Hopscotch Hashing

Maurice Herlihy[1], Nir Shavit[2,3], and Moran Tzafrir[3]

[1] Brown University, Providence, RI
[2] Sun Microsystems, Burlington, MA
[3] Tel-Aviv University, Tel-Aviv, Israel

Abstract. We present a new class of resizable sequential and concurrent hash map algorithms directed at both uni-processor and multicore machines. The new hopscotch algorithms are 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 algorithms provide tables 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 hopscotch 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 hopscotch consistently outperforms the most effective sequential hash table algorithms including cuckoo hashing and bounded linear probing.
The most interesting feature of the new class of hopscotch algorithms is that they continue to deliver good performance when the hash table is more than 90% full, increasing their advantage over other algorithms as the table density grows.

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 [2], and so optimizing this operation has been a target of many hash table algorithms.

This paper introduces *hopscotch hashing*, a new class of open addressed resizable hash tables that are directed at today's cache-sensitive machines.

1.1 Background

Chained hashing [3] 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 [4]. 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 [3] 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 [3]), and clustering of keys may cause a large variance in performance. After a period of use, a phenomenon called *contamination* [5], caused by deleted items, degrades the efficiency of unsuccessful contains() calls.

Cuckoo hashing [4] 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 [7] present another high-performance lock-free closed address resizable scheme. Purcell and Harris [8] 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 [9].

2 The Hopscotch Hashing Approach

Hopscotch hashing algorithms are open addressed algorithms that combine elements of cuckoo hashing, linear probing, and chaining, in a novel way. Let us begin by describing a simple variation of the hopscotch approach, later to be refined as we present our actual implementations.

The hash table is an array of buckets. The key characteristic of the hopscotch approach is the notion of a *neighborhood* of buckets around any given items

bucket. This neighborhood has the property that the cost of finding the desired item in any of the buckets in the neighborhood is the same or very close to the cost of finding it in the bucket itself. Hopscotch hashing algorithms will then attempt to bring an item into its neighborhood, possibly at the cost of displacing other items.

Here is how our simple algorithm works. 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 $H-1$ neighboring entries, where H is a constant (H could for example be 32, the standard machine word size). In other words, the neighborhood is a "virtual" bucket that 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 5). 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.

The hopscotch neighborhood is a virtual bucket with multiple items, but unlike in chaining these items have great locality since typically items can be located in adjacent memory locations and even in the same cache lines. Items are inserted in constant expected time as in linear probing, but without fear of clustering and contamination.

Finally, notice that in our simple algorithm, if more than a constant number of items are hashed by h into a given bucket, the table needs to be resized. Luckily, for a universal hash function h, the probability of this type of resize

