Hopscotch Hashing

Maurice Herlihy
Brown University
and
Nir Shavit
Tel-Aviv University
and
Moran Tzafrir
Tel-Aviv University

We present a new resizable sequential and concurrent hash map algorithm directed at both uniprocessor and multicore machines. The algorithm is based on a novel *hopscotch* multi-phased probing and displacement technique that has the flavors of chaining, cuckoo hashing, and linear probing, all put together, yet avoids the limitations and overheads of these former approaches. The resulting algorithm provides a table with very low synchronization overheads and high cache hit ratios.

In a series of benchmarks on a state-of-the-art 64-way Niagara II multicore machine, a concurrent version of the new algorithm proves to be highly scalable, delivering in some cases 2 or even 3 times the throughput of today's most efficient concurrent hash algorithm, Lea's `ConcurrentHashMap` from *java.concurr.util*. Moreover, in tests on both Intel and Sun uni-processor machines, a sequential version of the algorithm consistently outperforms the most effective sequential hash table algorithms including cuckoo hashing and bounded linear probing.

The most interesting feature of the new hopscotch algorithm is that it continues to deliver good performance when the table is more than 90% full, increasing its advantage over other algorithms as the table density grows.

[0] **Contact author is Moran Tzafrir**: `morantza@gmail.com`. This is a *Regular paper*. Moran is a *full time student* and the paper is eligible for the best student paper award.

1. INTRODUCTION

Hash tables are one of the most widely used data structures in computer science. They are also one of the most thoroughly researched, because any improvement in their performance can benefit millions of applications and systems.

A typical resizable 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 add(), remove(), and contains() method calls [1]. Typical usage patterns for hash tables have an overwhelming fraction of contains() calls [7], and so optimizing this operation has been a target of many hash table algorithms.

This paper introduces *hopscotch hashing*, a new type of open addressed resizable hash table that is directed at cache-sensitive machines, a class that includes most, if not all, of the state-of-the-art uniprocessors and multicore machines. On such machines it provides a contains() method that runs in deterministic constant time, and in practice requires two cache loads (Lemma 3.3).

1.1 Background

Chained hashing [5] is closed address hash table scheme consisting of an array of buckets each of which holds a linked list of items. Though closed addressing is superior to other approaches in terms of the time to find an item, its use of dynamic memory allocation and the indirection makes for poor cache performance [8]. It is even less appealing for a concurrent environment as dynamic memory allocation typically requires a thread-safe memory manager or garbage collector, adding overhead in a concurrent environment.

Linear probing [5] is an open-addressed hash scheme in which items are kept in a contiguous array, each entry of which is a bucket for one item. A new item is inserted by hashing the item to an array bucket, and scanning forward from that bucket until an empty bucket is found. Because the array is accessed sequentially, it has good cache locality, as each cache line holds multiple array entries. Unfortunately, linear probing has inherent limitations: because every contains() call searches linearly for the key, performance degrades as the table fills up (when the table is 90% full, the expected number of locations to be searched until a free one is found is about 50 [5]), and clustering of keys may cause a large variance in performance. After a period of use, a phenomenon called *contamination* [2], caused by deleted items, degrades the efficiency of unsuccessful contains() calls.

Cuckoo hashing [8] is an open-addressed hashing technique that unlike linear probing requires only a deterministic constant number of steps to locate an item. Cuckoo hashing uses two hash functions. A new item x is inserted by hashing the item to two array indexes. If either slot is empty, x is added there. If both are full, one of the occupants is displaced by the new item. The displaced item is then re-inserted using its other hash function, possibly displacing another item, and so on. If the chain of displacements grows too long, the table is resized. A disadvantage of cuckoo hashing is the need to access sequences of unrelated locations on different cache lines. A bigger disadvantage is that Cuckoo hashing tends to perform poorly when the table is more than 50% full because displacement sequences become too long, and the table needs to be resized.

Lea's algorithm [6] from *java.util.concurrent*, the Java™ Concurrency Package, is probably the most efficient known concurrent resizable hashing algorithm. It is a closed address hash algorithm that uses a small number of high-level locks rather than a lock per bucket. Shalev and Shavit [10] present another high-performance lock-free closed address resizable scheme. Purcell and Harris [9] were the first to suggest a nonresizable open-addressed concurrent hash table based on *linear probing*. A concurrent version of cuckoo hashing can be found in [3].

2. THE NEW HOPSCOTCH ALGORITHM

Hopscotch hashing combines the advantages of these three approaches in the following way. There is a single hash function h. The item hashed to an entry will always be found either in that entry, or in one of the next $H-1$ entries, where H is a constant (in our implementation, H is 32, the standard machine word size). In other words, a "virtual" bucket has fixed size and overlaps with the next $H-1$ buckets. Each entry includes a *hop-information* word, an H-bit bitmap that indicates which of the next $H-1$ entries contain items that hashed to the current entry's virtual bucket. In this way, an item can be found quickly by looking at the word to see which entries belong to the bucket, and then scanning through the constant number of entries (on most machines this requires at most two loads of cache lines).

