.5 or more when using lock-free memory management methods, and matches or exceeds its performance under the unrealistic assumption of no memory management, which is not the same as assuming automatic garbage collection (which is likely to cost much more than SMR). The cost of memory management for the new algorithm is consistently low.

The results indicate that for general practical use, the new algorithm is the algorithm of choice for implementing shared hash tables.

6. CONCLUSIONS

Prior lock-free algorithms for sets and hash tables suffer from serious drawbacks that prevent or limit their use in practice. These drawbacks include size inflexibility, dependence on DCAS, and dependence on problematic and highly-inefficient memory management techniques.

In this paper we presented the first CAS-based lock-free list-based set algorithm that is compatible with all memory management methods. We used it as a building block of a dynamic lock-free hash table algorithm.

Our experimental results showed significant performance advantages of the new algorithm over the best known lock-free as well as lock-based hash table implementations. The new algorithm outperforms the best prior lock-free algorithm, Harris' [6], by a factor of 2.5 or more, in all lock-free cases. It generally outperforms the best lock-based implementations, with and without multiprogramming, under high and low contention, often by significant margins. The results indicate that it is the algorithm of choice for implementing shared hash tables. Also, the new algorithm offers upper bounds on its memory use relative to the set size with all lock-free as well as blocking memory management method, while Harris' algorithm cannot provide such bound even with bounded maximum set size with any memory management method.

This paper also demonstrates the significant effect of memory management characteristics of dynamic lock-free algorithms on their performance, robustness and practicality. Evaluating the merits of prior and future lock-free algorithms must take into account their compatibility with memory management methods. A dynamic lock-free algorithm cannot be considered generally practical unless it is compatible with freelists.

7. REFERENCES

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

[2] 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.

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

[4] 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.

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

[6] 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.

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

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

[9] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12(3): 463–492, July 1990.

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

[11] 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.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley 1973.

[13] Vladimir Lanin and Dennis Shasha. Concurrent Set Manipulation without Locking. In *Proceedings of the 7th ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[14] Henry Massalin and Carlton Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report No. CUCS-005-91, Columbia University, 1991.

[15] 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.

[16] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113, April 1991.

[17] Maged M. Michael. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, July 2002.

[18] 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.

[19] 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.

[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] Larry Rudolph and Zary Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 340–347, June 1984.

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

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

[24] 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.

[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.