Dynamic-Sized Nonblocking Hash Tables*

Yujie Liu
Lehigh University
lyj@lehigh.edu

Kunlong Zhang
Tianjin University
zhangkl@tju.edu.cn

Michael Spear
Lehigh University
spear@cse.lehigh.edu

ABSTRACT

This paper presents nonblocking hash table algorithms that support resizing in both directions: shrinking and growing. The heart of the table is a freezable set abstraction, which greatly simplifies the task of moving elements among buckets during a resize. Furthermore, the freezable set abstraction makes possible the use of highly optimized implementations of individual buckets, including implementations in which a single flat array is used for each bucket, which improves cache locality.

We present lock-free and wait-free variants of our hash table, to include fast adaptive wait-free variants based on the Fastpath/Slowpath methodology. In evaluation on SPARC and x86 architectures, we find that performance of our lock-free implementation is consistently better than the current state-of-the-art split-ordered list, and that performance for the adaptive wait-free algorithm is compelling across microbenchmark configurations.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; E.2 [**Data Storage Representations**]: Hash-table representations

Keywords

Hash Table, Concurrent Data Structures, Nonblocking

1. INTRODUCTION

Hash tables are often chosen as the data structure to implement set and map objects, because they offer constant time insert, remove and lookup operations. Typically, a hash table consists of a static *bucket array*, where each bucket is a pointer to a dynamic set object, and a *hash function* that directs operations to buckets according to the values of the operations' operands. To preserve constant time complexity when the number of elements grows, a *resize* operation (or rehash) must be performed on the hash table to extend the size of the bucket array. However, resizing a hash table in the presence of concurrent operations in a nonblocking manner is a difficult problem. For example, Shalev and Shavit [17] state:

> *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.

In their paper, Shalev and Shavit proposed the split-ordered list [17], which circumvents explicit migration of keys between buckets. However, their algorithm has several limitations. A "shrinking" feature is missing in the resizing mechanism: the bucket array can only extend when the size of the set grows, during which "marker" nodes are permanently inserted into the underlying linked list; it is unclear how these marker nodes can be reclaimed when the set shrinks. Furthermore, the implementation leverages the assumption that memory size is bounded and known, and relies on a tree-based indexing structure with predetermined configurations.

We introduce new resizable hash table implementations that eliminate the above limitations, by solving the resizing problem with a direct and more efficient approach. In contrast to the split-ordered list, our implementations achieve three new properties. First, they are **dynamic**: the bucket array can adjust its size both upward *and downward*, according to the size of the set. Second, the bucket array is **unbounded** and we make no assumption about the size of memory. Third, our algorithm admits **wait-free** variants, where every insert, remove, and lookup operation completes in a bounded number of steps, even in the presence of resizing.

The major technical novelty of our implementations stems from the definition and use of freezable set objects. In addition to canonical set operations (i.e., insert, lookup, and remove), a freezable set

*This work was supported in part by the National Science Foundation under grants CNS-1016828, CCF-1218530, and CAREER-1253362.

PODC'14, July 15–18, 2014, Paris, France.

http://dx.doi.org/10.1145/2611462.2611495.

provides a "freeze" operation that makes the object immutable. In our algorithms, each bucket is implemented using a freezable set. To resize a hash table, buckets in the old bucket array are frozen before their key values are copied to the new table. The migration of keys during resizing is incrementally performed in a *lazy* manner, and more importantly, the logical state of the set is never changed by migration. This ensures that every insert, remove, and lookup operation is linearizable [9].

Practical lock-free and wait-free implementations of freezable sets can be derived from a recent unordered list algorithm [20]. In this paper, we introduce two new implementations, both of which are specialized and streamlined for use in our hash table algorithms to achieve better performance. Of particular interest is the fact that bounds on the size of freezable sets allow an array-based implementation, which increases locality and decreases space overhead relative to linked lists.

In experimental evaluation, we show that our hash table achieves both high scalability and low latency. In particular, our lock-free implementation significantly outperforms the state-of-the-art split ordered list. Although our implementation does not lower the asymptotic time complexity over prior work, it gains a performance benefit by reducing the number of memory indirections, which in turn translates to improved cache utilization on modern processors. The performance benefit, coupled with the dynamic feature and the possibility of wait-freedom, makes our algorithms ideal candidates for use in applications requiring progress and performance.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. We briefly introduce freezable set objects in Section 3, and discuss the implementations in Sections 6 and 7. The lock-free and wait-free hash set algorithms are discussed in Section 4 and Section 5. We evaluate performance in Section 8, and conclude in Section 9.

2. RELATED WORK

While there are many types of sequential hash tables, only a few concurrent hash tables exist. Hash tables using fine-grained locks have been known for decades [1, 2, 10, 12], but continue to see new innovations [7, 8, 18]. More recently, several nontrivial nonblocking implementations [3–5, 15–17, 19] have been discovered.

The first practical nonblocking hash table was designed by Michael [15], which uses a fixed-size bucket array of lock-free linked lists. The lists are a streamlined version of the lock-free ordered list by Harris [6]. Independently, Greenwald [5] implemented a lock-free closed addressing hash table. Greenwald's hash table is resizable, but relies on a DCAS (double-compare-and-swap) operation. Unfortunately, simulating DCAS in a lock-free manner is expensive [14], requiring at least 5 CAS operations, and implementing it via hardware transactional memory can only achieve obstruction-freedom.

Shalev and Shavit [17] presented a lock-free extendible hash table using the recursive split-ordering technique. Their hash table consists of two substructures: an ordered linked list based on the work of Michael [15], and a static tree-based directory structure. The ordered list contains both data and marker nodes, where marker nodes roughly partition the list into constant-size contiguous sublists. To find an element (or its predecessor), threads first perform a constant-time traversal of the directory to locate the closest preceding "marker" node, and then inspect the sub-list that follows. A clever bit-reversal technique used on the hash value of an element ensures that as buckets are split, and new marker nodes added, the order of elements within the list need not change. Thus while resizing may require a large number of updates to the directory, the relative position of elements in the list does not change. Zhang and Larson [19] announced that they had implemented a lock-free linear hash table, also using the recursive split-ordering technique.