Here is how to add item x where $h(x) = i$:

—Starting at i, use linear probing to find an empty entry at index j.

—If the empty entry's index j is within $H-1$ of i, place x there and return.

—Otherwise, j is too far from i. To create an empty entry closer to i, find an item y whose hash value lies between i and j, but within $H-1$ of j, and whose entry lies below j. Displacing y to j creates a new empty slot closer to i. Repeat. If no such item exists, or if the bucket already i contains H items, resize and rehash the table.

In other words, the idea is that hopscotch "moves the empty slot towards the desired bucket" instead of leaving it where it was found as in linear probing, or moving an item out of the desired bucket and only then trying to find it a new place as in cuckoo hashing. The cuckoo hashing sequence of displacements can be cyclic, so implementations typically abort and resize if the chain of displacements becomes too long. As a result, cuckoo hashing works best when the table is less than 50% full. In hopscotch hashing, by contrast, the sequence of displacements cannot be cyclic: either the empty slot moves closer to the new item's hash value, or no such move is possible. As a result, hopscotch hashing supports significantly higher loads (see Section 4). Moreover, unlike in cuckoo hashing, the chances of a successful displacement do not depend on the hash function h, so it can be a simple function that is easily shown to be close to universal.

Hopscotch has the advantage of buckets with multiple items, but unlike in chaining they have great locality since they are located in neighboring memory locations. It also inserts items in constant expected time as in linear probing, but without clustering and with the guarantee given by the hopscotch displacement scheme that items are always found in their bucket in deterministic constant time.

Finally, notice that if more than a constant number of items are hashed by h into a given bucket, the table needs to be resized. Luckily, as we show, for a universal hash function h, the probability of this type of resize happening given $H = 32$ is $1/32!$. [1] In practice, the hop information, 32 keys, and their pointers to data, all fit

[1] In our implementation we can actually use a pointer/displacement scheme instead of the easier to explain *hop-information* word above, and so buckets can actually grow H dynamically.

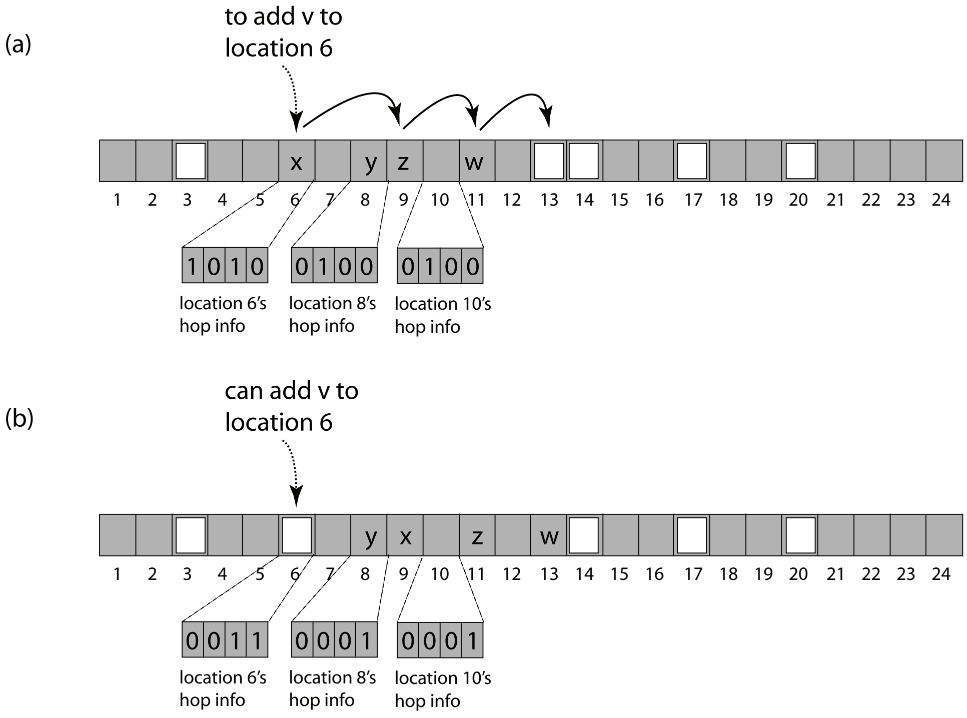

Fig. 1. The blank entries are empty, all others contain items. Here, H is 4. In part (a), we add item v with hash value 6. A linear probe finds entry 13 is empty. Because 13 is more than 4 entries away from 6, we look for an earlier entry to swap with 13. The first place to look is $H - 1 = 3$ entries before, at entry 10. That entry's hop information bit-map indicates that w at entry 11 can be displaced to 13, which we do. Entry 11 is still too far from entry 6, so we examine entry 8. The hop information bit-map indicates that z at entry 9 can be moved to entry 11. Finally, x at entry is moved to entry 9. Part (b) shows the table state just before adding v.

in two cache lines, so a contains() incurs at most two cache invalidations. Figure 1 shows an example execution of the algorithm.

