Split-Ordered Lists: Lock-Free Extensible Hash Tables

ORI SHALEV

Tel-Aviv University, Tel-Aviv, Israel

AND

NIR SHAVIT

Tel-Aviv University and Sun Microsystems Laboratories, Tel-Aviv, Israel

Abstract. We present the first lock-free implementation of an extensible hash table running on current architectures. Our algorithm provides concurrent insert, delete, and find operations with an expected $O(1)$ cost. It consists of very simple code, easily implementable using only load, store, and compare-and-swap operations. The new mathematical structure at the core of our algorithm is *recursive split-ordering*, a way of ordering elements in a linked list so that they can be repeatedly "split" using a single compare-and-swap operation. Metaphorically speaking, our algorithm differs from prior known algorithms in that extensibility is derived by "moving the buckets among the items" rather than "the items among the buckets." Though lock-free algorithms are expected to work best in multiprogrammed environments, empirical tests we conducted on a large shared memory multiprocessor show that even in non-multiprogrammed environments, the new algorithm performs as well as the most efficient known lock-based resizable hash-table algorithm, and in high load cases it significantly outperforms it.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.4.1 [**Operating Systems**]: Process Management—*Synchronization*; *concurrency*; *multiprocessing/multiprogramming/multitasking*; E.2 [**Data Storage Representation**]—*Hash-table representations*

General Terms: Algorithms, Theory, Performance, Experimentation

Additional Key Words and Phrases: Concurrent data structures, hash table, non-blocking synchronization, compare-and-swap

---

This work was performed while N. Shavit was at Tel-Aviv University, supported by a Collaborative Research Grant from Sun Microsystems.

A preliminary version of this article appeared in *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (Boston, MA), ACM, New York, 2003, pp. 102–111.

Authors' address: School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel 69978, e-mail: orish@post.tau.ac.il; shanir@sun.com.

1. *Introduction*

Hash tables, and specifically extensible hash tables, serve as a key building block of many high performance systems. A typical extensible hash table is a continuously resized array of buckets, each holding an expected constant number of elements, and thus requiring an expected constant time for insert, delete and find operations [Cormen et al. 2001]. The cost of resizing, the redistribution of items between old and new buckets, is amortized over all table operations, thus keeping the average complexity of any one operation constant. As this is an extensible hash table, "resizing" means extending the table. It is interesting to note, as argued elsewhere [Hsu and Yang 1986; Lea (e-mail communication 2005)], that many of the standard concurrent applications using hash tables require tables to only increase in size."

We are concerned in implementing the hash table data structure on multiprocessor machines, where efficient synchronization of concurrent access to data structures is essential. Lock-free algorithms have been proposed in the past as an appealing alternative to lock-based schemes, as they utilize strong primitives such as CAS (*compare-and-swap*) to achieve fine grained synchronization. However, lock-free algorithms typically require greater design efforts, being conceptually more complex.

This article presents the first lock-free extensible hash table that works on current architectures, that is, uses only loads, stores and CAS (or LL/SC [Moir 1997]) operations. In a manner similar to sequential linear hashing [Litwin 1980] and fitting real-time[1] applications, resizing costs are split incrementally to achieve expected $O(1)$ operations per insert, delete and find. The proposed algorithm is simple to implement, leading us to hope it will be of interest to practitioners as well as researchers. As we explain shortly, it is based on a novel *recursively split-ordered* list structure. Our empirical testing shows that in a concurrent environment, even without multiprogramming, our lock-free algorithm performs as well as the most efficient known lock-based extensible hash-table algorithm due to Lea [2003], and in high-load cases, it significantly outperforms it.

1.1. Background.

There are several lock-based concurrent hash table implementations in the literature. In the early eighties, Ellis [1983, 1987] proposed an extensible concurrent hash table for distributed data based on a two level locking scheme, first locking a table directory and then the individual buckets. Michael [2002a] has recently shown that on shared memory multiprocessors, simple algorithms using a reader-writer lock [Mellor-Crummey and Scott 1991] per bucket have reasonable performance for non-extensible tables. However, to resize one would have to hold the locks on all buckets simultaneously, leading to significant overheads. A recent algorithm by Lea [2003], proposed for *java.util.concurrent*, the Java™ Concurrency Package, is probably the most efficient known concurrent extensible hash algorithm. It is based on a more sophisticated locking scheme that involves a small number of high level locks rather than a lock per bucket, and allows concurrent searches while resizing the table, but not concurrent inserts or deletes. In general, lock-based hash-table algorithms are expected to suffer from the typical drawbacks of blocking synchronization: deadlocks, long delays, and

[1] In this article, by *real-time* we mean *soft real-time* [Buttazzo et al. 2005], where some flexibility on the real-time requirements is allowed.

priority inversions [Greenwald 1999]. These drawbacks become more acute when performing a *resize* operation, an elaborate "global" process of redistributing the elements in all the hash table's buckets among newly added buckets. Designing a lock-free extensible hash table is thus a matter of both practical and theoretical interest.

Michael [2002a], builds on the work of Harris [2001] to provide an effective compare-and-swap (CAS) based lock-free linked-list algorithm (which we will elaborate upon in the following section). He then uses this algorithm to design a lock-free hash structure: a fixed size array of hash buckets with lock-free insertion and deletion into each. He presents empirical evidence that shows a significant advantage of this hash structure over lock-based implementations in multiprogrammed environments. However, this structure is not extensible: if the number of elements grows beyond the predetermined size, the time complexity of operations will no longer be constant.

As part of his "two-handed emulation" approach, Greenwald [2002] provides a lock-free hash table that can be resized based on a double-compare-and-swap (DCAS) operation. However, DCAS, an operation that performs a CAS atomically on two non-adjacent memory locations, is not available on current architectures. Moreover, although Greenwald's hash table is extensible, it is not a true extensible hash table. The average number of steps per operation is not constant: it involves a helping scheme where that under certain scheduling scenario would lead to a time complexity linearly dependant on the number of processes.

Independently of our work, Gao et al. [2004] have developed a extensible and "almost wait-free" hashing algorithm based on an open addressing hashing scheme and using only CAS operations. Their algorithm maintains the dynamic size by periodically switching to a global resize state in which multiple processes collectively perform the migration of items to new buckets. They suggest performing migration using a write-all algorithm [Hesselink et al. 2001]. Theoretically, each operation in their algorithm requires more than constant time on average because of the complexity of performing the write-all [Hesselink et al. 2001], and so it is not a true extensible hash-table. However, the nonconstant factor is small, and the performance of their algorithm in practice will depend on the yet-untested real-world performance of algorithms for the write-all problem [Hesselink et al. 2001; Kanellakis and Shvartsman 1997].

1.2. THE LOCK-FREE RESIZING PROBLEM. What is it that makes lock-free extensible hashing hard to achieve? The core problem is that even if individual buckets are lock-free, when resizing the table, several items from each of the "old" buckets must be relocated to a bucket among "new" ones. However, in a single CAS operation, it seems impossible to atomically move even a single item, as this requires one to remove the item from one linked list and insert it in another. If this move is not done atomically, elements might be lost, or to prevent loss, will have to be replicated, introducing the overhead of "replication management". The lock-free techniques for providing the broader atomicity required to overcome these difficulties imply that processes will have to "help" others complete their operations. Unfortunately, "helping" requires processes to store state and repeatedly monitor other processes' progress, leading to redundancies and overheads that are unacceptable if one wants to maintain the constant time performance of hashing algorithms.

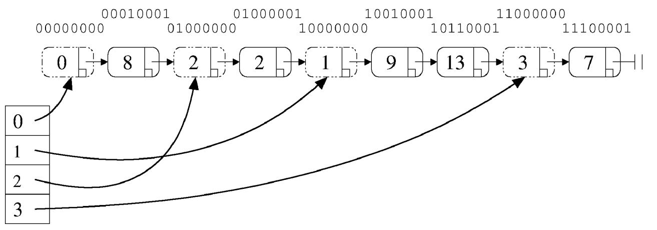


FIG. 1. A split-ordered hash table.

1.3. SPLIT-ORDERED LISTS. To implement our algorithm, we thus had to overcome the difficulty of atomically moving items from old to new buckets when resizing. To do so, we decided to, metaphorically speaking, flip the linear hashing algorithm on its head: our algorithm *will not move the items among the buckets*, rather, it *will move the buckets among the items*. More specifically, as shown in Figure 1, the algorithm keeps all the items in one lock-free linked list, and gradually assigns the bucket pointers to the places in the list where a sublist of "correct" items can be found. A bucket is initialized upon first access by assigning it to a new "dummy" node (dashed contour) in the list, preceding all items that should be in that bucket. A newly created bucket splits an older bucket's chain, reducing the access cost to its items. Our table uses a modulo 2^i hash (there are known techniques for "pre-hashing" before a modulo 2^i hash to overcome possible binary correlations among values Lea [2003]). The table starts at size 2 and repeatedly doubles in size.

Unlike moving an item, the operation of directing a bucket pointer can be done in a single CAS operation, and since items are not moved, they are never "lost". However, to make this approach work, one must be able to keep the items in the list sorted in such a way that any bucket's sublist can be "split" by directing a new bucket pointer within it. This operation must be recursively repeatable, as every split bucket may be split again and again as the hash table grows. To achieve this goal we introduced *recursive split-ordering*, a new ordering on keys that keeps items in a given bucket adjacent in the list throughout the repeated splitting process.