Gao, Groote, and Hesselink [4] proposed a resizable, lock-free, open addressing hash table. They maintain a second table during resizing; to migrate a key, they first mark the key as being moved, then copy it to the second table, and finally update the original key's mark to indicate that it has moved. Whenever an operation finds a marked key, it must help finish resizing the entire table, and then resume its execution on the second table. Purcell and Harris [16] proposed another lock-free open addressing hash table that is not resizable, but is space-efficient. In particular, their hash table can reuse the space occupied by deleted keys.

Feldman, LaBorde, and Dechev [3] demonstrated that with perfect hashing, it is possible to implement a wait-free hash table. Their implementation makes use of a tree-like array-of-arrays structure, with data stored in single-element leaf arrays.

abstract states of FSET

set : Set of **integer**
ok : **boolean**

abstract states of FSETOP

$type$: $\{INS, REM\}$
key : **integer**
$done$: **boolean**
$resp$: **boolean**

GETRESPONSE(op : FSETOP) : **boolean**
  atomic
    return $op.resp$

HASMEMBER(b : FSET, k : **integer**) : **boolean**
  atomic
    return $k \in b.set$

INVOKE(b : FSET, op : FSETOP) : **boolean**
  atomic
    if $b.ok \wedge \neg op.done$ **then**
      if $op.type = INS$ **then**
        $op.resp \leftarrow op.key \notin b.set$
        $b.set \leftarrow b.set \cup \{op.key\}$
      else if $op.type = REM$ **then**
        $op.resp \leftarrow op.key \in b.set$
        $b.set \leftarrow b.set \setminus \{op.key\}$
      $op.done \leftarrow$ **true**
    return $op.done$

FREEZE(b : FSET) : Set of **integer**
  atomic
    if $b.ok$ **then**
      $b.ok \leftarrow$ **false**
    return $b.set$

Figure 1: Specification of FSET and FSETOP Objects

3. FREEZABLE SETS

In this section, we briefly introduce FSET, a freezable set object that serves as the common building block of our lock-free and wait-free hash table algorithms. Figure 1 presents the FSET specifica-

tion. Discussion of nonblocking implementations of FSET appears in Section 6 and Section 7.

An FSET object implements an integer set with insert, remove, and lookup operations, and in addition, provides a special FREEZE operation. The abstract states consist of a set of integers, and an ok bit indicating whether the set is mutable. Modification of an FSET object can be either insertion or removal of a key. Logically, an insert returns true if the key was not in the set, and a remove returns true if the key was in the set; otherwise, the modification operation returns false. However, we encode insert and remove operations as FSETOP objects. The states of an FSETOP object include the operation type (*INS* or *REM*), the key value, a boolean $done$ field that indicates whether the operation was ever applied, and a boolean $resp$ field that holds the return value.

The INVOKE operation attempts to apply an insert or a remove operation op on an FSET object b. The operation op is executed only if b is mutable ($b.ok$) and op was not applied before ($\neg op.done$). In case op is successfully applied, it is marked as done, with the return value written in its $resp$ field. The INVOKE operation returns true if op is (or was already) applied; otherwise, the operation returns false, in which case b is immutable and op was not applied.

The HASMEMBER operation tests whether the given key is in the FSET object. The FREEZE operation marks the given FSET object as immutable by setting its ok bit to false, and returns all elements of the set. The GETRESPONSE operation returns the $resp$ field of the given FSETOP object.

FREEZE renders an FSET permanently immutable, and also returns the final state of an FSET object. This plays prominently in a nonblocking resize operations: when resizing, a thread first freezes the buckets (which are implemented using FSET objects) that will be merged or split; thereafter, keys in the frozen buckets can be safely migrated into new buckets without loss or duplication.

The role of the FSETOP's $done$ bit is to ensure that every modification is applied *at most* once. This is critical to our wait-free hash set design, where threads announce operations and help each other to make progress. Using $done$, we can be sure that helping does not cause an operation to execute multiple times.

Instead of letting threads invoke insert or remove operations on an FSET object, we adopt an alternative style where modifications are performed via the INVOKE and GETRESPONSE interface. This simplifies discussion of linearization points. The linearization of every modification is within INVOKE. However, linearization and subsequent pre-response computation can occur in different phases, which facilitates the wait-free helping mechanism. Details of the wait-free FSET appear in Section 7.

4. A LOCK-FREE, DYNAMIC-SIZED HASH SET ALGORITHM

Figure 2 presents a lock-free hash set algorithm. The hash set object provides three operations: INSERT adds a key value to the set and returns true if the key was not in the set, REMOVE removes a key value from the set and returns true if the key was in the set, and CONTAINS returns whether the given key is in the set.

Our algorithm assumes the availability of a nonblocking implementation of the FSET object. In particular, all INVOKE, FREEZE, and HASMEMBER operations performed on an FSET object must be lock-free with respect to the object. We also require the implementation of GETRESPONSE to be wait-free.

4.1 Implementation

Our hash set is a linked list of HNODE (Hash Table Node) objects, where an HNODE represents a version of the set (a new version is installed during a RESIZE operation). An HNODE consists of an array of FSETs ($buckets$), with the array length stored in the $size$ field, and a $pred$ pointer that points to a predecessor HNODE object. A shared pointer $head$ points to the head of the HNODE list. For simplicity, we make the following assumptions:

(1) A RESIZE operation either doubles (grows) or halves (shrinks) the size of the bucket array.

(2) The use of modular arithmetic ($index = k$ **mod** $size$) for the hash function is acceptable.

The key challenge in our algorithm is to coordinate the resizing mechanism (embodied in the RESIZE operation) with the set operations (INSERT, REMOVE, and CONTAINS).