The concurrent version of the hopscotch algorithm maps a bucket to each lock. This lock controls access to all table entries that hold items of that bucket. The contains() calls are wait-free, they ignore the lock and simply traverse the hop information and keys in the bucket, the while add() and remove() acquire the bucket lock before applying modifications to the data. For lack of space, we provide the detailed description and code in the appendix, and postpone the description of the resize () method, which is rather straightforward, to the full version of the paper..

3. COMPLEXITY

We analyze the complexity properties common to both the sequential and concurrent versions of the hopscotch algorithm. Outlines of the necessary proofs of safety and liveness for the concurrent implementation can be found in the appendix, to be viewed at the committee's discretion.

The most important property of a hash table is its expected constant time performance. In all the lemmas below we assume that the hash function h is universal and follow the standard practice of modeling the hash function as a uniform distribution over keys [1]. As before, H is the maximal number of keys a bucket can hold, a constant in our implementation, n is the number of keys in the table, m is the table size, and $\alpha = n/m$ is density or load of the table.

Since H, the bucket size, is a predefined constant, the following is immediated from the code:

Lemma 3.1. *The contains() and remove() methods complete in constant time (deterministic).*

How large do we need to make this constant so that bucket overflows do not cause repeated resize () calls?

Lemma 3.2. *The expected number of items in a bucket is*

$$f(m,n) = 1 + \frac{1}{4}((1+\frac{2}{m})^n - 2\frac{n}{m}) \approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{4}$$

Proof. The expected number of items in a hopscotch bucket is the same as the expected number of items in a chained-hashing bucket. The result follows from Theorem 15 in Chapter 6.4 of [5]. □

The following shows that in hopscotch hashing, as in chained hashing, in the common case there are very few items in a bucket.

Lemma 3.3. *The maximal expected number of items in a bucket is constant.*

Proof. Again, following [5], the function $f(m,n)$ is increasing, and the maximal value for n, the number of items, is m, so that the value the function evaluates to approximately 2.1. □

This implies that that typically there is very little work to be performed when searching for an item in a bucket.

It is known that the worst case number of items in a bucket, even when using a universal hash function, is not constant [5]. So what are the chances of a resize () due to overflow? It turns out that chances are low.

Lemma 3.4. *The probability of having more then H keys in a bucket is $1/H!$.*

Proof. The probability of having H keys in a bucket is equal to the probability of having a chain of length H in chained hashing. From section 3.3.10 of [2] it follows that

$$\Pr\{\text{H items in bucket}\} = \binom{n}{H}\frac{(m-1)^{n-H}}{m^n} =$$

$$\frac{n!}{H!(n-H)!}\frac{(m-1)^{n-H}}{m^n} = \frac{1}{H!}n(n-1)\ldots(n-H+1)\frac{(m-1)^{n-H}}{m^{n-H}m^H} \leq \frac{1}{H!}$$

We reach the last inequality by substituting n with m, since m is its maximal value, and the function is monotonically increasing. □

Thus, with uniform hashing, the probability of a resize () method call due to having more than $H = 32$ items in a bucket is $1/32!$.

It remains to be shown that:

Lemma 3.5. *The add() method completes within expected constant time.*

Proof. From [5] it follows that given an open-address hash table that is $\alpha = n/m < 1$ full, the expected number of entries until an open slot is found is at most $1/2(1+(1/(1-\alpha))^2)$ which is constant. Thus, the expected number of displacements, which is bounded by the number of entries tested until an open slot is found is at most constant. □

For example, if the hash table is 50% full, the expected number of entries tested until an open slot is found is at most $1/2(1+(1/(1-0.5))^2) = 2.5$ and when it is 90% full, the expected number of entries tested is $1/2(1+(1/(1-0.9))^2) = 50$.

In the full paper we will show that:

Lemma 3.6. *The resize() method completes within $O(n)$ time.*

4. PERFORMANCE EVALUATION

This section empirically compares the performance of hopscotch hashing to the most effective former algorithms in both concurrent (multicore) and sequential (uniprocessor) settings. Though in the future it would be great to evaluate our algorithm in the context of real applications, here we use the standard approach taken in the literature, using the micro-benchmarks similar to those used by recent papers in the area [8; 10], but with much higher table-densities (up to 90%).

4.1 The Overall Benchmarking Setup

In our benchmarks we sampled each test point 10 times, and plotted the average. To make sure that the table does not fit into the cache-line, and to make sure all behavioral aspects of the data-structure are exposed, we used a table-size of approx 2^{23} items. Each test used the same set of keys for all the hash-maps. All tested hash-maps were implemented using C++, and were compiled using the same compiler and settings. Closed-address hash-maps, like chained-hashing, dynamically allocate list nodes, in contrast with open-address hash-maps like linear-probing. To prevent any effects of memory-management scheme on performance, we show the results for the closed-address hash-maps both with the (mtmalloc library, multi-threaded malloc library) memory allocation library and with pre-allocated memory.

In our benchmarks in all the algorithms each bucket had a pair of pointers to the key and data (satellite key and data). This scheme is thus a general hash-map.

