Split-Ordered Lists: Lock-Free Extensible Hash Tables *

Ori Shalev
Tel-Aviv University
ori_shalev@yahoo.com

Nir Shavit
Tel-Aviv University
shanir@cs.tau.ac.il

ABSTRACT

We present the first lock-free implementation of an extensible hash table running on current architectures. It provides concurrent insert, delete, and search 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. Empirical tests conducted on a large shared memory multiprocessor show that even in non-multiprogrammed environments, the new algorithm significantly outperforms the most efficient known lock-based algorithm at all concurrency levels, exhibiting up to four times higher throughput at peak load. The incremental nature of our algorithm makes it well suited for real-time applications, as it offers predictable performance without unexpected breaks for resizing.

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**]: Data Storage Representation—*Hash-table representations*

General Terms

Algorithms, Theory, Performance, Experimentation.

Keywords

Concurrent Data Structures, Hash Table, Non-blocking Synchronization, Compare-and-Swap, Real-Time.

*This work was supported by a Sun Microsystems Collaborative Research Grant.

PODC'03, July 13–16, 2003, Boston, Massachusetts, USA.

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 search operations [1]. 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. In this paper, "resizing" means extending the table. It has been shown elsewhere [10] that as a practical matter, hash tables need only increase in size.

This paper presents the first lock-free extensible hash table that works on current architectures, that is, uses only loads, stores and CAS (or LL/SC [19]) operations. In a manner similar to sequential linear hashing [13] and fitting real-time applications, resizing costs are split incrementally to achieve expected $O(1)$ operations per insert, delete and search. It 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 preliminary empirical testing shows that in a concurrent environment, even without multiprogramming, our lock-free algorithm significantly outperforms the most efficient known resizable hash-table algorithm due to Lea [11].

1.1 Background

There are several lock-based concurrent hash table implementations in the literature. Michael [16] has shown that simple algorithms using a reader-writer lock [15] per bucket have reasonable performance without resizing. However, to resize one would have to hold the locks on all buckets simultaneously, leading to major overheads. The algorithm by Lea [11], currently proposed for *java.util.concurrent*, the Java™ Concurrency Package, is probably the most efficient known 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 priority inversions [3]. 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 new added buckets. Designing a lock-free

resizable hash table is thus a matter of both practical and theoretical interest.

Michael, in [16], builds on the work of Harris [5] 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 multi-programmed environments. However, this structure is not resizable: if the number of elements grows beyond the pre-determined size, the time complexity of operations will no longer be constant.

As part of his "two-handed emulation" approach, Greenwald [4] 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, Greenwald's hash table is resizable, but is not a true resizable hash table since the expected number of steps per operation is not constant: it involves an elaborate "checking" scheme in which every process independently traverses a linear number of buckets to guarantee the lock-free progress property.

Independently of our work, Gao et. al [2] have developed a resizable and "almost wait-free" hash table 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. Migration is performed using a write-all algorithm [9], based on which insertion appears to require slightly more than a constant expected number of steps per item. The practical value of this algorithm will depend on the yet-to-be-proven performance of available algorithms for the write-all problem.

The algorithm presented here provides lock-free extensible hashing with expected constant cost per operation on current architectures. In fact, in a manner similar to sequential linear hashing algorithms [13], and in contrast to all the above algorithms [2, 4, 11], resizing is done incrementally and there is no amortization of the cost of a global rehash over all insertions: only bad distributions (that have very low probability given a uniform hash function) can cause the cost of an operation to exceed constant time. This makes the algorithm well suited for real-time applications, offering predictable constant time performance, without unexpected breaks.

1.2 The Lock-free Resizing Problem

What is it that makes lock-free resizable 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.

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 [11]). 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[1]. In Figure 1 the split-order key values are written above the nodes. 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 (non-dashed) 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 *search* for) 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). 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 sketch of the proof appears in Section 3.

[1] As detailed in the next section, some additional bit-wise modifications must be made to make things work properly.