Magically, yet perhaps not surprisingly, recursive split-ordering is achieved by simple *binary reversal*: reversing the bits of the hash key so that the new key's most significant bits (MSB) are those that were originally its least significant. As detailed below and in the next section, some additional bit-wise modifications must be made to make things work properly. In Figure 1, the split-order key values are written above the nodes (the reader should disregard the rightmost binary digit at this point). For instance, the split-order value of 3 is the bit-reverse of its binary representation, which is 11000000. The dashed-line nodes are the special dummy nodes corresponding to buckets with original keys that are 0,1,2, and 3 modulo 4. The split-order keys of regular (nondashed) nodes are exactly the bit-reverse image of the original keys after turning on their MSB (in the example we used 8-bit words). For example, items 9 and 13 are in the "1 mod 4" bucket, which can be recursively split in two by inserting a new node between them.

To *insert* (respectively *delete* or *find*) an item in the hash table, hash its key to the appropriate bucket using recursive split-ordering, follow the pointer to the appropriate location in the sorted items list, and traverse the list until the key's proper location in the split-ordering (respectively, until the key or a key indicating the item is not in the list) is found. The solution depends on the property that the items' position is "encoded" in their binary representation, and therefore cannot be generalized to bases other than 2.

As we show, because of the combinatorial structure induced by the split-ordering, this will require traversal of no more than an expected constant number of items. A detailed proof appears in Section 3.

We note that our design is modular: to implement the ordered items list, one can use one of several non-blocking list-based set algorithms in the literature. Potential candidates are the lock-free algorithms of Harris [2001] or Michael [2002a], or the obstruction-free algorithms of Valois[2][1995] or Luchangco et al. [2003]. We chose to base our presentation on the algorithm of Michael [2002a], an extension of the Harris algorithm [Harris 2001] that fits well with memory management schemes [Herlihy et al. 2002; Michael 2002b] and performs well in practice.

1.4. COMPLEXITY. When analyzing the complexity of concurrent hashing schemes, there are two adversaries to consider: one controlling the distribution of item keys, the other controlling the scheduling of thread operations. The former appears in all hash table algorithms, sequential or concurrent, while the latter is a direct result of the introduction of concurrency. We use the term *expected time* to refer to the expected number of machine instructions per operation in the worst case scheduling scenario, assuming (as is standard in the literature [Cormen et al. 2001]) a hash function of uniform distribution. We use the term *average time* to refer to the number of machine instructions per operation averaged over all executions, also assuming a uniform hash function. It follows that constant expected time implies constant average time.

As we show in Section 3, if we make the standard assumption of a hash function with a uniform distribution, then under any scheduling adversary our new algorithm provides a lock-free extensible hash table with $O(1)$ average cost per operation.

The complexity improves to expected constant time if we assume a *constant extendibility rate*, meaning that the table is never extended (doubled in size) a non-constant number of times while a thread is delayed by the scheduler. Constant expected time is an improvement over average expected time since it means that given a good hash function, the adversary cannot cause any single operation to take more than a constant number of steps.

One feature in which the new algorithm is similar in flavor to sequential linear hashing algorithms [Litwin 1980] (in contrast to all the above algorithms [Gao et al. 2004; Greenwald 2002; Lea 2003]) is that resizing is done incrementally and only bad distributions (ones that have very low probability given a uniform hash function) or extreme scheduling scenarios can cause the cost of an operation to exceed constant time. This possibly makes the algorithm a better fit for soft real-time applications [Buttazzo et al. 2005] where relaxable timing deadlines need to be met.

1.5. PERFORMANCE. We tested our new *split-ordered list* hash algorithm against the most-efficient known lock-based implementation due to Lea [2003]. We created an optimized `C++` based version of the algorithm and compared it to split-ordered lists using a collection of tests executed on a 72-node shared memory machine. We present experiments in Section 4 that show that split-ordered lists

[2] Valois' algorithm was labeled "lock-free" by mistake. It is livelock-prone.

perform as well as Lea's algorithms, even in nonmultiprogrammed cases, although lock-free algorithms are expected to benefits systems mainly in multiprogrammed environments. Under high loads, they significantly outperform Lea's algorithm, exhibiting up to four times higher throughput. They also exhibit greater robustness, for example in experiments where the hash function is biased to create nonuniform distributions.

The remainder of this article is organized as follows: In the next section, we describe the background and the new algorithm in depth. In Section 3, we present the full correctness proof. In Section 4, the empirical results are presented and discussed.

2. *The Algorithm in Detail*

Our hash table data structure consists of two interconnected substructures (see Figure 1): A linked list of nodes containing the stored items and keys, and an expanding array of pointers into the list. The array entries are the logical "buckets" typical of most hash tables. Any item in the hash table can be reached by traversing down the list from its head, while the bucket pointers provide shortcuts into the list in order to minimize the search cost per item.

The main difficulty in maintaining this structure is in managing the continuous coverage of the full length of the list by bucket pointers as the number of items in the list grows. The distribution of bucket pointers among the list items must remain dense enough to allow constant time access to any item. Therefore, new buckets need to be created and assigned to sparsely covered regions in the list.

The bucket array initially has size 2, and is doubled every time the number of items in the table exceeds $size \cdot L$, where L is a small integer denoting the *load factor*, the maximum number of items one would expect to find in each logical bucket of the hash table. The initial state of all buckets is *uninitialized*, except for the bucket of index 0, which points to an empty list, and is effectively the head pointer of the main list structure. Each bucket goes through an initialization procedure when first accessed, after which it points to some node in the list.

When an item of key k is inserted, deleted, or searched for in the table, a hash function modulo the table size is used, that is, the bucket chosen for item k is $k \ mod \ size$. The table size is always equal to some power 2^i, $i \geq 1$, so that the bucket index is exactly the integer represented by the key's i least significant bits (LSBs). The hash function's dependency on the table *size* makes it necessary to take special care as this size changes: an item that was inserted to the hash table's list before the resize must be accessible, after the resize, from both the buckets it already belonged to and from the new bucket it will logically belong to given the new hash function.

2.1. RECURSIVE SPLIT-ORDERING.

The combination of a modulo-size hash function and a 2^i table size is not new. It was the basis of the well known sequential extensible Linear Hashing scheme proposed by Litwin [1980], was the basis of the two-level locking hash scheme of Ellis [1983], and was recently used by Lea [2003] in his concurrent extensible hashing scheme. The novelty here is that we use it as a basis for a combinatorial structure that allows us to repeatedly "split" all the items among the buckets without actually changing their position in the main list.

When the table size is 2^i, a logical table bucket b contains items whose keys k maintain $k \ mod \ 2^i = b$. When the size becomes 2^{i+1}, the items of this bucket are split into two buckets: some remain in the bucket b, and others, for which $k \ mod \ 2^{i+1} = b + 2^i$, migrate to the bucket $b + 2^i$. If these two groups of items were to be positioned one after the other in the list, splitting the bucket b would be achieved by simply pointing bucket $b + 2^i$ after the first group of items and before the second. Such a manipulation would keep the items of the second group accessible from bucket b as desired.

Looking at their keys, the items in the two groups are differentiated by the i'th binary digit (counting from right, starting at 0) of their items' key: those with 0 belong to the first group, and those with 1 to the second. The next table doubling will cause each of these groups to split again into two groups differentiated by bit $i + 1$, and so on. For example, the elements 9 ($1001_{(2)}$) and 13 ($1101_{(2)}$) share the same two least significant bits (01). When the table size is 2^2, they are both in the same bucket, but when it grows to 2^3, having a different third bit will cause to to be separated. This process induces *recursive split-ordering*, a complete order on keys, capturing how they will be repeatedly split among logical buckets. Given a key, its order is completely defined by its bit-reversed value.

Let us now return to the main picture: an exponentially growing array of (possibly uninitialized) buckets maps to a linked list ordered by the split-order values of inserted items' keys, values that are derived by reversing the bits of the original keys. Buckets are initialized when they are accessed for the first time. List operations such as `insert`, `delete` or `find` are implemented via a linearizable lock-free linked list algorithm. However, having additional references to nodes from the bucket array introduces a new difficulty: it is nontrivial to manage deletion of nodes pointed to by bucket pointers. Our solution is to add an auxiliary dummy node per bucket, preceding the first item of the bucket, and to have the bucket pointer point to this dummy node. The dummy nodes are not deleted, which helps keep things simple.