4.2 Concurrent Hash-Maps on Multicores

We begin by presenting a comparison of the concurrent version of hopscotch hashing to the two best performing resizable concurrent hash-table algorithms in the literature on a multicore machine.

—*Lock-based Chained*: This is the algorithm due to Lea[6].

—*Lock-Free Chained*: This is the lock-free split-ordered hash table of Shalev and Shavit [10]. (The final paper would include this benchmark)

—*New Hopscotch*: we used the concurrent version of our algorithm with a hop-information representation of the buckets.

Finally, to neutralize any effects of the choice of locks, we used the same type of locks and the same lock structure, a lock per memory segment, in all the lock-based concurrent algorithms.

We ran a series of benchmarks on a 64-way Sun UltraSPARC T2^{TM}. This is a multi-core machine based on the Niagara architecture that has 8 cores, each supporting 8 multiplexed hardware threads.

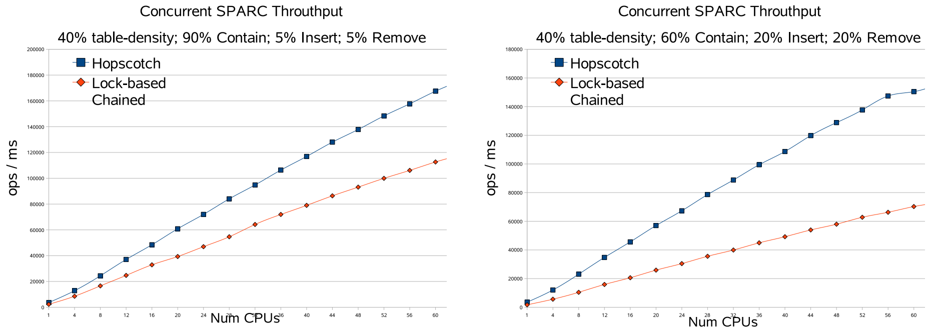

Fig. 2. The throughput as a function of concurrency at 40% table-density.

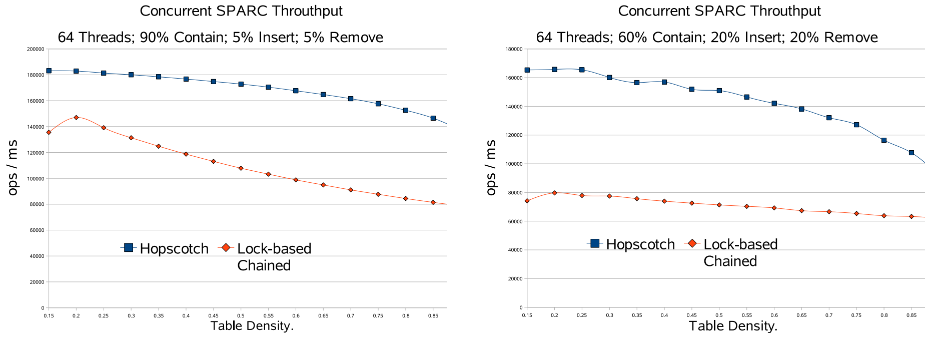

Fig. 3. Throughput at 64 thread concurrency with varying table density.

4.3 Concurrent Benchmark Results

In Figure 2, we analyzed the general throughput of the commonly found distribution of actions: 90% contains(), 5% add(), and 5% remove() and the less common one of 60% contains(), 20% add(), and 20% remove(). We used table-densities of 40%, and increased the concurrency up to 64 threads, the maximal number of actual hardware threads supported by the machine. As can be seen, the new hopscotch throughput scales better, and has about 2 times the throughput of the Lea's chained algorithm. There is still a significant gap between the algorithms even when we eliminate the memory management and pre-allocate all the space the dynamic memory algorithms require. Given that the locks are the same, and the memory management effects were eliminated, what is seen in the graphs is the pure advantage of the Hopscotch approach. The reason for this advantage is that it has better cache behavior: finding an item takes two cache misses, proved at Lemma 3.3.

In Figure 3, we analyze the change in throughput as a function of the table density at a high level of 64 thread concurrency, effectively the number of hardware threads on the machine. In each sample point we maintained the table-density stable at a given level. As can be seen, hopscotch has significantly superior performance at all densities. as the density increases, the hopscotch

4.4 Sequential Hash-Maps on Uniprocessors

We selected the most effective known sequential hash-maps.

—*Linear-Probing*: we used an optimized version [5], that stores the maximal probe length in each bucket.

—*Chained*: We used an optimized version of [5] that stores the first element of each linked-list directly in the hash table.

—*Cuckoo*: Thanks to the kindness of the authors of [8], we obtained the original cuckoo hash map code.

—*New Hopscotch*: we used the sequential version, and a list-like hop-information representation, that enables us to test the performance at table-densities up to 99%.

4.5 Sequential Setup

We ran a series of benchmarks on two fundamentally different uniprocessor architectures: *SPARC Architecture*: a single core of a Sun UltraSPARC T1^{TM} multicore CPU running at 1.20GHz, and an *INTEL Architecture*: Xeon^{TM} CPU 3.00GHz.