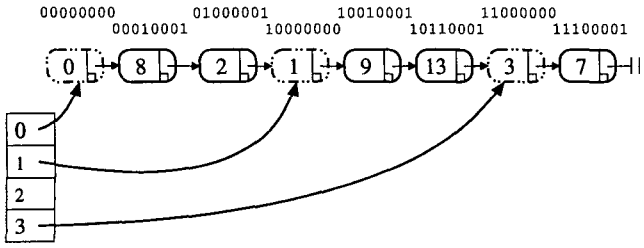


Figure 1: A Split-Ordered Hash Table

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 [5] or Michael [16], or the obstruction-free algorithms of Valois[2][20] or Luchangco et. al [14]. We chose to base our presentation on the algorithm of Michael [16], an extension of the Harris algorithm [5] that fits well with memory management schemes [6, 17] and performs well in practice.

1.4 Performance

We tested our new *split-ordered list* hash algorithm versus the most-efficient known lock-based implementation due to Lea [11]. 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. Though lock-free algorithms are expected to benefit systems especially in multiprogrammed environments, our experiments, presented in Section 4, show that split-ordered lists significantly outperform Lea's algorithm even in non-multiprogrammed test cases, exhibiting up to four times higher throughput at peak load. They also exhibit greater robustness, for example, in experiments where the hash function is biased to create non-uniform distributions.

2. THE ALGORITHM IN DETAIL

Our hash table data structure consists of two interconnected sub-structures (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 costs 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, i.e. 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 resizable Linear Hashing scheme proposed by Litwin [13], and was recently used by Lea in his concurrent resizable hashing scheme [11]. The novelty here is that we use it as a basis for a combinatorial structure that allows us to logically "split" items among 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. 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 non-trivial 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 us 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

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

```
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)
}
```

Figure 2: The Split-Ordering Transformation

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 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[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.

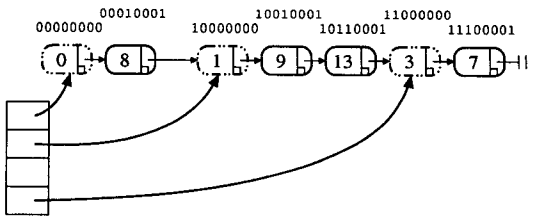
2.2 The Continuously Growing Table

We can now complete the presentation of our algorithm. We use the lock-free ordered linked-list algorithm of Michael [16] 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 [5]. Our presentation will not discuss the various memory reclamation options of such linked-list schemes, and we refer the interested reader to [5, 6, 16, 17]. To keep our presentation self contained, we provide in Appendix B 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.

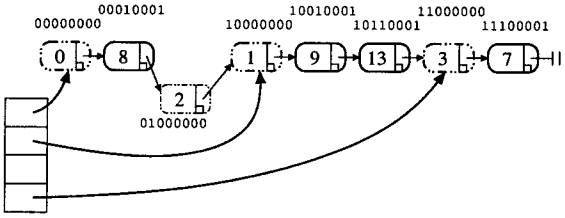
Our algorithm decides to double table size based on the average bucket load. This load is determined by maintaining a shared counter that tracks the number of items in the table. The final detail we need to deal with is how the array of buckets is repeatedly extended. To simplify the presentation, we keep the table 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 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 code for this dynamic allocation scheme appears in the appendix in Section A.

Finally, the reader may have noticed that based on the

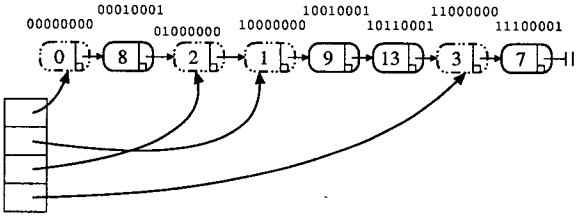
[3] An efficient implementation of the REVERSE function utilizes a 2^8 or 2^{16} lookup table holding the bit-reversed values of $[0..2^8 - 1]$ or $[0..2^{16} - 1]$ respectively.