In more detail, when the table size is 2^{i+1}, the first time bucket $b + 2^i$ is accessed, a dummy node is created, holding the key $b + 2^i$. This node is inserted to the list via bucket b, the *parent* bucket of $b + 2^i$. Under split-ordering, $b + 2^i$ precedes all keys of bucket $b + 2^i$, since those keys must end with $i + 1$ bits forming the value $b + 2^i$. This value also succeeds all the keys of bucket b that do not belong to $b + 2^i$: they have identical i LSBs, but their bit numbered i is "0". Therefore, the new dummy node is positioned in the exact location in the list that separates the items that belong to the new bucket from other items of bucket b. In the case where the parent bucket b is uninitialized, we apply the initialization procedure on it recursively before inserting the dummy node. In order to distinguish dummy keys from regular ones we set the most significant bit of regular keys to "1", and leave the dummy keys with "0" at the MSB. Figure 2 defines the complete split-ordering transformation using the functions `so_regularkey` and `so_dummykey`. The former, reverses the bits after turning on the MSB, and the latter simply performs the bit reversal.[3]

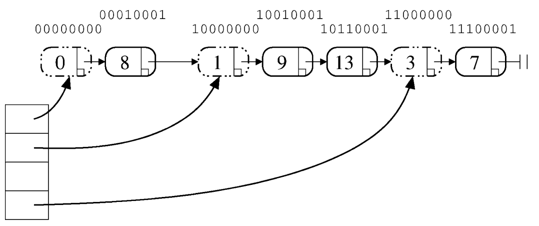
Figure 3 describes a bucket initialization caused by an insertion of a new key to the set. The insertion of key 10 is invoked when the table size is 4 and buckets 0,1 and 3 are already initialized.

[3] An efficient implementation of the `REVERSE` function utilizes a 2^8 or 2^{16} lookup table holding the bit-reversed values of $[0..2^8 - 1]$ or $[0..2^{16} - 1]$ respectively.

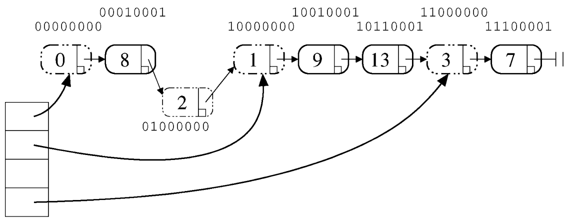
```
so_key_t so_regularkey(key_t key) {
   return REVERSE(key OR 0x8000...0000);
}

so_key_t so_dummykey(key_t key) {
   return REVERSE(key);
}
```

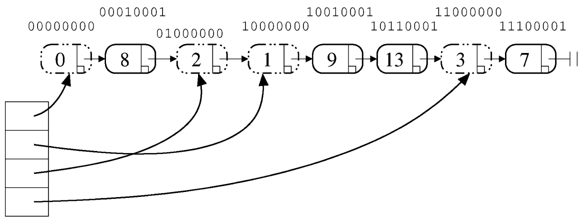
FIG. 2. The Split-Ordering Transformation. The function `so_regularkey` computes the split-order value for regular nodes, where the MSB is set before reversing the bits. The split-order value of dummy nodes is the exact bit reverse of the key.



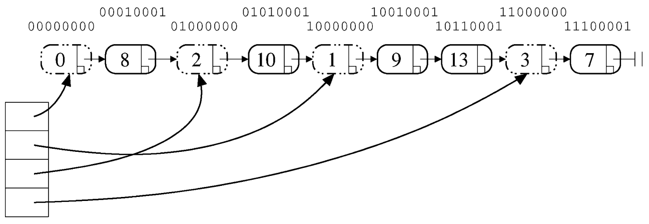
(a) Buckets 0,1 and 3 are initialized. Bucket 2 is uninitialized



(b) Insert(10) is invoked, requiring bucket 2's initialization.
A new dummy node is inserted, with split–order key of 2.



(c) Bucket 2 is assigned to the new dummy node



(d) The split–order regular key 10 is inserted to bucket 2

FIG. 3. Insertion into the split-ordered list.

Since the bucket array is growing, it is not guaranteed that the parent bucket of an uninitialized bucket is initialized. In this case, the parent has to be initialized (recursively) before proceeding. Though the total complexity in such a series of recursive calls is potentially logarithmic, our algorithm still works. This is because given a uniform distribution of items, the chances of a logarithmic-size series of recursive initialization calls are low, and in fact, the expected length of such a bad sequence of parent initializations is constant.

2.2. THE CONTINUOUSLY GROWING TABLE. We can now complete the presentation of our algorithm. We use the lock-free ordered linked-list algorithm of Michael [2002a] to maintain the main linked list with items ordered based on the split-ordered keys. This algorithm is an improved variant, including improved memory management, of an algorithm by Harris [2001]. Our presentation will not discuss the various memory reclamation options of such linked-list schemes, and we refer the interested reader to Harris [2001], Herlihy et al. [2002], and Michael [2002a, 2002b]. To keep our presentation self contained, we provide in Appendix A the code of Michael's linked list algorithm. This implementation is linearizable, implying that each of these operations can be viewed as happening atomically at some point within its execution interval.

Our algorithm decides to double the table size based on the average bucket load. This load is determined by maintaining a shared counter that tracks the number of items in the table. The final detail we need to deal with is how the array of buckets is repeatedly extended. To simplify the presentation, we keep the table of buckets in one continuous memory segment as depicted in Figure 4. This approach is somewhat impractical, since table doubling requires one process to reallocate a very large memory segment while other processes may be waiting. The practical version of this algorithm, which we used for performance testing, actually employs an additional level of indirection in accessing buckets: a main array points to segments of buckets, each of which is a bucket array. A segment is allocated only upon the first access to some bucket within it. The code for this dynamic allocation scheme appears in Section 2.4.

2.3. THE CODE. We now provide the code of our algorithm. Figure 4 specifies some type definitions and global variables. The accessible shared data structures are the array of buckets `T`, a variable `size` storing the current table size, and a counter `count` denoting the number of `regular` keys currently inside the structure.[4] The counter is initially 0, and the buckets are set as *uninitialized*, except the first one, which points to a node of key 0, whose `next` pointer is set to `NULL`. Each thread has three private variables `prev`, `cur`, and `next`, that point at a currently searched node in the list, its predecessor, and its successor. These variables have the same functionality as in Michael's algorithm [Michael 2002a]: they are set by `list_find` to point at the nodes around the searched key, and are subsequently used by the same thread to refer to these nodes in other functions. In Figure 5, we show the implementation of the `insert`, `find` and `delete` operations. The `fetch-and-inc` operation can be implemented in a lock-free manner via a simple repeated loop of

[4] Though for the sake of brevity, we do not mention it in the presented code, to reduce contention, we have threads accumulate updates locally and update the shared counter `count` only periodically. We included this optimization in the code used in our benchmarks.

```
struct MarkPtrType {      // Markable pointer type
  <mark, next>: <bool, NodeType* >;
};

struct NodeType {         // Node: contains key and next pointer
  so_key_t key;
  MarkPtrType <mark, next>;
};

/* shared variables */
MarkPtrType* T[ ];        // buckets
uint count;               // total item count
uint size;                // current table size


/* thread-private variables */

MarkPtrType *prev;          /* prev */

MarkPtrType <pmark, cur>;  /* curr */

MarkPtrType <cmark, next>; /* next */
```

FIG. 4. Types and Structures. The angular brackets notation denotes a single word type divided to the two fields `mark` and `next`. `mark` is a single bit, while the size of `next` is the rest.

CAS operations, which as we show, given the low access rates, has a negligible performance overhead.

The function `insert` creates a new node and assigns it a split-order key. Note that the keys are stored in the nodes in their split-order form. The bucket index is computed as `key` *mod* `size`. If the bucket has not been initialized yet, `initialize_bucket` is called. Then, the node is inserted to the bucket by using `list_insert`. If the insertion is successful, one can proceed to increment the item count using a `fetch-and-inc` operation. A check is then performed to test whether the load factor has been exceeded. If so, the table size is doubled, causing a new segment of uninitialized buckets to be appended.

The function `find` ensures that the appropriate bucket is initialized, and then calls `list_find` on `key` after marking it as regular and inverting its bits. `list_find` ceases to traverse the chain when it encounters a node containing a higher or equal (split-ordered) key. Notice that this node may also be a dummy node marking the beginning of a different bucket.

The function `delete` also makes sure that the `key`'s bucket is initialized. Then it calls `list_delete` to delete `key` from its bucket after it is translated to its split-order value. If the deletion succeeds, an atomic decrement of the total item count is performed.