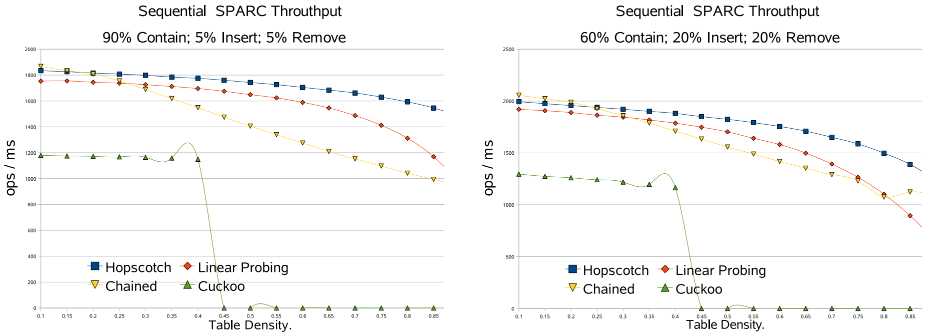

Fig. 4. SPARC throughput as the table density changes.

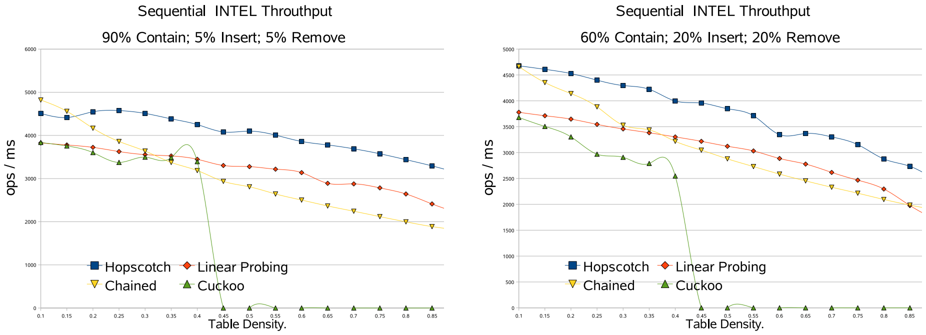

Fig. 5. Intel throughput as the table density changes.

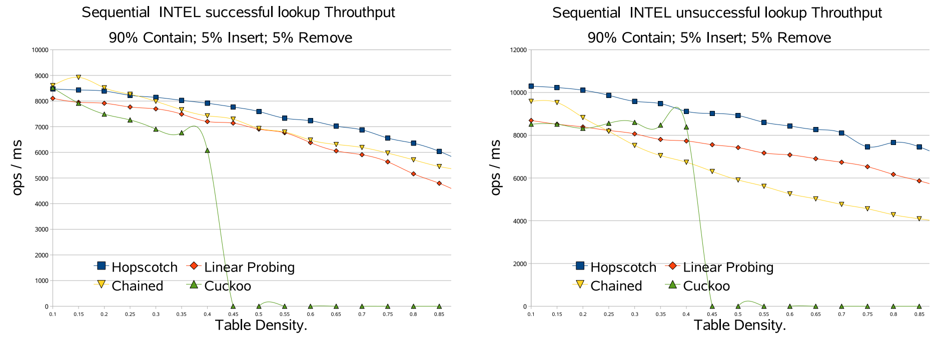

Fig. 6. Intel lookup throughput as the table density changes.

4.6 Sequential Benchmark Results

The following graphs shows change in throughput, as function of the average table-density. In Figure 3 we show the general throughput of a mix of actions. The density range for these tests is from 10 to 90%. As can be observed the new hopscotch-algorithm outperforms the other hash-maps, but the important point is the high level of immunity to increases in table-density. We use neutralized versions of the dynamic memory algorithms by having all their memory pre-allocated. Thus, what is seen is the pure advantage of the Hopscotch approach. The reason for this advantage is the better cache behavior: finding an item takes two cache misses, and this is true even when the table is almost full. This is not the case for linear probing, where as the table fills up, items are located further and further from their original bucket. It is also not true of cuckoo hashing, whose theoretical analysis shows that above 50% table density an insert can not find a valid arrangement that will allow insertion of new keys.

In summary, hopscotch hashing has superior performance in both concurrent and sequential settings, at times quite a significant one. This is due to its better cache behavior.

5. CONCLUSION AND FUTURE WORK

We presented the new Hopscotch algorithm and showed its advantages over all known algorithms, both sequential and concurrent. We note that the resize() method of hopscotch hashing can be parallelized in the same style as in [6], and plan to do so in future work. It would also be interesting to evaluate the performance of the algorithm in real applications, and to run cahce-miss benchmarks. Finally, it would be interesting to formally analyze why hopscotch performs exceptionally well at high table densities. This would amount to showing that even when the table is full, the probability that all hash table buckets that start less than H away from any given bucket i (such as the one found empty) have at least one item that was hashed by h into this bucket (and could therefore be displaced to i).

6. ACKNOWLEDGMENTS

We thank Dave Dice for helping us with the execution of our algorithm on the N2 machine at Sun Labs. And Doug Lea for his guidance and insight.