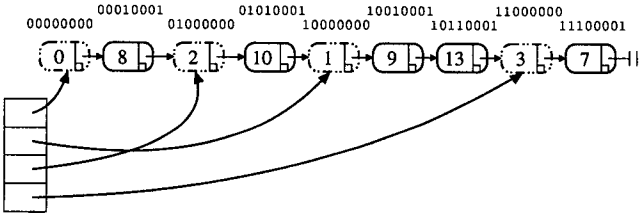
(a) Buckets 0,1 and 3 are initialized. Bucket 2 is uninitialized



(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

Figure 3: Insertion into the split-ordered list

approach described above, when an uninitialized bucket is accessed in a table of size *size*, one might need to recursively initialize (i.e. split) all $O(\log size)$ of its parent buckets to allow insertion of a new item. Though the total complexity in such a case is logarithmic, not constant, our algorithm still works. This is because given a uniform distribution of items, the chances of the above scenario happening are low, and in fact, the expected length of such a bad sequence of parent initializations is constant.

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. 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`. Three private variables serve each one of the running threads: `prev`,

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

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

MarkPtrType* T[ ]; // buckets
unsigned int count; // total item count
unsigned int size; // current table size

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

Figure 4: Types and Structures

cur and next. Those variables have the same functionality as in Michael's algorithm [16], as they are set by list_find to point at the nodes around the searched key, and subsequently used by the same thread inside other functions. In Figure 5 we show the implementation of the insert, search and delete operations. The fetch-and-inc operation is implemented in a lock-free manner via a simple repeated loop of 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. The bucket 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 (fetch-and-inc can be implemented in a lock-free manner [18]). 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 search 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. 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 our the algorithm's choice of parent uniquely (the split-order is denoted by $\prec$):

$$\forall k \neq \text{parent}, k \prec \text{bucket} \Rightarrow \text{parent} \succ k$$
$$\text{parent} \prec \text{bucket}$$
$$\text{parent} < \text{bucket}$$

This value is achieved by unsetting bucket's most significant turned-on bit. If the exact dummy key already exists in the list, it may 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.

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.

```
int insert(KeyType 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 search(KeyType 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(KeyType 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:  CAS(&(T[bucket]), UNINITIALIZED, dummy);
}
```

Figure 5: Our split-order based hashing algorithm

3. CORRECTNESS PROOF

This section contains the skeleton of a formal proof that our algorithm has the desired properties of a resizable hash table. Our model of multiprocessor computation follows [8], 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.

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 [1], including the three functions *insert, delete*, and *search*. The *insert* operation returns 1 if the key was successfully inserted to the set, and 0 if that key already existed in the table. The *search* 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 performed 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 [5, 16]. 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 [5, 16]. 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.*

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

Definition 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 `search` 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.

LEMMA 1. *If* `key` *is in H in line I5, then* `insert` *fails and if it is not,* `insert` *succeeds and* `key` *joins H.*

LEMMA 2. *If* `key` *is in H at line S4, the* `search` *succeeds, and otherwise the* `search` *fails.*

LEMMA 3. *If* `key` *is in H in line D4,* `delete` *succeeds and removes* `key` *from H, and otherwise* `delete` *fails.*

From Lemma 1, Lemma 2, and Lemma 3 it follows that:

THEOREM 1. *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 and a shared counter 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 underlaying implementation of these objects. Since in this presentation we used use as building blocks the lock-free list-based sets of [5, 16] and a lock-free shared counter, 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 [14]. If [14] is used together with a lock-free shared counter, our hash table will be obstruction free [7].

THEOREM 2. *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`, `search`, `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 2:

COROLLARY 1. *The split-ordered list algorithm of Figure 5 with lock-free implementations of* `list_find`, `list_insert`, `list_delete`, `fetch-and-inc`, `fetch-and-dec` *operations is lock-free.*

The `fetch-and-inc`/`fetch-and-dec` operations have known lock-free implementations [18].