The role of `initialize_bucket` is to direct the pointer in the array cell of the index `bucket`. The value assigned is the address of a new dummy node containing the dummy key `bucket`. First, the dummy node is created and inserted to an existing bucket, `parent`. Then, the cell is assigned the node's address. If the parent bucket is not initialized, the function is called recursively with `parent`. In order to control the recursion, we maintain the invariant that `parent` $<$ `bucket`, where "$<$" is the regular order among keys. It is also wise to choose `parent` to be as close as possible to `bucket` in the list, but still preceding it. Formally, the following constraints define

```
int insert(so_key_t key) {
I1:  node = new node(so_regularkey(key));
I2:  bucket = key % size;
I3:  if (T[bucket] == UNINITIALIZED)
I4:    initialize_bucket(bucket);
I5:  if (!list_insert(&(T[bucket]), node)) {
I6:    delete_node(node);
I7:    return 0;
     }
I8:  csize = size;
I9:  if (fetch-and-inc(&count) / csize > MAX_LOAD)
I10:   CAS(&size, csize, 2 * csize);
I11: return 1;
}

int find(so_key_t key) {
S1:  bucket = key % size;
S2:  if (T[bucket] == UNINITIALIZED)
S3:    initialize_bucket(bucket);
S4:  return list_find(&(T[bucket]),
                      so_regularkey(key));
}

int delete(so_key_t key) {
D1:  bucket = key % size;
D2:  if (T[bucket] == UNINITIALIZED)
D3:    initialize_bucket(bucket);
D4:  if (!list_delete(&(T[bucket]),
                 so_regularkey(key)))
D5:    return 0;
D6:  fetch-and-dec(&count);
D7:  return 1;
}

void initialize_bucket(uint bucket) {
B1:  parent = GET_PARENT(bucket);
B2:  if (T[parent] == UNINITIALIZED)
B3:    initialize_bucket(parent);
B4:  dummy = new node(so_dummykey(bucket));
B5:  if (!list_insert(&(T[parent]), dummy)) {
B6:    delete dummy;
B7:    dummy = cur;
     }
B8:  T[bucket] = dummy;
}
```

FIG. 5. Our split-order-based hashing algorithm.

our the algorithm's choice of `parent` uniquely, where "$<$" is the regular order and "$\prec$" is the split-order among keys:

$$\begin{aligned} \forall k \prec \texttt{bucket}, (k = \texttt{parent} \vee k \prec \texttt{parent}) \\ \texttt{parent} \prec \texttt{bucket} \\ \texttt{parent} < \texttt{bucket}. \end{aligned}$$

This value is achieved by calling the `GET_PARENT` macro that unsets `bucket`'s most significant turned-on bit. If the exact dummy key already exists in the list, it may

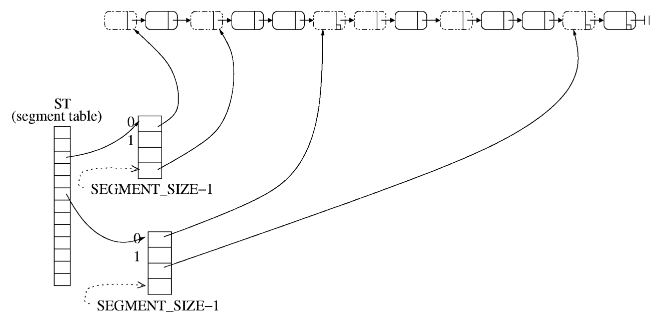


FIG. 6. Structure of the dynamic-sized table.

be the case that some other process tried to initialize the same bucket, but for some reason has not completed the second step. In this case, `list_insert` will fail, but the private variable `cur` will point to the node holding the dummy key. The newly created dummy node can be freed and the value of `cur` used. Note that when line B8 is executed concurrently by multiple threads, the value of `dummy` is the same for all of them.

As we will show in the proof, traversing the list through the appropriate bucket and dummy node will guarantee the node matching a given key will be found, or declared not-found in an expected constant number of steps.

2.4. DYNAMIC-SIZED ARRAY. Our presentation so far simplified the algorithm by keeping the buckets in one continuous memory segment. This approach is somewhat impractical, since table doubling requires one process to reallocate a very large memory segment while other processes may be waiting. In practice, we avoid this problem by introducing an additional level of indirection for accessing buckets: a "main" array points to segments of buckets, each of which is a bucket array. A segment is allocated only on the first access to some bucket within it. The structure of the dynamic-sized hash table is illustrated in Figure 6.

Applying this variation is done by replacing the array of buckets `T` by `ST`, an array of bucket segments, and accessing the table via calls to `get_bucket` and `set_bucket` as defined in Figure 7. Referring to the code of Figure 5, the lines I3, S2, D2, D4, B2, and B5 will use `get_bucket` to access the bucket, and in line B8 `set_bucket` will be called instead of the assignment. Accessing a bucket involves calculating the segment index and then the bucket index within the segment. In `get_bucket`, if the segment has not been allocated yet, it is guaranteed that the bucket was never accessed, and we can return `UNINITIALIZED`. When setting a bucket, in `set_bucket`, if the segment does not exist we have to allocate it and set its pointer in the segment table.

Asymptotically, introducing additional levels of indirection makes the cost of a single access $O(\log n)$. However, one should view the asymptotic in the context of overall memory size, which is bounded. In our case, each level extends the range exponentially with a very high constant, reaching the maximum integer value using a very shallow hierarchy. A level-4 hirarchy can exhaust the memory of a 64-bit machine. Therefore, taking memory size into consideration, the overhead of our construction can be considered as constant.

```
typedef MarkPtrType[SEGMENT_SIZE] segment_t;
segment_t ST[ ];   // the segment table

MarkPtrType * get_bucket(uint bucket) {
  segment = bucket / SEGMENT_SIZE;
  if (ST[segment] == NULL)
    return UNINITIALIZED;
  return &ST[segment][bucket % SEGMENT_SIZE];
}

void set_bucket(uint bucket, NodeType *head) {
  segment = bucket / SEGMENT_SIZE;
  if (ST[segment] == NULL) {
    new_segment = new segment_t;
    new_segment[0..SEGMENT_SIZE-1] =
      UNINITIALIZED;
    if (!CAS(&ST[segment], NULL, new_segment))
      free(new_segment);
  }
  ST[segment][bucket % SEGMENT_SIZE] = head;
}
```

FIG. 7. Dynamic sized array.

3. *Correctness Proof*

This section contains a formal proof that our algorithm has the desired properties of a resizable hash table. Our model of multiprocessor computation follows [Herlihy and Wing 1990], though for brevity, we will use operational style arguments.

Our linearizable hash table data structure implements an abstract *set* object in a lock-free way so that all operations take an expected constant number of steps on average. Our correctness proof will thus have to prove that our concurrent implementation is linearizable to a sequential set specification, that it is lock-free, and that given a "good" class of hash functions, all operations take an expected constant number of steps on average.

3.1. CORRECT SET SEMANTICS.

We begin by proving that the algorithm complies with the abstract set semantics. We use the sequential specification of a "dynamic set with dictionary operations" as defined in Cormen et al. [2001], including the three functions *insert, delete* and *find*. The *insert* operation returns 1 if the key was successfully inserted into the set, and 0 if that key already existed in the table. The *find* operation returns 1 if the key is in the set, 0 otherwise. The *delete* operation returns 1 if the key was successfully deleted from the set and 0 if it was not found.

Given a sequential specification of a set, our proof will provide specific linearization points mapping operations in our concurrent implementation to sequential operations so that the histories meet the specification.

Let *list* refer to the non-blocking ordered linked list of all items, pointed to by the buckets of the hash table. Execution histories of our algorithm include sequences of `list_find`, `list_insert`, and `list_delete` operations on this list. Though we argue about these as operations on the shared *list* and not as abstract set operations, our proof will treat these operations as atomic operations. This is a valid approach since they are linearizable by definition of the list-based set algorithms [Harris 2001; Michael 2002a]. We do however need to make additional claims about properties of operations on the *list*, since we will apply them to various "midpoints" pointed to by buckets, and not only to the start of the list as in the original use of these algorithms of Harris [2001] and Michael [2002a]. To this end, we present the following invariant, which refers to the structure of the list in any state in the execution history of our algorithm.

INVARIANT 1. *In any state:*

—*all keys in the list starting at* `T[0]` *are sorted in an ascending order.*

—*for every* $0 \leq i < size$ *if* `T[i]` *is initialized, then the node pointed by* `T[i]` *holds the key* `so_dummykey[i]` *and is reachable from* `T[0]` *by traversing the list following the nodes' next pointers.*

PROOF. Initially, the invariant holds. We will show that every operation that modifies the data structure preserves the invariant. Lines I9 and D6 manipulate the shared counter, but have no impact on the invariant. Line I10 doubles `size`, which adds new buckets, but since `size` only grows, those new buckets are uninitialized, and the invariant is unaffected.

Assuming that the invariant is true just before line I5, we will show that it is preserved. If `list_insert` fails, the shared state has not changed. Otherwise, we use the induction assumption that `T[bucket]` points to a node holding the key `so_dummykey(bucket)`, and that `node` is in the list beginning at `T[0]`. The procedure `list_insert` inserts `node` to the list `T[bucket]`. This trivially preserves the second condition of the invariant for the bucket. The new node's key is the bit reverse of `key OR 0×800...0`. The array index `bucket` and the value of `key` share the same $\log size$ least significant bits, while the rest of `bucket`'s bits are 0. Therefore, the new node's key is ordered after the first node of `T[bucket]`, whose key is the bit reverse of `bucket`. The first part is also preserved, that is, the list reachable from `T[0]` remains sorted since all keys before `T[bucket]` are by the inductive assumption ordered and have lower keys than `so_dummykey(bucket)` and so are properly positioned before the new node, and all other keys are positioned properly by the inductive assumption and the correctness of the `list_insert` operation, since they are a part of the list pointed to by `T[bucket]`.

The `list_delete` operation of line D4 only deletes a key, and thus cannot affect the order. The deleted `node` cannot be the first node of `T[bucket]`, since the least significant bit of its key is 0 and the deleted key's least significant bit is 1.

The function `list_insert` in line B5 inserts a `node` with key `so_dummykey(bucket)` to the sublist `T[parent]`, starting with a node holding `so_dummykey(parent)`. The key `parent` is defined by turning off the index `bucket`'s most significant "1" bit, so the insertion is not before the first node of the sublist starting at `T[parent]`, and as in the above proof for the case of I5, the invariant is preserved.