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] Gonnet, G. H., and Baeza-Yates, R. *Handbook of algorithms and data structures: in Pascal and C (2nd ed.).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

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

[4] 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.

[5] Knuth, D. E. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[6] Lea, D. Hash table `util.concurrent.concurrenthashmap` in *java.util.concurrent* the Java Concurrency Package. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/-src/main/java/util/concurrent/.

[7] Lea, D. Personal communication, Jan. 2003.

[8] Pagh, R., and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[9] Purcell, C., and Harris, T. Non-blocking hashtables with open addressing. *In Lecture Notes in Computer Science, Distributed Computing, Springer Berlin / Heidelberg 3724/2005* (October 2005), 108–121.

[10] Shalev, O., and Shavit, N. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM 53*, 3 (2006), 379–405.

APPENDIX

A. CONCURRENT HOPSCOTCH HASHING

```
template <class _KEY, class _DATA>
class ConcurrentHopscotchHashSet {
private:
   static const int HOP_RANGE = 32;
   static const int ADD_RANGE = 256;
   static const int MAX_SEGMENTS = 1024;
   static const int MAX_TRIES = 2;
   struct Bucket {
      unsigned int volatile _hop_info;
      _KEY*        volatile _key;
      _DATA*       volatile _data;
      unsigned int volatile _lock;
      unsigned int volatile _timestamp;
      void lock();
      void unlock();
   };
   Bucket* segments_ary[MAX_SEGMENTS][ ];
   unsigned int segment_mask;
   unsigned int bucket_mask;
public:
   bool contains(_KEY_* key);
   _DATA* add(_KEY_* key, _DATA* data);
   _DATA* remove(_KEY_* key);
};
```

Fig. 7. ConcurrentHopscotchHashSet and Bucket fields.

Figure 7 shows the fields of the hash table and buckets. As in Lea's algorithm, the segment_array field is a two-dimensional array indexed by *segment* and by *bucket*. Initially, there is only one segment: segment_array[0] is a pointer to an array of buckets whose size is a closest power of two to the user-requested capacity. Each time the table is resized, the number of segments doubles, up to a maximum of MAX_SEGMENTS. The segment_mask and bucket_mask fields are used to map a key's hash value into these indexes.

The bucket hop_info field is a bit-map indicating which of the HOP_RANGE adjacent buckets have data that originally hashed to this bucket. The lock field is used by the add() and remove() methods for mutual exclusion, and the _timestamp field is used by the contains() method to detect concurrent displacements. It must thus be a volatile field.

The add() method (Figure 8) hashes the key, computes segment and bucket indexes, and locks the bucket (Lines 2-6). If the key is present in that bucket, it unlocks and returns it's data. Otherwise, it scans through that segment trying to change one bucket's key from NULL to BUSY (Lines 13-18). If it fails to find empty slot within ADD_RANGE of the starting bucket (Line 14), it will have to resize () the table (Line 32). The ADD_RANGE can be determined dynamically and is application dependent.

Otherwise, it proceeds to use the empty slot. If it succeeds (Line 19), and the empty slot is within HOP_RANGE of the starting bucket (Line 21), then it updates the hop_info and key fields. The update of the hop_info must be before

```
1  _DATA∗ add(_KEY_∗ key, _DATA∗ data) {
2      unsigned int hash      = CalcHashFunc( key−>hashCode() );
3      unsigned int iSegment = hash & segment_mask;
4      unsigned int iBucket   = hash & bucket_mask;
5      Bucket∗   start_bucket  = segments_ary[iSegment][iBucket ];
6      start_bucket −>lock();
7      if  (contains(key))  {
8          _DATA∗ rc = key's data;
9          start_bucket −>unlock();
10         return  rc;
11     }
12     Bucket∗   free_bucket  = start_bucket;
13     int   free_distance =0;
14     for (;   free_distance  < ADD_RANGE; ++free_distance) {
15       if (NULL == free_bucket−>_key && NULL == ATOMIC_CAS(&(free_bucket−>_key), NULL, BUSY)))
16         break;
17       ++free_bucket;
18     }
19     if  ( free_distance   < ADD_RANGE) {
20       do {
21           if  ( free_distance   < HOP_RANGE) {
22              start_bucket −>_hopInfo |= (1 << free_distance);
23              free_bucket −>_data = data;
24              free_bucket −>_key = key;
25              start_bucket −>unlock();
26              return  NULL;
27           }
28            find_closer_free_bucket  (&free_bucket,  &free_distance );
29       } while  (NULL != free_bucket);
30     }
31     start_bucket −>unlock();
32     resize ();  return  add(key,  data);
33  }
```

Fig. 8. The add() method.

that of the key field, so it must be volatile (on some architectures one will need to insert a memory barrier here). It then unlocks the bucket, and returns NULL (Lines 21-26). If the empty slot is not within the HOP_RANGE, it repeatedly calls find_closer_free_bucket () to displace the empty slot even closer to the starting bucket (Line 28).