3.3 Complexity

The most important property of a hash table is its expected constant time performance. As we will see, given a uniform hash function, the expected time for any operation is constant. Two 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 n as the total number of items in the table, s as the number of buckets. Let L denote the load factor `MAX_LOAD` in our code, typically a small constant.

LEMMA 4. *For any number p of threads, at all times the following condition holds:*

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

LEMMA 5. *Assuming a hash function of uniform distribution, the probability that a bucket is not accessed in states where the table size is s, is bounded by* $e^{-L/2}$.

LEMMA 6. *For any key* k, *when the table size is* s *and the bucket* k mod size *is initialized, there is no dummy node after the node holding the split-order representation of key* k mod size *and before any node whose key is greater or equal to* k, *under the split-order.*

LEMMA 7. *In any execution history, if the hash function distributes the keys uniformly, the expected number of nodes over which* `list_find` *traverses while searching for a key is constant.*

PROOF. For a table of size s, the expected number of uninitialized buckets among the first $s/2$ buckets is no more than $s/2 \cdot e^{-L/2}$, by Lemma 5. 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 $s/2 \cdot (1 - e^{-L/2})$ dummy nodes with keys from $0..s/2 - 1$. Those values divide the integer range to $s/2$ equal segments, while the missing items are distributed evenly. There are on average less than

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

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 6 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 each $\frac{2L+2p/s}{1-e^{-L/2}}$ nodes, implying that the search will take no more than a constant number of steps. □

LEMMA 8. *Given a hash function with expected uniform distribution, the expected number of steps of the function* `initialize_bucket` *is O(1).*

PROOF. Recursive calls to `initialize_bucket` terminate when the parent bucket is initialized. To have m recursive calls, m uninitialized ancestor buckets are needed. Applying Lemma 5, this may happen with probability less than $e^{-L(m-1)/2}$, making the expected number of recursive calls constant. By Lemma 7, the `list_insert` call inside `initialize_bucket` costs a constant number of steps on average. □

THEOREM 3. *Given a hash function with expected uniform distribution, all hash table operations complete within expected constant number of steps.*

PROOF. All hash operations call a list traversing routine twice at most (actually only `hash_delete` may cause `list_find` to run twice). By Lemma 7 the list traversals are of constant number of steps, and by Lemma 8 the `initialize_bucket` operation also completes within a constant number of steps. □

We note that in our algorithm, the only source of expected behavior is the distribution of elements, in the sense that, each of our operations takes exactly (not amortized, as in [4, 2, 11]) constant time, and only a bad distribution can hamper this behavior. We thus have the following corollary:

COROLLARY 2. *Given a hash function with a perfectly uniform distribution, the worst-case number of steps is exactly constant.*

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 resizable hash table, it needs to be proven efficient in comparison to existing lock-based resizable hash algorithms. We have thus chosen to compare our algorithm to the resizable hash table algorithm of Lea [11], *util.concurrent.ConcurrentHashMap*, suggested as a part of 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 32 locks, dividing the bucket range to 32 interleaved regions, i.e. lock i is obtained when bucket b is accessed if $b \ mod \ 32 = i$. `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 32 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.

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 in this algorithm a trade-off, 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 32 locks as originally chosen by Lea.

We compared our split-ordered hashing algorithm to Lea's algorithm using a collection of experiments on a 72 node