Finally, the assignment in B8 sets `T[bucket]` to either the dummy node created at B4, or the one assigned at B7. In the first case, since a dummy node created in line B4 is inserted, the second condition of the invariant follows immediately from the correctness of the `list_insert` operation. The first condition follows since the dummy node is inserted in order *after* its parent node which is necessarily ordered before it. In the second case, `list_insert` failed because the key `so_dummykey(bucket)` was in the list and `cur` was by the definition of *list_insert* set to the node holding that key, so both parts of the invariant follow. □

We now define the set H of keys whose items are in the hash table in any given state.

Definition 3.1. For any pointer p, let $S(p)$ be the set of keys in the sorted linked list beginning with the pointer p. Let the *hash table set*

$$H = \{\texttt{k} \mid \texttt{so\_regularkey(k)} \in S(\texttt{T[0]})\}.$$

The set H defines the abstract state of the table. For each one of the hash table operations, we will now show that one can pick a linearization point within its execution interval, so that at this point it has modified the abstract state, that is, the set H, according to the specified operation's semantics. Specifically, we will choose the following linearization points:

—the `insert` operation is linearized in line I5, at the `list_insert` operation,
—the `find` operation is linearized in line S4, at the `list_find` operation, and
—the `delete` operation is linearized in line D4, at the `list_delete` operation.

We start with the following helpful lemma:

LEMMA 3.2. *In lines I5, S4, and D4,* `T[bucket]` *is already initialized, and at B5* `T[parent]` *is already initialized.*

PROOF. All of the lines above follow a validation that `T[bucket]` is initialized. If `T[bucket]` is not initialized, `initialize_bucket` is called and the `bucket` is initialized in B8. □

Note that, in the proof above, we were not interested in whether the initialization sequence (where initializing a bucket causes initialization of the parent) actually terminates, but rather that if it did terminate then all parents of a bucket were initialized.

LEMMA 3.3. *If* `key` *is in H in line I5, then* `insert` *fails, and if it is not,* `insert` *succeeds and* `key` *joins H.*

PROOF. When `key` is in H, `so_regularkey(key)` $\in S($`T[0]`$)$. According to Lemma 3.2, `T[bucket]` is initialized, and using Invariant 1, we conclude that the node pointed by `T[bucket]` has the key `so_dummykey(bucket)` and it is a part of the list. The list is sorted, and

$$\begin{gathered}\texttt{so\_dummykey(bucket)} = \texttt{REVERSE(bucket)} = \\ \texttt{REVERSE(key mod size)} < \texttt{REVERSE(key OR 0}\times\texttt{800..0)} = \\ \texttt{so\_regularkey(key)}.\end{gathered} \tag{1}$$

Thus, the searched key is in the sublist, $S($`T[bucket]`$)$. The `list_insert` at I5 will fail and so will `insert`. If `key` is not in H, it is also not in $S($`T[bucket]`$)$, and `list_insert` inserts `so_regularkey(key)` in the `bucket`'s sublist. From that state on, `so_regularkey` $\in S($`T[0]`$)$, that is, `key` is in H. □

LEMMA 3.4. *If* `key` *is in H at line S4, the* `find` *succeeds, and otherwise the* `find` *fails.*

PROOF. If line S4 is executed when `key` is in H, then `so_regularkey(key)` is in $S($`T[0]`$)$. `T[bucket]` is assigned to a node in that list, holding the key `so_dummykey(bucket)`. Using Eq. (1), we conclude that the searched key is in

S(`T[bucket]`), so `list_find` succeeds and so does `find`. If in line S4 `key` is not in H, it cannot be in S(T[bucket]), so `list_find` fails. □

LEMMA 3.5. *If* `key` *is in H in line D4,* `delete` *succeeds and removes* `key` *from H, and otherwise* `delete` *fails.*

PROOF. If `key` is in H, then `so_regularkey(key)` is in S(`T[0]`). `T[bucket]` is assigned to a node inside that list, where the key of that node is `so_dummykey(bucket)`. Using Eq. (1), we conclude that the searched key is in S(`T[bucket]`), so `list_delete` removes it. If `key` is not in H, it cannot be in S(`T[bucket]`), so `list_delete` fails. □

From Lemma 3.3, Lemma 3.4, and Lemma 3.5, it follows that:

THEOREM 3.6. *The split-ordered list algorithm of Figure* 5 *is a linearizable implementation of a set object.*

3.2. LOCK FREEDOM. Our algorithm uses loads and stores together with implementations of a list-based set, a shared counter, and memory allocation routines as primitive objects/operations. As we will show, in terms of these primitive operations the algorithm's implementation is wait-free, that is, each thread always completes in a finite number of operations. This implies that its overall progress condition in terms of primitive machine operations will be exactly that of the underlying implementation of those objects. Since we used the lock-free list-based sets of Harris [2001] and Michael [2002a] and a lock-free shared counter as building blocks in this presentation, our implementation will also be lock-free. As noted in the introduction, in some cases, there are advantages in using the obstruction free list-based set algorithm of Luchangco et al. [2003]. If Luchangco et al. [2003] is used together with a lock-free shared counter, our hash table will be obstruction-free [Herlihy et al. 2003].

THEOREM 3.7. *The split-ordered list algorithm of Figure* 5 *is a wait-free implementation of a set object in terms of* `load`, `store`, `fetch-and-inc`, `fetch-and-dec`, `list_find`, `list_insert` *and* `list_delete` *operations.*

PROOF. The functions `insert`, `find`, `delete` and `initialize_bucket` all take a finite number of steps, each of which is a machine level `load` or `store` operation or an operation on the list based set object or the shared counter. The `initialize_bucket` procedure is the only one with a recursive call. However, the recursion of `initialize_bucket` is limited, since each step is executed on the `parent` of a `bucket`, which satisfies `parent` $<$ `bucket`. Since `bucket` 0 is initialized from the start, the recursion is finite, and the implementation is wait-free. □

The lock-freedom property means that a thread executing the hash table operation completes in a finite number of steps unless other threads are infinitely making progress. Thus, it is a weaker requirement than wait-freedom, and by combining implementations the following is a corollary of Theorem 3.7:

COROLLARY 3.8. *The split-ordered list algorithm of Figure* 5 *with lock-free implementations of* `list_find`, `list_insert`, `list_delete`, `fetch-and-inc`, *and the* `fetch-and-dec` *operations is lock-free.*

COROLLARY 3.9. *The split-ordered list algorithm of Figure 5 with obstruction-free implementations of* `fetch-and-inc`, `fetch-and-dec`, `list_find`, `list_insert` *and* `list_delete` *operations is obstruction-free.*

The `fetch-and-inc` and `fetch-and-dec` operations have known lock-free implementations [Michael and Scott 1998].

3.3. COMPLEXITY. The most important property of a hash table is its expected constant time performance. When analyzing the complexity of hashing in a concurrent environment there are two adversaries one needs to consider: one controlling the distribution of hash values of keys by the hash function (i.e., how good is the hash), the other controlling the scheduling of thread operations. We will follow the standard practice of modelling the hash function as a uniform distribution over keys [Cormen et al. 2001]. The uniformity of keys we assume is global, that is, it extends across all threads in a given execution (A simple way to think of this is that we apply the standard uniform distribution assumption [Cormen et al. 2001] on the linearization of any given execution). We will use the term *expected time* (or *expected number of steps*) to refer to the expected number of machine instructions per operation in the worst case scheduling scenario, assuming a hash function of uniform distribution. We will use the term *average time* (or *average number of steps*) to refer to the number of machine instructions per operation averaged over all executions, also assuming a uniform hash function. It follows that constant expected time implies constant average time.

In our complexity analysis, we assume that loops within the underlying linked list code involve no more than a constant number of retries. This assumption is realistic since a nonconstant number of retry loops implies *Compare& Swap* failures caused by contention within a single bucket, which cannot occur due to the global uniformity of the hash function.

We will show that under any scheduling adversary, our algorithm performs all hash table operations in constant average time. The complexity improves to constant expected time if we assume a *constant extendibility rate*. This is a restriction on the scheduler that requires that the table is never forced to extend a nonconstant number of times while a thread is delayed by the scheduler. It means that given a good hash function, the adversary cannot cause any single operation to take more than a constant number of steps unless it delays its progress through more than a constant number of global resize operations. Formally, when there are n items in the data structure, a thread must complete a single operation before $n \cdot 2^c$ successful insertions of elements by other threads were completed, where $c \in O(1)$. We believe this is the common situation in practice.

Two algorithmic issues require a detailed proof: one is the complexity of list operations, which is essentially the complexity of executing a `list_find`, and the other is the complexity of `initialize_bucket`, which involves recursive calls.

Denote by n the total number of items in the set, and by s the number of buckets. For the complexity analysis, we are not interested in the cases where the table is small, so we make the assumption that s is greater than the number of threads. Let L denote the load factor `MAX_LOAD` in our code, typically a small constant.

LEMMA 3.10. *For any number p of threads, at all times the following condition holds:*

$$\frac{n-p}{s} \leq L.$$

PROOF. Focus on the successful completed `insert` and `delete` operations. Each successful insertion incremented `count` by 1, and each successful deletion decremented it. In any state, there are no more than p concurrent operations. Every one of the "already completed" insert operations checked, when executing line I9, that the ratio of `count` and `csize` is not more than L, and doubled the `size` if the gap was exceeded. At all times, there are no more than p currently executing `insert` operations. Therefore, when $n/s > L$ and a resize is needed, no more than p new keys can be inserted to the data structure before the resize takes place. $\square$

LEMMA 3.11. *Assuming a hash function of uniform distribution, the probability that a bucket is not accessed during the time where the table size is s, is asymptotically bounded by* $\exp(-L/2)$.