A RESIZE operation takes a boolean parameter that indicates whether the caller intends to grow or shrink the hash table. The thread must first ensure that all the logical key values of the set are *physically* stored in the buckets of the head HNODE. This is achieved by invoking INITBUCKET on each bucket (line 23), which migrates to t those key values stored in t's predecessor but not yet in t. After the migration is complete, we set $t.pred$ to **nil** (line 24) to allow the predecessor (which is now immutable) and its buckets to be garbage collected. The thread then allocates a new HNODE t' with t as the predecessor, and uses a CAS instruction to make t' the new head HNODE (line 28). The operation does *not* initialize entries of the new bucket array: these entries are initialized lazily as they are later accessed by INSERT and REMOVE operations.

The INITBUCKET operation initializes the i-th bucket of a given HNODE t, by merging or splitting the corresponding buckets of t's predecessor HNODE s, if s exists. The operation compares the sizes of t and s to determine whether t is growing or shrinking with respect to s, and then freezes the corresponding bucket(s) of s before copying the elements to t. If t doubles the size of s, then (roughly) half of the elements in the $(i$ **mod** $s.size)$-th bucket of s migrate to the i-th bucket of t (line 44). Otherwise, t halves the size of s, in which case the i-th and $(i + t.size)$-th buckets of s are merged to form the i-th bucket of t (line 48). Note that a new FSET object is allocated to store the merged or split bucket (line 49), and a CAS instruction is used to prevent races with helping threads (line 50).

Both INSERT and REMOVE operations delegate their work to the APPLY operation (lines 2 and 7), which applies the modification to the appropriate bucket. APPLY first allocates an FSETOP object to represent the modification request (line 30), and then repeatedly attempts to apply the request to the corresponding bucket b (line 36). If b is **nil**, the thread must invoke INITBUCKET to initialize the bucket (line 35) before applying the modification. After the modification is successfully applied (line 36 returns true), the operation receives its return value via GETRESPONSE (line 37).

A modification may trigger the resizing mechanism according to heuristic policies. Since the choice of policy is orthogonal to the algorithm, we leave it unspecified in our presentation (lines 3 and 8). As typical heuristics, INSERT might approximate the bucket size by the number of elements it visits, and grow the hash table if the cost exceeds some threshold; upon completing a REMOVE, the thread may sample the sizes of randomly selected buckets and shrink the hash table if their sizes all fall below some threshold.

A CONTAINS operation starts by searching for the given key value in the corresponding bucket b of the head HNODE. If b is not **nil**, the thread simply searches the bucket (line 18) to determine if the key value is in the set. Otherwise, the thread must trace back to t's predecessor HNODE(s) (line 15) and perform the search in s. There is one troublesome interleaving, which occurs when s is resized concurrently with the CONTAINS. Thus we must double-check (line 16) if s has become **nil** between lines 13 and 15,

```
record HNode
    buckets : FSet[ ]
    size    : integer
    pred    : HNode

shared variables
    head : HNode

initially
    head ← new HNode⟨new FSet[1], 1, nil⟩
    head.buckets[0] ← new FSet⟨∅, true⟩

1  Insert(k : integer) : boolean
2      resp ← Apply(INS, k)
3      if ⟨heuristic-policy⟩ then
4          Resize(true)
5      return resp

6  Remove(k : integer) : boolean
7      resp ← Apply(REM, k)
8      if ⟨heuristic-policy⟩ then
9          Resize(false)
10     return resp

11 Contains(k : integer) : boolean
12     t ← head
13     b ← t.buckets[k mod t.size]
14     if b = nil then
15         s ← t.pred
16         if s ≠ nil then b ← s.buckets[k mod s.size]
17         else b ← t.buckets[k mod t.size]
18     return HasMember(b, k)

19 Resize(grow : boolean)
20     t ← head
21     if t.size > 1 ∨ grow then
22         for i from 0 to t.size − 1 do
23             InitBucket(t, i)
24         t.pred ← nil
25         size ← grow ? t.size ∗ 2 : t.size/2
26         buckets ← new FSet[size]
27         t′ ← new HNode⟨buckets, size, t⟩
28         CAS(&head, t, t′)

29 Apply(type : {INS, REM}, k : integer) : boolean
30     op ← new FSetOp⟨type, k, false, −⟩
31     while true do
32         t ← head
33         b ← t.buckets[k mod t.size]
34         if b = nil then
35             b ← InitBucket(t, k mod t.size)
36         if Invoke(b, op) then
37             return GetResponse(op)

38 InitBucket(t : HNode, i : integer) : FSet
39     b ← t.buckets[i]
40     s ← t.pred
41     if b = nil ∧ s ≠ nil then
42         if t.size = s.size ∗ 2 then
43             m ← s.buckets[i mod s.size]
44             set ← Freeze(m) ∩ {x | x mod t.size = i}
45         else
46             m ← s.buckets[i]
47             n ← s.buckets[i + t.size]
48             set ← Freeze(m) ∪ Freeze(n)
49         b′ ← new FSet⟨set, true⟩
50         CAS(&t.buckets[i], nil, b′)
51     return t.buckets[i]
```

Figure 2: A Lock-free Dynamic-Sized Hash Set Implementation

in which case we re-read the corresponding bucket of t (line 17), which must have become initialized during prior to s becoming **nil**, and perform the search in it.

4.2 Linearizability

We sketch a proof of linearizability of the lock-free hash set object by showing that every INSERT, REMOVE, and CONTAINS operation happens at its linearization point, defined as follows:

An INSERT or a REMOVE operation by thread p linearizes at an INVOKE operation (line 36) that returns true (which logically sets $op_p.done$ to true). A CONTAINS operation linearizes at the HASMEMBER operation (line 18) if b is mutable ($b.ok$ is true) when the operation is performed; otherwise, b must have been made immutable by some FREEZE operation, in which case we let the CONTAINS operation linearize at the FREEZE operation that sets $b.ok$ to false, or at p's step at line 12, whichever happens later.

The major obligation of the proof is to show that a concrete hash set object refines an abstract set object (AbsSet), with respect to the mapping function in Figure 3. Intuitively, the abstract set object is defined as the union of all bucket sets of the head HNODE. A bucket set (BuckSet(t, i)) is defined as the elements of the bucket (Elems(t, i)) if the bucket pointer is not **nil**, or otherwise, the union (Merge(t, i)) or intersection (Split(t, i)) of the corresponding buckets of the predecessor HNODE.