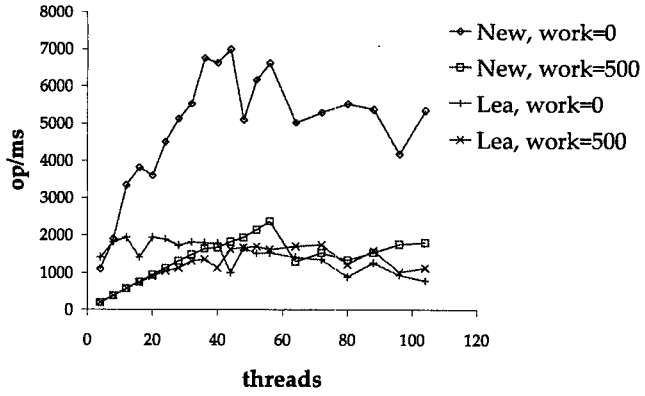


Figure 6: Throughput with `work` = 0 and `work` = 500

Sun Fire™ 15K, which is a cache-coherent NUMA machine formed from 18 boards of four 900MHz UltraSPARC® processors, connected by 18 × 18 crossbars. The C/C++ code was compiled by Sun *cc* compiler 5.3, with the flags `-xO5` and `-xarch=v8plusa`.

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 more common non-real-time environments with no multiprogramming.

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*. We also varied the workload work in various tests by choosing a delay uniformly at random from [0...work].

To capture performance under typical hash-table usage patterns [12] we first look at a mix that consists of about 88% *find* operations, 10% *inserts* and 2% *deletes*. Our first graph, in Figure 6 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 algorithm at all concurrency levels.

- When `work` = 0, Lea's algorithm reaches peak performance at about 20 threads and at the same concurrency level, our new algorithm has two times higher throughput. When `work` = 500, Lea's algorithm performs slightly worse. Both algorithms exhibit an almost linear speedup up to 60 threads.

- Our algorithm reaches peak performance at 44 threads when `work` = 0, where it is almost four times faster than Lea's. When `work` = 500, it peaks at 56 threads, having 30% higher throughput.

- 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. It reaches peak performance at 44 threads when `work` = 0 and scales better to a peak at 56 threads with `work` = 500, that is, when threads perform work between hash table accesses.

- In both cases the performance of our algorithm deteriorates after the peak because of the cost of concurrent communication over the shared crossbar. This contrasts with Lea's algorithm which suffers a much milder deterioration because it never reaches high concurrency levels and its overall performance is limited by the bottlenecks introduced by the shared locks.

- We also measured the performance of both algorithms under low load, when `work` = 3000. Under these conditions Lea's algorithm performed just slightly worse than ours, probably because of our more efficient manipulation of list items and not because of any synchronization related factor.

Figure 7 shows the results of an experiment varying the chosen distribution of `inserts`, `deletes`, and `finds`, where `work` = 0. 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 8 for the case `work` = 0, 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.

Additionally, we 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 given data. To do so we generated keys in a non-uniform 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 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 even in non-multiprogrammed environments, under medium to high loads, split-ordered hashing scales better than Lea's algorithm and is thus the algorithm of choice. In low-load non-multiprogrammied environments both algorithms offer comparable performance.

5. CONCLUSION

Our paper introduces split-ordered lists and shows how to use them to build resizable concurrent hash tables. We believe the split-order list structure may have broader applications. We are currently completing the design of a lock-free priority queue implementation based on split-ordering. It might also be interesting to test empirically if a 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