PROOF. Focus on a growing table from size $s/2$ to s and then to $2s$. According to Lemma 3.10, in the state in which line I10 doubled the table from $s/2$ to s, the number of items in the table was less or equal to $p + Ls/2$. When later in line I10 the table doubled in size to $2s$, the condition of line I9 implies that the number of items was at least Ls. The last two observations imply that during the set of states in which `size` was s, the item count increased by at least $Ls/2 - p$, that is, line I9 was executed at least $Ls/2 - p$ times. When we consider at most p processes that may have begun the insert operation when `size` was less than s, we get that line I2 was executed at least $Ls/2 - 2p$ times.

Assuming a uniform distribution of the keys, the probability that a bucket b was not accessed during this period is at most $(\frac{s-1}{s})^{Ls/2-2p}$. When p is significantly smaller than s, as assumed, the last expression is asymptotically equal to $\exp(-L/2)$. $\square$

LEMMA 3.12. *For any key k, when the table size is s and the bucket k mod $size$ is initialized, there is no dummy node with key d such that $k \bmod size \prec d \prec k$, that is, d's split-order value is between those of k mod $size$ and k.*

PROOF. Assume by way of contradiction that d is the key of a node such that: $k \bmod size \prec d \prec k$. It is the case that $d < size$ because d is in the list, and bucket indices are always smaller than the table size. Therefore, d has less than $\log_2(size)$ non-zero bits. The keys k and k *mod size* have at least $\log_2(size) - 1$ identical less significant bits. The split-order value of d is between them, so it must have the same low $log_2(size) - 1$ bits, that actually constitute all of its non-zero bits. This implies that $d = k \bmod size$ under the split-order, a contradiction to the assumption that $d \succ k \bmod size$. $\square$

LEMMA 3.13. *If the hash function distributes the keys uniformly then:*

—*In any execution history, the list traversal of* `list_find` *takes constant time on average.*

—*Under the constant extendibility rate assumption, the traversal of* `list_find` *takes expected constant time.*

PROOF. For a table of size s, the expected number of uninitialized buckets among the first $s/2$ buckets is no more than $s/2 \cdot \exp(-L/2)$, by Lemma 3.11. For each of the initialized buckets, there is a dummy node in the list holding the bucket index as the split-order value. Therefore, there are at least $s/2 \cdot (1 - \exp(-L/2))$ dummy nodes with keys from $0..s/2 - 1$. Those values divide the integer range into $s/2$ equal segments, while the missing items are distributed evenly. Using Lemma 3.10, there are on average less than

$$\frac{n}{s/2 \cdot (1 - \exp(-L/2))} \leq \frac{Ls + p}{s/2 \cdot (1 - \exp(-L/2))} = \frac{2L + 2p/s}{1 - \exp(-L/2)} \qquad (2)$$

nodes between every two dummy nodes. The operation `list_find` is called to search for a key k from the bucket *k mod size*, so, using Lemma 3.12, we conclude that in the state in which it was called there were no dummy nodes between the bucket's dummy node and the node at which the search would be completed. We have just computed that dummy nodes are distributed in intervals of less than

$$\frac{2L + 2p/s}{1 - \exp(-L/2)}$$

nodes, implying that if the table size does not change, the search will take no more than a constant expected number of steps.

We will now show that if the search took more than constant time, there were enough successful inserts to maintain a constant number of steps on average. If `list_find` took $\Omega(r)$ steps, $\Omega(r)$ dummy nodes must have been traversed, since at any time the expected distance between them is constant. All of these dummy nodes were inserted to the list after `list_find` started. The number of dummy nodes in the original bucket doubles each time the table is extended, so there were $\Omega(\log r)$ table resize events. Since there were exactly n items in the table when the `list_find` operation started, the number of items had to rise by $\Omega(rn)$, that is, $\Omega(rn)$ successful insertions to the list. There were no more than p threads that successfully executed `list_insert` but then were delayed before completing the `insert` routine. Therefore, we can consider only $\Omega(rn - p)$ as complete hash table insertions. According to the constant extensibility rate assumption, a thread must complete a single operation within $n \cdot 2^c$ successful insertions. Looking at the single operation that took $\Omega(r)$ steps, we now know that during that time there were at least $\Omega(rn - p)$ successful inserts, but we also know that the operation lasted less than $n \cdot 2^c$ successful operations. We get that $\log(r - p/n) \in O(1)$, and thus $r \in O(1)$. □

LEMMA 3.14. *Given a hash function with an expected uniform distribution, the number of steps performed by the function* `initialize_bucket` *is constant on average. Under the constant extendibility rate assumption, the number of expected steps in the worst case execution is constant.*

PROOF. A recursive call to `initialize_bucket` terminates when the parent bucket is initialized. To have m recursive calls, m uninitialized ancestor buckets are needed. Applying Lemma 3.11, this may happen with probability less than $\exp(-L(m - 1)/2)$. The number of m-deep executions among m calls to initialize_bucket is $m \cdot \exp(-L(m-1)/2) \in O(1)$, implying that the expected number of recursive calls is constant. By Lemma 3.13, the list_insert call inside initialize_bucket costs a constant number of steps on average. If we assume constant extendibility rate (threads are not delayed while the table is doubled a nonconstant number of times), a recent ancestor of every bucket is always initialized, and the recursion depth is constant. Also, according to Lemma 3.13, the execution of list_insert is of expected constant time. □

THEOREM 3.15. *Given a hash function with expected uniform distribution, all hash table operations complete within a constant number of steps on average. Assuming a constant extendibility rate, all hash table operations complete within expected constant number of steps.*

PROOF. Beside executing a constant number of simple instructions, all hash operations call a list traversing routine twice at most (actually, only hash_delete may cause list_find to run twice). By Lemma 3.13, the list traversals cost a constant average number of steps, and by Lemma 3.14, the initialize_bucket operation also completes within a constant average number of steps. Both of the above lemmas imply that under the constant extendibility rate assumption, the number of steps is constant in the worst case execution assuming a uniform distribution. □

4. *Performance*

We ran a series of tests to evaluate the performance of our lock-free algorithm. Since our algorithm is the first lock-free extensible hash table, it needs to be proven efficient in comparison to existing lock-based extensible hash table algorithms. We have thus chosen to compare our algorithm to the resizable hash table algorithm of Lea [2003] (revision 1.3), originally suggested as a part of *util.concurrent.ConcurrentHashMap*, the proposed Java™ Concurrency Package, JSR-166.

Lea's algorithm is based on an exponentially growing table of buckets, doubled when the average bucket load exceeds a given load factor. Access to the table buckets is synchronized by 64 locks, dividing the bucket range to 64 interleaved regions, that is, lock i is obtained when bucket b is accessed if $b \ mod \ 64 = i$. Insert and delete operations always acquire a lock, but find operations are first attempted without locking, and retried with locking upon failure. When a process decides to resize the table, it locks all 64 locks, allocates a larger array and rehashes the buckets' items to their new buckets, utilizing the simplicity of power-of-two hashing. This scheme offers good performance, in comparison to simpler schemes that separately lock each bucket, by significantly reducing the number of locks that need to be acquired when resizing. Figure 8 illustrates the effect of different concurrency levels on Lea's algorithm performance.

We translated the Java™ code by Lea to C++ and simplified it to handle integer keys that also serve as values, exactly as in our new algorithm's code. There is a trade-off in this algorithm: the more locks used, the lower the contention on them, but the higher the global delay when resizing. We thus ran an experiment to confirm that in the translated algorithm there is no significant advantage to using more or less than 64 locks.

We compared our split-ordered hashing algorithm to Lea's algorithm using a collection of experiments on a 30-processor Sun Enterprise 6000, a cache-coherent

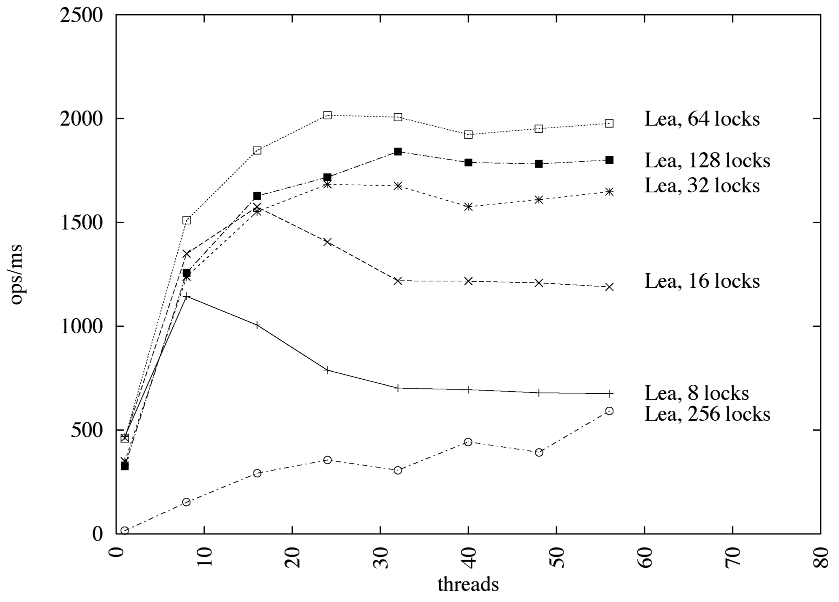


FIG. 8. Lea's algorithm with different concurrency levels.