For convenience, we say an FSET object X is *mutable* if $X.ok$, and *immutable* otherwise. We say an HNODE object X is *growing* if $X.pred \neq$ **nil** and $X.size = X.pred.size * 2$, and *shrinking* if $X.pred \neq$ **nil** and $X.size = X.pred.size \,/\, 2$.

For HNODE object X and bucket index i, we define the *predecessor buckets* of $X.buckets[i]$ as follows: if X is growing, then $X.pred.buckets[i \bmod X.pred.size]$ is the only predecessor bucket of $X.buckets[i]$; if X is shrinking, both $X.pred.buckets[i]$ and $X.pred.buckets[i + X.size]$ are predecessor buckets of $X.buckets[i]$.

The following observations can be verified from the pseudo-code and do not require proof.

OBSERVATION 1. *$head \neq$ **nil**.*

OBSERVATION 2. *The pred field of an* HNODE *does not change unless it is set to* **nil** *by a step at line 24.*

OBSERVATION 3. *The size field of a* HNODE *does not change and always equals the size of the buckets array.*

OBSERVATION 4. *For every* HNODE *object X, if $X.pred \neq$ **nil**, then X is either growing or shrinking.*

$$\text{AbsSet} \equiv \text{NodeSet}(head)$$

$$\text{NodeSet}(t) \equiv \bigcup_{i=0}^{t.size-1} \text{BuckSet}(t, i)$$

$$\text{BuckSet}(t,i) \equiv \begin{cases} \text{Elems}(t,i) & \text{if } t.buckets[i] \neq \textbf{nil} \\ \text{Split}(t,i) & \text{if } t.buckets[i] = \textbf{nil} \wedge t.pred.size * 2 = t.size \\ \text{Merge}(i) & \text{if } t.buckets[i] = \textbf{nil} \wedge t.pred.size \,/\, 2 = t.size \end{cases}$$

$$\begin{aligned} \text{Elems}(t,i) &\equiv t.buckets[i].set \\ \text{Split}(t,i) &\equiv \text{Elems}(t.pred,\ i \bmod t.pred.size) \cap \{x \mid x \bmod t.size = i\} \\ \text{Merge}(t,i) &\equiv \text{Elems}(t.pred,\ i) \cup \text{Elems}(t.pred,\ i + t.size) \end{aligned}$$

Figure 3: Refinement Mapping Function from Concrete Hash Sets to Abstract Sets

OBSERVATION 5. *For every* HNODE *object X, $X.buckets[i]$ never changes after it is initialized on line 50.*

The following invariants are mutually dependent on each other, and we prove them together in a conjunction.

INVARIANT 6. *If p is at lines 24 - 28, then $t.buckets[i] \neq$ **nil**.*

INVARIANT 7. *If p is at line 24 and $t.pred \neq$ **nil**, then $t = head$, and $t.pred.buckets[k]$ is immutable for every valid index k.*

INVARIANT 8. *If p is at lines 39 - 50 and $t.buckets[i] =$ **nil** $\vee$ $t.pred \neq$ **nil**, then $t = head$.*

INVARIANT 9. *If p is at lines 39 - 50 and $t.buckets[i] \neq$ **nil** $\wedge$ $t.pred \neq$ **nil**, then every predecessor bucket of $t.buckets[i]$ is immutable.*

INVARIANT 10. *If p is at line 50 and $t.buckets[i] =$ **nil**, then every predecessor bucket of $t.buckets[i]$ is immutable.*

INVARIANT 11. *For every* HNODE *object X, if exists i such that $X.buckets[i] =$ **nil**, then $X.pred \neq$ **nil**.*

INVARIANT 12. *For every* HNODE *object X, if $X.pred \neq$ **nil**, then $X.pred.buckets[k] \neq$ **nil** for every valid index k.*

INVARIANT 13. *For every* HNODE *object X, if $X.buckets[i] \neq$ **nil** $\wedge$ $X.pred \neq$ **nil**, then every predecessor bucket of $X.buckets[i]$ is immutable.*

The following lemma ensures that the resizing mechanism cannot change the abstract states of the set object defined by the refinement mapping function.

LEMMA 14. *AbsSet does not change between the pre-state and post-state of a step α, if α is a CAS at line 28 or line 50.*

We say an HNODE object X is *reachable* if $head = X$, or $head.pred \neq$ **nil** and $head.pred = X$. Otherwise, we say the node is *unreachable*.

LEMMA 15. *For every* HNODE *object X that is unreachable, $X.buckets[k]$ is* **nil** *or immutable for every valid index k.*

We say a thread p is *viable* if p is at the INVOKE(b, op) operation at line 36 and b is mutable.

The following lemma captures the key insight of the algorithm: for an insert or remove operation with key value k, there is always a *unique* FSET object to which the operation can be applied (The FSET dictates the corresponding subset of the abstract set). If two threads p and q both attempt to insert or remove the same key value (after modulo arithmetic), and both are at the INVOKE operation, they either invoke on the *same* bucket b, or at least one of them is invoked on an immutable bucket. This property precludes the possibility that multiple INVOKE operations concurrently insert or remove the same key value at different buckets, and hence, prevents the violation of linearizability (e.g., two INSERT operations both insert the same key values to the set and return true).