Figure 9 shows the find() method. It takes two arguments, both indirect pointers used to return results. On entry, free_bucket is an indirect pointer to the newly-emptied bucket, and on exit, it either points to a newly-emptied bucket closer to the starting bucket, or it is NULL, if no such bucket could be found. Similarly, on entry, free_distance points to the free bucket's distance from the starting bucket, and on exit, to the newly-emptied bucket's distance from the starting bucket.

Starting at the bucket HOP_RANGE-1 slots below the empty bucket, the method examines each bucket in ascending order until it reaches the empty bucket (Lines 2-35). For each bucket, it iterates through its hop_info bit map, from lowest index to highest, looking for an occupied bucket whose key can be moved to the empty bucket (Lines 7-12). If it finds one, it moves the bucket's key, updates its timestamp to alert concurrent contains() calls of the update, and returns (Lines 13-32). If it

```
void find_closer_free_bucket (Bucket** free_bucket, int* free_distance ) {
    Bucket* move_bucket = *free_bucket - (HOP_RANGE - 1);
    for (int free_dist = (HOP_RANGE - 1); free_dist > 0; --free_dist) {
      unsigned int start_hop_info = move_bucket->hop_info;
      int move_free_distance = -1;
      unsigned int mask = 1;
      for (int i=0; i< free_dist ; ++i, mask <<= 1) {
        if (mask & start_hop_info) {
          move_free_distance = i;
          break;
        }
      }
      if (-1 != move_free_distance) {
        move_bucket->lock();
        if ( start_hop_info == move_bucket->hop_info) {
          Bucket* new_free_bucket = move_bucket + move_free_distance;

          move_bucket->hop_info |= (1 << free_dist);
          free_bucket ->_data = new_free_bucket->_data;
          free_bucket ->_key = new_free_bucket->_key;

          ++(move_bucket->_timestamp);

          new_free_bucket->_key = BUSY;
          new_free_bucket->_data = BUSY;
          move_bucket->hop_info &= ~(1 << move_free_distance);

          *free_bucket = new_free_bucket;
          * free_distance -= free_dist;
          move_bucket->unlock();
          return;
        }
        move_bucket->unlock();
      }
      ++move_bucket;
    }
    *free_bucket = NULL; *free_distance = 0;
}
```

Fig. 9. The find_closer_free_bucket () method.

is unable to find a bucket to swap, it sets free_bucket to NULL and returns.

The contains() method (Figure 10) is wait-free. In the *fast path* section, which we expect to be common, it tries MAX_TRIES times to scan through the key's range without overlapping an add() that displaces the key (Lines 8-16). Concurrent add() and remove() calls that do not displace an existing key will not cause the contains() method to abandon the fast path.

Each scan starts by reading the lock's timestamp to ensure that it views a consistent state of the hop_info field (Line 9). It then compares each slot marked by hop_info to the target key, and computes a tentative return value. If it found the key it returns. If it did not find the key, then it rereads the lock's timestamp (Line 17). If the timestamp is unchanged, then it observed a consistent state, and returns the tentative value.

If, after MAX_TRIES attempts, the contains() method fails to observe a consistent

```
1  bool contains(_KEY_* key) {
2      unsigned int hash      = CalcHashFunc( key->hashCode() );
3      unsigned int iSegment = hash & segment_mask;
4      unsigned int iBucket   = hash & bucket_mask;
5      Bucket*   start_bucket   = segments_ary[iSegment][iBucket];
6      unsigned int  try_counter  = 0;
7      unsigned int  timestamp;
8      do {
9       timestamp = start_bucket->_timestamp;
10      unsigned int  hop_info  = start_bucket->_hop_info;
11      for each check_bucket  in  hop_info  {
12         if  (key.equal(check_bucket->_key))
13           return  true;
14      }
15      ++try_counter;
16     } while  (timestamp != start_bucket->_timestamp && try_counter < MAX_TRIES)
17     if(timestamp != start_bucket->_timestamp ) {
18      Bucket* check_bucket = start_bucket;
19      for  (int  i=0; i  < HOP_RANGE; ++i) {
20         if  (key.equal(check_bucket->_key))
21              return  true;
22         ++check_bucket;
23      }
24      }
25     return  false;
26 }
```

Fig. 10. The contains() method.

state, then the method takes the *slow path*, which we expect to be rare (Lines17-25). It simply scans all buckets within HOP_RANGE, ignoring the hop_info field, again without locking.

The remove() method (Figure 11) locates the starting bucket and locks it (Lines 2-6). It then scans through the buckets marked by the hop_info field, checking for the target key (Lines 8-18). If it finds the key, it sets that bucket's key field to NULL, unlocks the bucket. To avoid contamination [2] as in linear probing, it updates the hop_info field. Unlike with the add() method, the order in which the hop_info field and key field are updated does not matter. It then returns the previous data. Otherwise it unlocks the bucket and returns NULL.

We postpone the description of the resize() method to the full version of the paper.

B. CORRECTNESS OF THE CONCURRENT HOPSCOTCH ALGORITHM

This section contains a an outline of the proof that our concurrent hopscotch algorithm has the desired properties of a resizable hash table.