NUMA machine formed from 15 boards of two 300 MHz UltraSPARC® II processors and 2 GB of RAM on each. The C/C++ code was compiled with a Sun *cc* compiler 5.3, with the flags `-xO5` and `-xarch=v8plusa`. We executed each experiment three times to lower the effect of temporary scheduling anomalities.

Lea's algorithm has significant vulnerability in multiprogrammed environments since whenever the resizing processor is swapped out or delayed, the algorithm as a whole grinds to a `halt`. The significant latency overhead while resizing would also make it less of a fit for real-time environments. However, our tests here are designed to compare the performance of the algorithms in the currently more common environments without multiprogramming or real-time requirements.

Since Lea's algorithm behaves differently when hash table operations fail rather than succeed, we also tested the algorithms in scenarios where they begin after a significant amount of elements have been inserted. Since the range from which the elements are selected is limited, the more we pre-insert, the more chances are that an element is already in the table when search for it. Additionally, we ran a series of experiments measuring the change in throughput as a function of concurrency under various synthetic distributions of *insert*, *delete* and *find*.

To capture performance under typical hash-table usage patterns [Lea (personal communication, 2003)], we first look at a mix that consists of about 88% *find* operations, 10% *insert*s and 2% *delete*s. Our first graph, in Figure 9, shows the results of comparing the algorithms under such a pattern. The hash table load factor (the number of items per bucket) for both tested algorithms was chosen as 3. In the presented graph we show the change in throughput as a function of concurrency. As can be seen, at high loads the lock-free split-ordered hashing algorithm significantly outperforms Lea's when the concurrency level goes beyond eight threads.

The first data point, corresponding to the throughput when executed by a single thread, is a measure for the overhead cost of the new algorithm. According to this data point, the new algorithm is 23% slower than the lock-based algorithm when

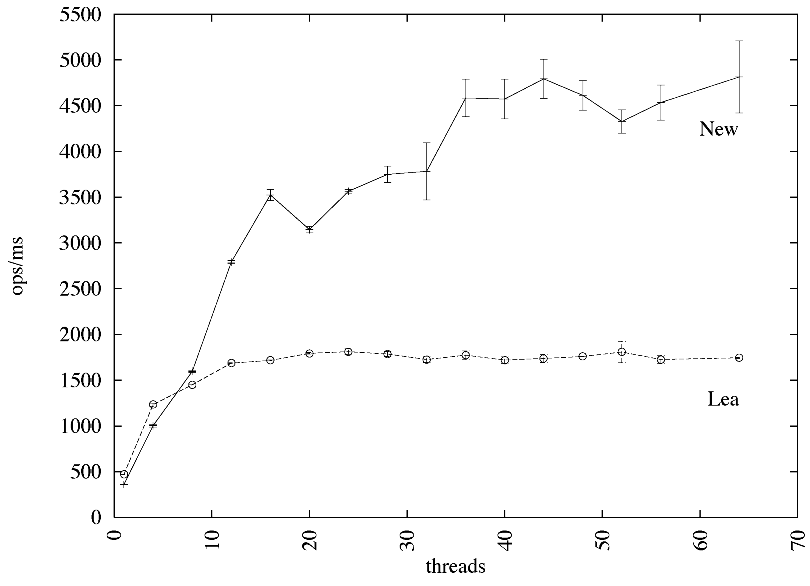


FIG. 9. Throughput of both algorithms. Standard deviation is denoted by vertical bars.

run by a single thread.

—Lea's algorithm reaches peak performance at about 24 threads and at the same concurrency level, our new algorithm has two times higher throughput.

—Our algorithm reaches peak performance at 44 threads, where it is almost three times faster than Lea's.

—Our algorithm's performance fluctuates after reaching peak performance because it involves significantly higher concurrent communication and is thus much more sensitive to the specific layout of threads on the machine and to the load on the shared crossbar.

—Lea's algorithm suffers a much milder deterioration caused by the architectural critical paths because it never reaches high concurrency levels and its overall performance is limited by the bottlenecks introduced by the shared locks.

Figure 10 shows the results of an experiment varying the chosen distribution of `inserts`, `deletes`, and `finds`. Note that our algorithm consistently outperforms Lea's algorithm throughout the full range of tested distributions. We also ran an experiment that varies the load factor in our algorithm. As seen in Figure 11, the load factor does not affect the performance significantly, and its effect is in any case minimal when compared to those of the thread layout and the overall communication overhead.

Figure 12 shows the throughput of both algorithms when the amount of pre-insertions varied among 0, 300 K, 600 K, and 900 K. The range from which elements were selected was $[0, 1e+6]$, so pre-insertions affected significantly the success rate of the hash table operations. The performance of Lea's algorithm slightly improves on lower concurrency levels, but from 12 threads and on the new algorithm is faster.

We also tested the robustness of the algorithms under a biased hash function, mimicking conditions in case of a bad choice of a hash function relative to the

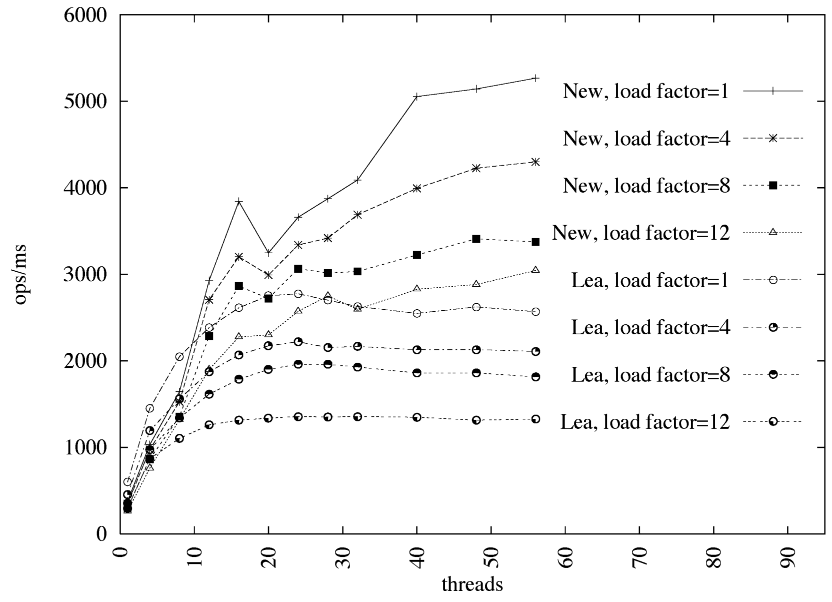


FIG. 10. Varying operation distribution.

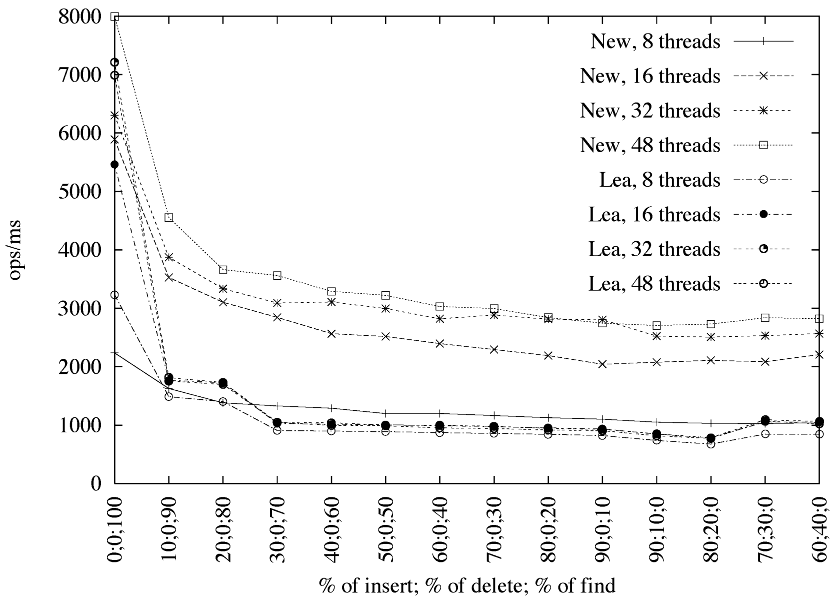


FIG. 11. Varying load factor.

given data. To do, so we generated keys in a nonuniform distribution by randomly turning off 0 to 3 LSBs of randomly chosen integers. Our empirical data shows that our algorithm shows greater robustness: it was slowed down by approximately 7%, while Lea's algorithm's performance decreased by more than 30%. The reason for this is that a biased hash function causes some number of buckets to have many more items than the average load. The locks controlling these buckets in Lea's