LEMMA 16. *For any viable thread p, exactly one of the following claims holds:*

(1) $b_p = head.buckets[op_p.key \bmod head.size]$*;*

(2) $b_p = head.pred.buckets[op_p.key \bmod head.pred.size]$*, and $head.buckets[j] =$ **nil** for all j such that b_p is a predecessor bucket of $head.buckets[j]$.*

THEOREM 17. *The algorithm in Figure 2 is a linearizable implementation of a set object.*

4.3 Lock Freedom

We show that from any reachable configuration, some INSERT, REMOVE or CONTAINS operation completes in finite number of steps. First, note that a CONTAINS operation cannot delay indefinitely between lines 12 and 17, and the final call to HASMEMBER is lock-free by definition. An INSERT or a REMOVE operation consists of a call to APPLY and a potential call to RESIZE. We show that an APPLY operation takes the back edge of the while loop at line 31 only if another RESIZE operation (called by an INSERT or REMOVE) completes. Since we maintain the invariant that every bucket of the head HNODE is mutable, for an INVOKE operation of thread p to fail, p must encounter an immutable bucket. Since a bucket is made immutable only by a RESIZE, then for T threads, if p's APPLY fails more than T times, then it means that even if $T-1$ threads were all in RESIZE when APPLY was called, the T-th failure of p's APPLY indicates that some thread must have finished its RESIZE, then called APPLY again, indicating that it succeeded in another INSERT or REMOVE.

Suppose S is the maximum size of the head HNODE during execution. Then a RESIZE operation contains at most $2S$ FSET operations. Each iteration of the while loop in APPLY includes at most 3 FSET operations (at most 2 in INITBUCKET, and one in INVOKE). Therefore, at least one INSERT, REMOVE or CONTAINS operation must complete upon the completion of $(3T + 2S) \cdot T$ FSET operations, and hence, the hash set implementation is lock-free by the assumption that the FSET operations are lock-free.

5. A WAIT-FREE ALGORITHM

In this section, we extend the lock-free implementation to obtain a wait-free hash set algorithm. Our wait-free hash set algorithm assumes that a *wait-free* FSET implementation is available. We start with an illustration of the main challenge of achieving wait-freedom. Recall that in our lock-free hash set algorithm, FSET objects are only required to be lock-free. Now suppose a wait-free FSET implementation is given. Does the algorithm immediately become wait-free? The answer is negative: as demonstrated in the following example, an APPLY operation may take an infinite number of steps to complete, due to concurrent RESIZE operations performed on the hash set object.

Let thread p attempt to insert some key value k into the hash set, and stall at the INVOKE operation at line 36. Let t be the head of the HNODE list and let b be the corresponding bucket where p wishes to perform the insertion. Now let another thread q complete an INSERT operation that triggers a RESIZE operation on the hash set, after which a new object t' becomes the head of the HNODE list. Now suppose q inserts the same key value k into the hash set, and since all buckets of t' are **nil**, q invokes INITBUCKET to initialize the corresponding bucket of t', which freezes b, the corresponding bucket of its predecessor t. When thread p resumes, its INVOKE operation will fail since b is frozen (immutable), and p will repeat the while loop in the APPLY operation. The above process can repeat forever, by alternating removals and insertions of k by q, so that p's APPLY operation never completes.

5.1 A Wait-free Implementation of Apply

We present a wait-free implementation of the APPLY operation in Figure 4. The basic idea is to let threads help each other to complete their APPLY operations instead of constantly competing to change and/or freeze the buckets. Our helping mechanism is similar to the doorway stage of Lamport's bakery algorithm [13].

In the wait-free algorithm, an insert or remove operation is represented using a WFOP object which adds a $prio$ field to the FSETOP object. The $prio$ field represents the "priority" of an operation, which dictates the operation's precedence in the helping mechanism: an operation with smaller $prio$ has the precedence over one with larger $prio$. The priorities of operations are generated from a strictly increasing counter (initially 0), implemented using an atomic fetch-and-increment instruction (line 53).

In APPLY, thread p first allocates an WFOP object for its modification operation, associated with a unique priority, and then announces the object (line 55) in a shared array A, indexed by p's thread id. Then p iterates through A and for any operation op announced by thread q (including p itself), if $op.prio$ is smaller than (or equal to) p's most recent priority, p helps q complete (lines 59 to 64). Finally, p invokes GETRESPONSE to get the return value of its own operation (line 65).

5.2 Wait Freedom

First, observe that CONTAINS is wait-free, as it does not have any loops, and the call to HASMEMBER is wait-free. To demonstrate that INSERT and REMOVE are wait-free, we show that for T threads, the inner while loop at line 58 executes at most T iterations. For thread p whose INVOKE at line 63 returns false, the head HNODE must be changed between p's line 59 and line 63, since no bucket of head can be in a frozen state. The change of head must be made by a step at line 28 of some RESIZE, indicating the completion of the outer INSERT or REMOVE operation. After T iterations of the while loop, some thread must have completed at least 2 INSERT or REMOVE operations, where the second one must have a lower priority (larger $prio$) than the priority of op_p. Thus, $op_p.done$ must have been set to true, and p's INVOKE will return true in the next iteration. Therefore, APPLY operations are wait-free, since each contains at most $O(T^2)$ FSET operations, which are wait-free by assumption.

```
record WFOP extends FSETOP
  prio    : integer

additional shared variables
  A       : WFOP[THREADS]
  counter : integer

initially
    counter ← 0
    for tid ← 0 to (THREADS − 1) do
        A[tid] ← new WFOP⟨−, −, −, −, ∞⟩

52  APPLY(type : {INS, REM}, k : integer) : boolean
53      prio ← F&I(&counter)
54      myop ← new WFOP⟨type, k, false, −, prio⟩
55      A[threadid] ← myop
56      for tid ← 0 to (THREADS − 1) do
57          op ← A[tid]
58          while op.prio <= prio do
59              t ← head
60              b ← t.buckets[op.key mod t.size]
61              if b = nil then
62                  b ← INITBUCKET(t, op.key mod t.size)
63              if INVOKE(b, op) then
64                  break
65      return GETRESPONSE(myop)
```

Figure 4: A Wait-free Implementation of APPLY

6. A SPECIALIZED LOCK-FREE FSET IMPLEMENTATION

Figure 5 presents a lock-free FSET implementation specialized for the lock-free hash set in Figure 2. It exploits the property that in the lock-free hash set algorithm, every FSETOP object can only be applied to a bucket FSET by the allocating thread, due to absence of helping, and thus, we need not keep an actual $done$ field in an FSETOP object.