Notice that the complexity analysis presented earlier holds for the concurrent case, where the term *expected time* in the concurrent case means the expected number of machine instructions in the worst case scheduling scenario, assuming a hash function of uniform distribution. Notice that the theorems presented earlier are true as is, except during a concurrent resize(), which could take $O(m)$ time. Moreover, we need to notice that the proof of Lemma 3.5 holds in the concurrent case since overlapping add() calls cannot interfere with the displacements of one

```
_DATA* remove(_KEY_* key) {
  unsigned int hash      = CalcHashFunc( key->hashCode() );
  unsigned int iSegment = hash & segment_mask;
  unsigned int iBucket  = hash & bucket_mask;
  Bucket*  start_bucket  = segments_ary[iSegment][iBucket];
  start_bucket ->lock();

  unsigned int hop_info = start_bucket->hop_info;
  for each check_indx in hop_info {
    Bucket* check_bucket = start_bucket + check_indx;
    if (key.equal(check_bucket->_key)) {
      _DATA* rc = check_bucket->_data;
      check_bucket->_key = NULL;
      check_bucket->_data = NULL;
      start_bucket ->_hop_info &= ~( 1 << check_indx );
      start_bucket ->unlock();
      return rc;
    }
  }
  start_bucket ->unlock();
  return NULL;
}
```

Fig. 11. The remove() method.

another, so the overall expected time for an add() is constant as shown.

We now proceed to prove safety and liveness properties of the algorithm. Our model of multiprocessor computation follows [4], 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.

B.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]. The add() method returns true if the key was successfully inserted to the set, and false if that key already existed in the set. The contains() operation returns true if the key is in the set, false otherwise. The remove() operation returns true if the key was successfully deleted from the set and false if it was not found. A resize () operation should have no effect on the set but its side effect is that the maximal cardinality of the set is doubled.

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.

Unsuccessful add() and remove() methods are linearized respectively at the points where their internal contains() method calls are successful in finding the key (for an add()) or unsuccessful (for a remove()). A successful add() is linearized when it writes the key to the table. (Notice that before this successful writing of the key the table may go through a resize). A successful remove() is linearized when it removes (overwrites) the actual key from the table entry. A successful contains() is linearized when it finds the searched key in the array entry. An unsuccessful contains() can occur in one of two ways. One possibility is when it does not find the searched key in the hop information locations and later finds the timestamp in the entry's lock has not changed. The other possibility is that it failed to see the timestamp unchanged several times, following which it made a pass over all 32 possible locations to its right and did not find the searched key. In both cases we can linearize to the last of these operations, unless there was a write of the searched key by some concurrent add() method call, in which case the linearization point is immediately before the last overlapping write of this key by an add().

The proof of most of these cases is straightforward because add() and remove() are mutually exclusive on the the modified location's. Moreover, the add() first updates the hop_info and only then writes the data and key. This means that if a contains() call finds the added key, any subsequent contains() call will also find it, and they can all be linearized after the add(). It also means that if a contains() did not find the key based on the hop_info, then if the key exists it is because of a concurrent add(), and the contains() can correctly be linearized before it. The remove() call removes the key, and independently of whether the hop info points to its old entry or not, the contains() will fail once the key is removed.

The interesting case is when an add() dispalces an item concurrently with a contains() call. The add() calls first copies the key to be moved from its old entry i to its new entry j, then increments the timestamp, and only then erases the key from entry i. Thus, if the contains() fails to read the key it can only be because it used the old hop_info to look in the entry i, and then failed to find the key in entry i. However, this means that the timestamp field must have been incremented between the reading of the old hop_info and the reading of the empty entry i, and the fast path will fail. The slow path of the contains() is immune to displacements since these happen from left to right, and the slow path searches from left to right, so if it missed the key in entry i it will find it in entry j.

B.2 Progress

Our algorithm uses loads, stores, and compareAndSet() operations to acquire locks. As we will show, in terms of these primitive operations the algorithm's contains() method is wait-free, that is, each thread always completes in a finite number of operations. Its add(), remove() and resize () use locks but are deadlock free.

The contains() method is wait-free by definition since its code contains no unbounded loops. Each array entry has an associated lock that protects the hop information, and also has a kind of full-empty bit defined by the hash field of free_bucket. There are many entries that could potentially have the given entry in their hop_info, but, as we prove, there can only be one that actually has it at any given point. To remove() an item, only one lock, the lock of the location with the hop information is acquired. To add() and item, one lock is taken and then to displace an item, while the lock is held, the full-empty hash field of free_bucket of the empty location is set (no thread ever spins on this full-empty information, so it cannot be a source of deadlock) and then a key and its data are moved. This requires acquiring a second lock, but the second lock is always to the right of the first, and since all add() calls acquire the locks in this way, holding first the leftmost and then a lock to its right, there cannot be a deadlock.

The algorithm is however not starvation-free: it trusts the natural distribution of hashed values to prevent starvation, and if this distribution is unfair, one can replace the simple locks we use with more expensive fair locks.