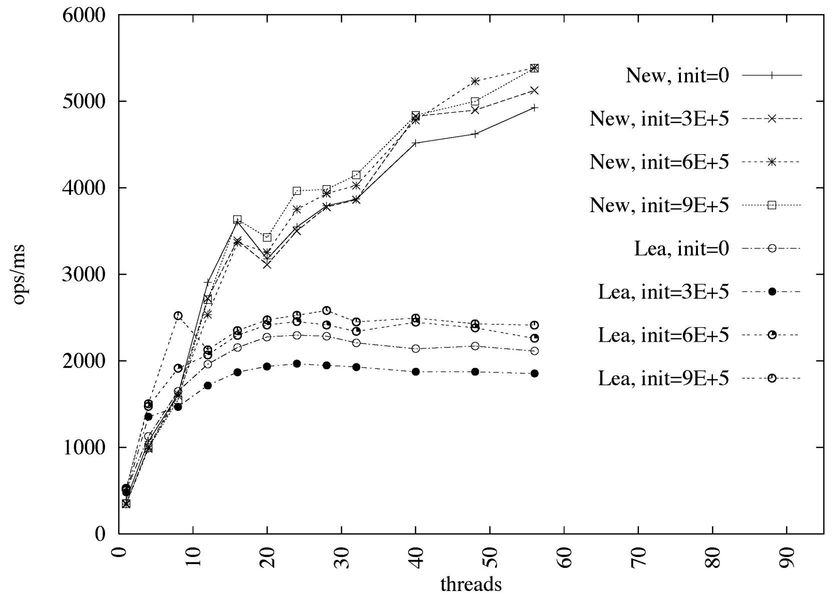


FIG. 12. Varying amount of pre-insertions.

algorithm are thus contended, causing a performance degradation. This does not happen in the lock-free list used by the new algorithm.

Based on the above results, we conclude that in low-load nonmultiprogrammed environments both algorithms offer comparable performance, while under medium to high loads, split-ordered hashing scales better than Lea's algorithm and is thus the algorithm of choice.

5. *Conclusion*

Our article introduced split-ordered lists and showed how to use them to build resizable concurrent hash tables. We believe the split-order list structure may have broader applications, and in particular it might be interesting to test empirically if a purely sequential variation of split-ordered hashing will offer an improvement over linear hashing in the sequential case. This follows since splitting buckets in split-ordered hash tables does not require redistribution of individual items among buckets, but rather only the insertion of a dummy node, and in the sequential case the need for the dummy nodes might be avoidable altogether.

Appendix

A. *Additional Code*

For the purpose of being self contained, we provide in Figures 13 and 14 the code for the lock-free CAS-based ordered list algorithm of Michael [2002a].

The difficulty in implementing a lock-free ordered linked list is in ensuring that during an insertion or deletion, the adjacent nodes are still valid, that is, they are still in the list and are still adjacent. Both the implementation of Harris [2001] and that of Michael [2002a] do so by "stealing" one bit from the pointer to mark a node as deleted, and performing the deletion in two steps: first marking the node,

```
struct MarkPtrType {
  <mark, next>: <bool, NodeType *>
};

struct NodeType {
  key_t key;
  MarkPtrType <mark, next>;
};

/* thread-private variables */
MarkPtrType *prev;
MarkPtrType <pmark, cur>;
MarkPtrType <cmark, next>;

int list_insert(MarkPtrType *head,
                NodeType *node) {
  key = node->key;
  while (1)  {
    if (list_find(head, key) return 0;
    node-><mark,next> = <0,cur>;
    if (CAS(prev, <0,cur>, <0,node>))
      return 1;
  }
}

int list_delete(MarkPtrType *head,
                so_key_t key) {
  while (1) {
    if (!list_find(head, key))
      return 0;
    if (!CAS(&(cur-><mark,next>), <0,next>,
        <1,next>))
      continue;
    if (CAS(prev, <0,cur>, <0,next>))
      delete_node(cur);
    else list_find(head, key);
    return 1;
  }
}
```

FIG. 13. Michael's lock free list based sets.

and then deleting it. This bit and the *next* pointer are set atomically by the same CAS operation.[5] The `list_find` operation is the most complicated: it traverses through the list, and stops when it reaches an item that is equal-to or greater-than the searched item. If a marked-for-deletion node is encountered, the deletion is completed and the traversal continues. The `list_find` in Michael's scheme thus improves on that of Harris since by completing the deletion immediately when a marked node is encountered it prevents other operations from traversing over marked nodes, that is, ones that have been logically deleted.

[5] Stealing one bit in a pointer in such a manner is straightforward assuming properly aligned memory, and can be achieved with indirection using a "dummy bit node" [Agesen et al. 2000] in languages like Java^{TM} where stealing a bit in a pointer is a problem. The new Java^{TM} Concurrency Package proposes to eliminate this drawback by offering "tagged" atomic variables.

```
int list_find(NodeType **head, so_key_t key) {
try_again:
  prev = head;
  <pmark,cur> = *prev;
  while (1) {
    if (cur == NULL) return 0;
    <cmark,next> = cur-><mark,next>;
    ckey = cur->key;
    if (*prev != <0,cur>)
      goto try_again;
    if (!cmark) {
      if (ckey >= key)
        return ckey == key;
      prev = &(cur-><mark,next>);
    }
    else {
      if (CAS(prev, <0,cur>, <0,next>))
        delete_node(cur);
      else goto try_again;
    }
    <pmark,cur> = <cmark,next>;
  }
}
```

FIG. 14. Michael's lock free list based sets–continued.

```
int fetch-and-inc(int *p) {
  do {
    old = *p;
  } while (!CAS(p, old, old+1);
  return old;
}

int fetch-and-dec(int *p) {
  do {
    old = *p;
  } while (!CAS(p, old, old-1);
  return old;
}
```

FIG. 15. Lock free atomic counter implementation.

Figure 15 depicts a simple lock-free implementation of a shared incrementable (or decrementable) counter using CAS.

ACKNOWLEDGMENTS. We thank Mark Moir, Victor Luchangco and Paul Martin for their help and patience in accessing and running our tests on several of Sun's large multiprocessor machines. This paper could not have been completed without them. We also thank Victor Luchangco, Mark Moir, Maged Michael, Sivan Toledo, and the anonymous PODC 2003 referees for their helpful comments and insights. We thank Doug Lea for his constructive skepticism and for sharing with us real-world data on the growth characteristics of dynamic hash tables. Finally, the comments of the anonymous referees assisted greatly in improving this manuscript.

REFERENCES

AGESEN, O., DETLEFS, D., FLOOD, C., GARTHWAITE, A., MARTIN, P., SHAVIT, N., AND STEELE, G. 2000. DCAS-based concurrent deques. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York.

BUTTAZZO, G., LIPARI, G., ABENI, L., AND CACCAMO, M. 2005. *Soft Real-Time Systems: Predictability vs. Efficiency*. Series: Series in Computer Science. Springer-Verlag, New York.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, MA.

ELLIS, C. S. 1983. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM, New York, 106–116.

ELLIS, C. S. 1987. Concurrency in linear hashing. *ACM Trans. Database Syst. 12,* 2, 195–217.

GAO, H., GROOTE, J., AND HESSELINK, W. 2004. Almost wait-free resizable hashtables. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPOPS).*

GREENWALD, M. 1999. Non-blocking synchronization and system design. Ph.D. dissertation. Stanford University Tech. Rep. STAN-CS-TR-99-1624, Palo Alto, CA.

GREENWALD, M. 2002. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*. ACM, New York, 260–269.

HARRIS, T. L. 2001. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of 15th International Symposium on Distributed Computing (DISC 2001)*. 300–314.

HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 2002. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing (DISC 2002)*. 339–353.

HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*. ACM, New York, 92–101.

HERLIHY, M. P., AND WING, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12,* 3, 463–492.

HESSELINK, W., GROOTE, J., MAUW, S., AND VERMEULEN, R. 2001. An algorithm for the asynchronous write-all problem based on process collision. *Distrib. Comput. 14,* 2, 75–81.

HSU, M., AND YANG, W. 1986. Concurrent operations in extendible hashing. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB'86)* (Kyoto, Japan, Aug. 25–28). W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds. Morgan-Kaufmann, San Francisco, CA, 241–247.

KANELLAKIS, P. C., AND SHVARTSMAN, A. 1997. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers.

LEA, D. 2003. Hash table util.concurrent.ConcurrentHashMap, revision 1.3, in JSR-166, the proposed Java Concurrency Package. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/.

LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB'80)* (Montreal, Que., Canada, Oct. 1–3). IEEE Computer Society, Press, Los Alamitos, CA, 212–223.

LUCHANGCO, V., MOIR, M., AND SHAVIT, N. 2003. Nonblocking k-compare single swap. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York.

MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. 1991. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 106–113.

MICHAEL, M. M. 2002a. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York, 73–82.

MICHAEL, M. M. 2002b. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*. ACM, New York, 21–30.

MICHAEL, M. M., AND SCOTT, M. L. 1998. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared-memory multiprocessors. *J. Parall. Distrib. Comput. 51*, 1, 1–26.

MOIR, M. 1997. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*. ACM, New York.

VALOIS, J. D. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the Symposium on Principles of Distributed Computing*. ACM, New York, 214–222.

RECEIVED MARCH 2004; REVISED SEPTEMBER 2005; ACCEPTED FEBRUARY 2006