The idea of the implementation is straightforward: we keep the underlying FSET objects (namely FSETNODEs) immutable, and let all updates be performed in a copy-on-write manner. We maintain a pointer $node$ that points to the current FSETNODE object, which consists of the elements of the set (set) and a bit (ok) indicating whether the set is mutable. Any update to an FSET, either via an INVOKE or an FREEZE operation, must first allocate a new

```
record FSetNode
  set  : Set of integer
  ok   : boolean

record FSet
  node : FSetNode

record FSetOp
  type : {INS, REM}
  key  : integer
  resp : boolean

66 FREEZE(b : FSET) : Set of integer
67   o ← b.node
68   while o.ok do
69     n ← new FSetNode⟨o.set, false⟩
70     if CAS(&b.node, o, n) then
71       break
72     o ← b.node
73   return o.set

74 INVOKE(b : FSET, op : FSetOp) : boolean
75   o ← b.node
76   while o.ok do
77     if op.type = INS then
78       resp ← op.key ∉ o.set
79       set ← o.set ∪ {op.key}
80     else if op.type = REM then
81       resp ← op.key ∈ o.set
82       set ← o.set \ {op.key}
83     n ← new FSetNode⟨set, true⟩
84     if CAS(&b.node, o, n) then
85       op.resp ← resp
86       return true
87     o ← b.node
88   return false

89 HASMEMBER(b : FSET, k : integer) : boolean
90   o ← b.node
91   return k ∈ o.set

92 GETRESPONSE(op : FSetOp) : boolean
93   return op.resp
```

Figure 5: A Specialized Lock-free FSet Implementation

FSetNode object (cloned from the current FSetNode), then apply its change, and then finalize the modification with a CAS instruction that points *node* to the new FSetNode.

The immutable nature of FSetNode objects allows us to implement the inner set using any sequential algorithm. Since each bucket of a hash table tends to contain only a small number of elements, one appealing option is an unsorted array: it exploits better cache locality than list-based alternatives, and affords the compiler an opportunity to employ wide "vector" operations.

We also note that some simple (but useful) optimizations are elided in the pseudo code to avoid clutter. In particular, it would be wise to let an insert (or remove) operation exit early if the key value is (or is not) in the set, thereby avoiding an unnecessary update (allocation, CAS).

7. A COOPERATIVE WAIT-FREE FSET IMPLEMENTATION

Figure 6 presents an FSet implementation designed for our wait-free hash set algorithm. The FSet implementation "cooperates" with the helping mechanism (Figure 4) to achieve wait-freedom[1]. We inherit the immutable design of the underlying set objects as in the previous lock-free FSet implementation. Now, however, the wait-free implementation must prevent duplicate execution of operations due to the presence of helping. This is achieved by leveraging the $prio$ field: for any FSetOp object op, its abstract $done$ field is **true** if $op.prio$ is set to ∞, and we maintain an invariant that op is performed only if $op.prio$ is *not* ∞.

The crux of the protocol is to let contending threads synchronize at the op field of an FSetNode object. To perform an FSetOp (op), a thread first attempts to change $node.op$ from **nil** using CAS. Subsequently, the thread invokes HelpFinish to compute the return value of op, marks op as done, and replaces the current FSetNode with the result set (a new object) using CAS.

A Freeze operation first announces its intention by setting $flag$ to **true**, and invokes DoFreeze to set $node.op$ to $\perp$. To show a Freeze operation is wait-free, it is sufficient to show that DoFreeze completes in a finite number of steps. For any thread p in a DoFreeze operation, we notice that the CAS at line 123 can fail only because $node.op$ is changed by a concurrent CAS at line 123 or line 102. In the former case, $node.op$ is set to $\perp$ and p's while loop will terminate in the next iteration. In the latter case, any subsequent Invoke operation will see that $flag$ is set, and invoke DoFreeze. Thus, either p succeeds in the CAS in its next iteration, or a concurrent DoFreeze sets $node.op$ to $\perp$, which forces p to terminate its while loop in the following iteration.

A HasMember operation must first check if there exists a linearized insert or remove operation by inspecting the op field. This is necessary because an Invoke operation on an FSet object b may return **true** without invoking HelpFinish on b (in cases where op is performed by a concurrent thread), leaving b in an intermediate state. Neglecting to check the op field can cause a subsequent Contains operation to erroneously miss the immediately preceding insert or remove operation, which violates linearizability.

8. PERFORMANCE EVALUATION

We evaluate the performance of our hash tables via a stress-test microbenchmark. The experiments were run on a Niagara2 system with one 1.165 GHz, 64-way Sun UltraSPARC T2 CPU with 32 GB of RAM, running Solaris 10. The Niagara2 has eight cores, each eight-way multi-threaded, for a total of 64 threads. We used the Oracle JDK version 1.7.0_13. We also run experiments on an x86 system with one 2.66GHz Intel Xeon X5650 processor and 6GB of RAM, running Linux kernel 3.11. The processor has six cores, each two-way multi-threaded, for a total of 12 threads. On the x86, we used OpenJDK version 1.7.0_51. Due to space limitations, we elide the x86 performance charts.

We compare eight implementations. We use the Shalev and Shavit split ordered list [17] (**SplitOrder**) as our baseline. Our Java implementation of SplitOrder used the latest C++ version as a reference, to ensure a faithful implementation. To the best of our knowledge, this is the best-performing algorithm for implementing an extensible (but not shrinkable) hash table. Furthermore, the implemen-