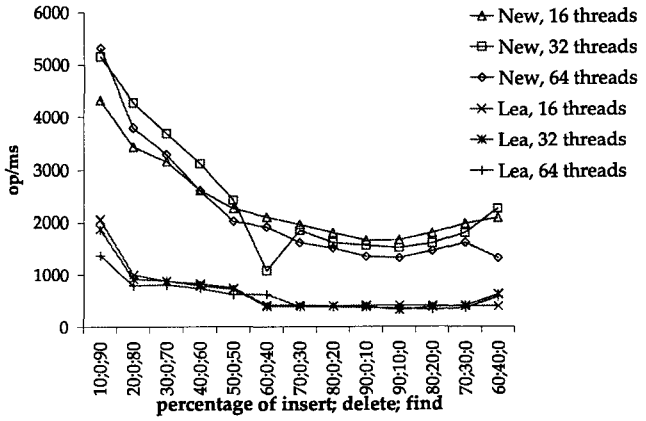


Figure 7: Varying operation distribution

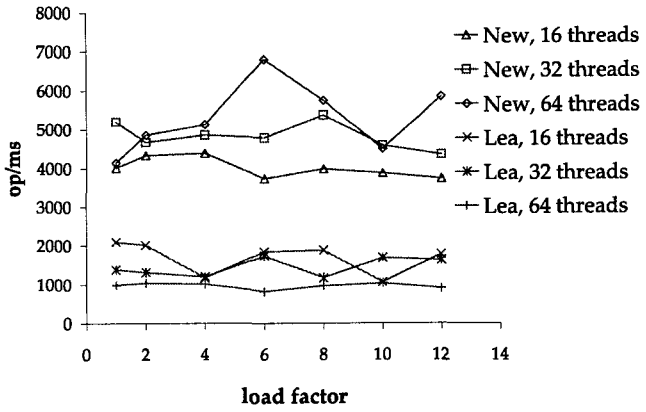


Figure 8: Varying load factor

rather only the insertion of a dummy node, and in the sequential case the need for the dummy nodes might be avoidable altogether.

6. 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, Jan Friso Groote, Maged Michael, Sivan Toledo, and the anonymous PODC 2003 referees for their helpful comments and insights.

7. REFERENCES

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

[2] GAO, H., GROOTE, J., AND HESSELINK, W. Efficient almost wait-free parallel accessible dynamic hashtables, March 2003. Unpublished manuscript.

[3] GREENWALD, M. *Non-Blocking Synchronization and System Design.* PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 8 1999.

[4] GREENWALD, M. 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* (July 2002), pp. 260–269.

[5] HARRIS, T. L. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science 2180* (2001), 300–314.

[6] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. 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)* (October 2002), pp. 339–353.

[7] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. Sotware transactional memory for dynamic-sized data structures. In *Proc. 22th Annual ACM Symposium on Principles of Distributed Computing* (July 2003).

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

[9] HESSELINK, W., GROOTE, J., MAUW, S., AND VERMEULEN, R. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing 14*, 2 (2001), 75–81.

[10] HSU, M., AND YANG, W. Concurrent operations in extendible hashing. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings* (1986), W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds., Morgan Kaufmann, pp. 241–247.

[11] LEA, D. Hash table `util.concurrent.concurrenthashmap` in JSR-166, the proposed Java Concurrency Package. `http://gee.cs.oswego.edu/dl/`.

[12] LEA, D. Personal communication, Jan. 2003.

[13] LITWIN, W. Linear hashing: A new tool for file and table addressing. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings* (1980), IEEE Computer Society, pp. 212–223.

[14] LUCHANGCO, V., MOIR, M., AND SHAVIT, N. Nonblocking k-compare single swap. In *Proc. 15th Annual ACM Symposium on Parallel Algorithms and Architectures* (June 2003).

[15] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the third ACM SIGPLAN symposium on Principles & practice of parallel programming* (1991), ACM Press, pp. 106–113.

[16] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), ACM Press, pp. 73–82.

[17] MICHAEL, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (2002), ACM Press, pp. 21–30.

[18] MICHAEL, M. M., AND SCOTT, M. L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing 51*, 1 (1998), 1–26.

[19] MOIR, M. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing* (Aug. 1997).

[20] VALOIS, J. D. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing* (1995), pp. 214–222.

APPENDIX

A. 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 introduce 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.

Applying this variation is done by defining T as an array of bucket segments, and accessing the table by calls to get_bucket and set_bucket as defined in Figure 9.

```
typedef MarkPtrType[SEGMENT_SIZE] segment_t;
segment_t T[ ];

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

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

Figure 9: Dynamic Sized Array

B. ADDITIONAL CODE

For the purpose of being self contained, this appendix provides the code for the lock-free CAS based ordered list algorithm of Michael [16].

```
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;
  }
}

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>;
  }
}
```

Figure 10: Michael's lock free list based sets