[1]The implementation is lock-free by itself.

```
record FSetNode
  set  : Set of integer
  op   : FSetOp ∪ {⊥}

record FSet
  node : FSetNode
  flag : boolean

record FSetOp
  type : {INS, REM}
  key  : integer
  resp : boolean
  prio : integer

94  Invoke(b : FSet, op : FSetOp) : boolean
95      while b.node.op ≠ ⊥ ∧ op.prio ≠ ∞ do
96          if b.flag then
97              DoFreeze(b)
98              break
99          o ← b.node
100         if o.op = nil then
101             if op.prio ≠ ∞ then
102                 if CAS(&o.op, nil, op) then
103                     HelpFinish(b)
104                     return true
105         else
106             HelpFinish(b)
107     return op.prio = ∞

108 Freeze(b : FSet) : Set of integer
109     b.flag ← true
110     return DoFreeze(b)

111 HasMember(b : FSet, k : integer) : boolean
112     o ← b.node
113     op ← o.op
114     if op ≠ nil ∧ op ≠ ⊥ ∧ op.key = k then
115         return op.type = INS
116     return k ∈ o.set

117 GetResponse(op : FSetOp) : boolean
118     return op.resp

119 DoFreeze(b : FSet) : Set of integer
120     while b.node.op ≠ ⊥ do
121         o ← b.node
122         if o.op = nil then
123             if CAS(&o.op, nil, ⊥) then
124                 break
125         else
126             HelpFinish(b)
127     return b.node.set

128 HelpFinish(b : FSet)
129     o ← b.node
130     op ← o.op
131     if op ≠ nil ∧ op ≠ ⊥ then
132         if op.type = INS then
133             resp ← op.key ∉ o.set
134             set ← o.set ∪ {op.key}
135         else if op.type = REM then
136             resp ← op.key ∈ o.set
137             set ← o.set \ {op.key}
138         op.resp ← resp
139         op.prio ← ∞
140         CAS(&b.node, o, new FSetNode⟨set, nil⟩)
```

Figure 6: A Cooperative Wait-free FSet Implementation

tation optimized its configuration of the directory structure (using a two-level tree) for the size of each experiment. This ensures a minimal bucket size for the duration of each experiment.

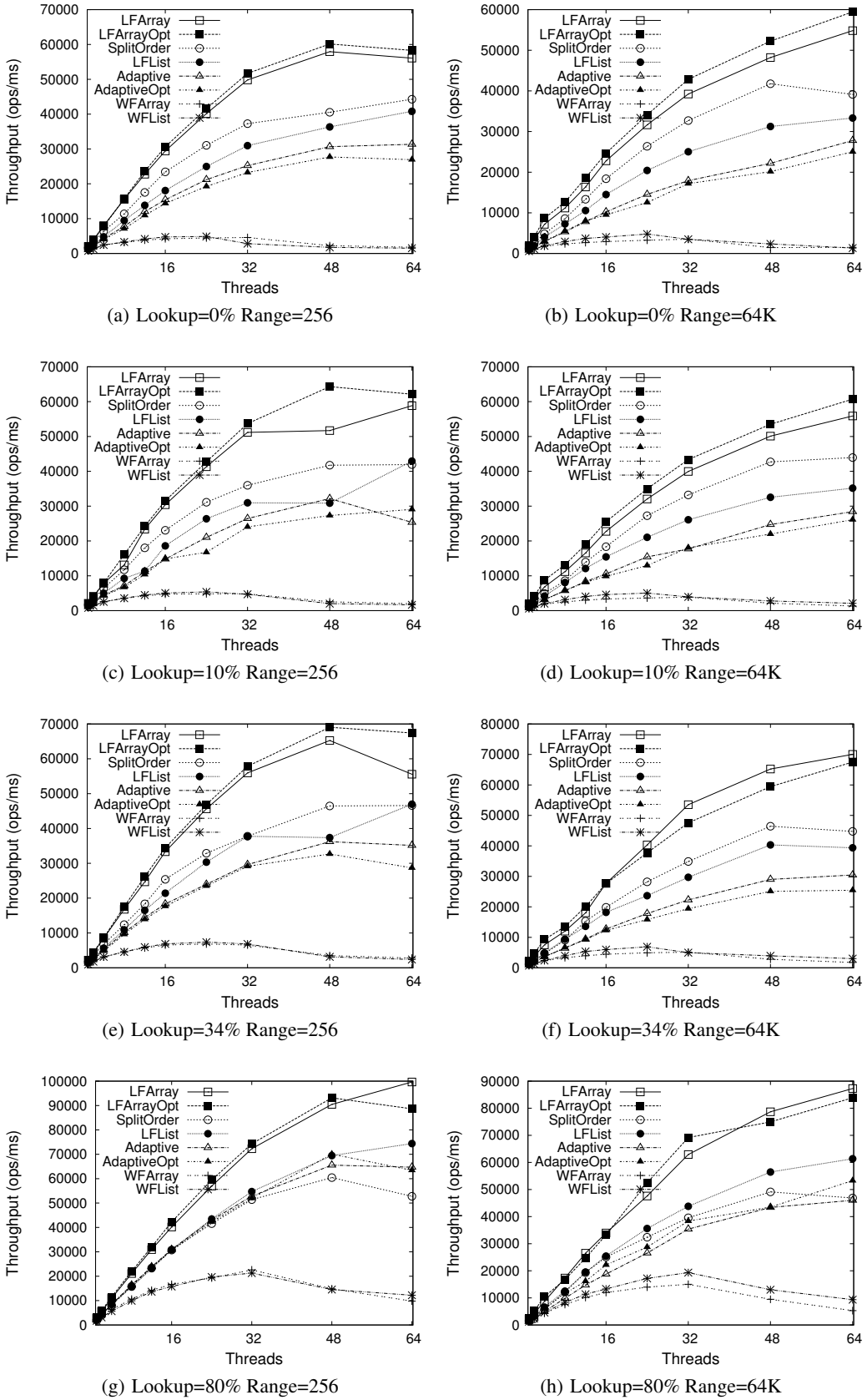
Unlike the baseline, the remaining seven algorithms were run with support for dynamic resizing. **LFArray** is our lock-free hash table, in which per-bucket freezable sets are implemented as arrays of unsorted elements (Section 6). **LFArrayOpt** removes a level of indirection from LFArray by pointing buckets directly to array elements, rather than FSET markers. **LFList** is similar to **LFArray**, except it uses an unsorted list implementation [20] for its freezable sets. In addition to the above lock-free implementations, we consider four wait-free implementations, which use a wait-free FSET (Section 7). **WFArray** and **WFList** employ the straightforward wait-free APPLY from Figure 4 to make the LFArray and LFList algorithms wait-free. **Adaptive** applies the Fast-path/Slowpath technique [11] to reduce the overhead of WFArray, and **AdaptiveOpt** applies the optimizations from LFArrayOpt to Adaptive. All adaptive algorithms used a threshold of 256 consecutive failures to trigger a switch to the slow path. All implementations (to include SplitOrder) were optimized using techniques from the `java.util.concurrent` package.

Our main focus is performance in the absence of resizing operations. To this end, for a given experiment we begin by pre-populating each hash table to hold half of the experiment's key range. For a lookup ratio L, we randomly select operations such that insert and remove are chosen with equal probability $(1-L)/2$. Thus while operations and keys are randomly selected, the number of elements in the table remains steady. We report the average of five 5-second trials. Variance was negligible.

8.1 SPARC Performance

Figure 7 presents performance on the Niagara2. The Niagara2 has simple in-order cores with per-core L1 caches and a shared L2 cache. While memory access latencies are low, there is no out-of-order execution to hide the latency of memory accesses: during a cache miss, another hardware thread is scheduled. For all but the smallest key range, the random distribution of keys ensures that every access will incur an L1 cache miss to dereference the bucket, regardless of the implementation. Since SplitOrder uses a sorted list, whereas LFList uses an unsorted list, SplitOrder should traverse half as many pointers, on average. However, our efficient mechanism for finding buckets keeps the gap below $2\times$.

When we implement each per-bucket FSET as an array, a new trade-off is introduced. On the one hand, all pointer chasing within a bucket is eliminated; on the other, insert and remove operations must copy the entire array. The high memory bandwidth of the Niagara2, coupled with the absence of pointer chasing within each bucket, result in superior performance for LFArray and LFArrayOpt. The most important factor in hash table performance on Niagara2 appears to be pointer chasing.



(a) Lookup=0% Range=256

(b) Lookup=0% Range=64K

(c) Lookup=10% Range=256

(d) Lookup=10% Range=64K

(e) Lookup=34% Range=256

(f) Lookup=34% Range=64K

(g) Lookup=80% Range=256

(h) Lookup=80% Range=64K

Figure 7: Microbenchmark Performance on Niagara2

The WFArray and WFList implementations show limited scaling, except when lookups dominate. The poor performance is due to the cost of announcing every operation: incrementing a single global shared counter is a bottleneck, as is the use of the WFOP array for announcing operations and finding threads to help. The Fastpath/Slowpath technique does much to recover this cost. However, even though our threshold of 256 failures virtually guarantees no fallbacks to the slow path, wait-free FSET operations still carry a cost due to extra memory indirection and allocation.

8.2 x86 Performance

In experiments on a 6-core/12-thread Intel Xeon 5650, our algorithms behaved similarly to the Niagara2. The key differences were that there was little difference between LFArray and LFArray-Opt, and that, at high lookup ratios, the adaptive algorithms were able to close much of the gap with LFList. Given the much different microarchitecture of the X5650, the lack of significant difference between these results and those reported for the Niagara2 give confidence that the behaviors we observed were consequences of the fundamental characteristics of our algorithms, rather than architecture-specific anomalies. In particular, the locality afforded by implementing each bucket as an array enables LFArray to outperform SplitOrder in most cases.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced new lock-free and wait-free hash tables, which differ from prior efforts in that they allow keys to be moved among buckets during a resize, without sacrificing throughput or progress. Our technique allows the hash table's buckets to be implemented as arrays, which increases locality and reduces pointer chasing. Our practical wait-free algorithms also demonstrate the resilience of the Fastpath/Slowpath technique [11]: it took little effort to make our lock-free algorithms wait-free, and the result was dramatically faster than a naive wait-free solution.

The interaction between the FSET objects used within the table and the implementation of scaffolding of the hash table leaves some opportunity for improvement. Most significantly, the copy-on-write algorithm we use might be replaced by hardware transactions, if it were possible to still guarantee progress. Another question involves extending the set to a map: in this case, again, the copy-on-write technique is likely to prove valuable, since it avoids the need to atomically modify distinct key and value fields.

Acknowledgments

We thank Alex Kogan, Victor Luchangco, and our anonymous reviewers, whose detailed suggestions greatly improved this paper.

10. REFERENCES

[1] C. S. Ellis. Extendible Hashing for Concurrent Operations and Distributed Data. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Atlanta, GA, Mar. 1983.

[2] C. S. Ellis. Concurrency in Linear Hashing. *ACM Transactions on Database Systems*, 12(2):195–217, 1987.

[3] S. Feldman, P. LaBorde, and D. Dechev. Concurrent Multi-Level Arrays: Wait-Free Extensible Hash Maps. In *Proceedings of the 13th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos Island, Greece, July 2013.

[4] H. Gao, J. F. Groote, and W. H. Hesselink. Almost Wait-Free Resizable Hashtable. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, Apr. 2004.

[5] M. Greenwald. Two-Handed Emulation: How to Build Non-Blocking Implementation of Complex Data-Structures using DCAS. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.

[6] T. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, Oct. 2001.

[7] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[8] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing*, Arcachon, France, Sept. 2008.

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

[10] M. Hsu and W.-P. Yang. Concurrent Operations in Extendible Hashing. In *Proceedings of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, Aug. 1986.

[11] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-Free Data Structures. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012.

[12] V. Kumar. Concurrent Operations on Extendible Hashing and its Performance. *Communications of the ACM*, 33(6):681–694, 1990.

[13] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[14] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2003.

[15] M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, Aug. 2002.

[16] C. Purcell and T. Harris. Non-blocking Hashtables with Open Addressing. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[17] O. Shalev and N. Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *Journal of the ACM*, 53(3):379–405, 2006.

[18] J. Triplett, P. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the USENIX Annual Technical Conference*, Portland, OR, June 2011.

[19] D. Zhang and P.-Å. Larson. LHlf: Lock-Free Linear Hashing (Poster Paper). In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012.

[20] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear. Practical Non-Blocking Unordered Lists. In *Proceedings of the 27th International Symposium on Distributed Computing*, Jerusalem, Israel, Oct. 2013.