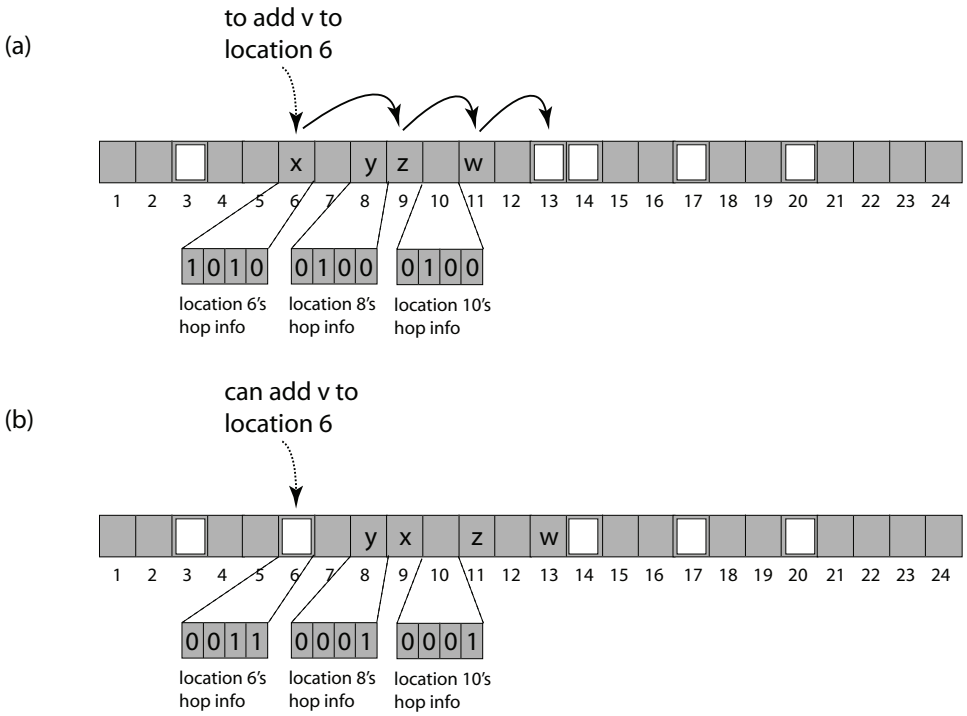

Fig. 1. The blank entries are empty, all others contain items. Here, H is 4. In part (a), we add item v with hash value 6. A linear probe finds entry 13 is empty. Because 13 is more than 4 entries away from 6, we look for an earlier entry to swap with 13. The first place to look is $H - 1 = 3$ entries 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.

happening for a given H is $\Theta(1/H!)$. [4] Figure 1 shows an example execution of the simple hopscotch algorithm.

The concurrent version of the simple 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 obstruction-free [10], while the add() and remove() acquire the bucket lock before applying modifications to the data.

In the next section we describe in detail a cache aware version of the hopscotch approach.

[4] In our implementation, as we describe in the next section, we can actually use a pointer scheme in the *hop-information* word instead of the easier to explain bitmap, and so buckets can actually grow H dynamically.

3 A Hopscotch Hashing Algorithm

```
1  template <typename _tKey, typename _tData, typename _tHash, typename _tEqual,
2            _tKey _empty_key, _tData _empty_data, typename _tLock>
3  class ConcurrentHopscotchHashMap {
4    static const short  _null_delta  = SHRT_MIN;
5    //inner classes
6    struct Bucket{short volatile  _first_delta ; short  volatile  _next_delta ;
7                  _tKey volatile  _key;          _tData volatile  _data ;};
8    struct Segment {_tLock       _lock ;      Bucket*      _table ;
9                    int volatile _timestamp; int volatile _count;
10                   Bucket*      _last_bucket ;};
11   // fields
12   const int       _segment_shift ;
13   const int       _segment_mask;
14   Segment* const _segments;
15   int             _bucket_mask;
16   const int       _cache_mask;
17   const bool       _is_cacheline_alignment ;
18   cont int        _hop_range;
19 public:
20   ConcurrentHopscotchHashMap(int initial_capacity , int  concurrency_level ,
21                   int  cache_line_size , bool  is_optimize_cacheline , int hop_range);
22   virtual ~ConcurrentHopscotchHashMap();
23   virtual bool contains(const _tKey key);
24   virtual _tData get(const _tKey key);
25   virtual _tData add(const _tKey key, const _tData data);
26   virtual _tData remove(const _tKey key);
27 };
```

Fig. 2. The hopscotch class.

We now present the more complex algorithm in the hopscotch class, the one used in our benchmarked implementations. The high level ideas behind this algorithm, in both the sequential and concurrent case, are as follows.

The table consists of segments of buckets, and in the concurrent case each segment is protected by a lock in a manner similar to that used in [6]. Instead of a bitmap representation of the *hop-information* word. We represent keys in each "virtual" bucket as a linked list, starting at the key's original bucket, and distributed throughout other buckets reachable via a sequence of pointers from the original one. Each pointer can point a distance *hop_range* to the next item.

The add() method in Figure 3 first tries to add it to some free bucket in its original bucket's cache line (line 8). If that fails, it probes linearly to find an empty slot within hop range of the key's original bucket, and then points to it from the last bucket in key's virtual bucket, that is, the last bucket in the list

```
1  virtual _tData add(const _tKey key, const _tData data) {
2    const int      hash          (_tHash::Calc(key));
3    Segment&       segment       (getSegment(hash));
4    Bucket* const start_bucket(getBucket(segment, hash));
5    segment._lock.aquire();
6    if(contains(key)) {segment._lock.release(); return keys data;}
7    Bucket* free_bucket=freeBucketInCacheline(segment, start_bucket);
8    if(null != free_bucket) {
9      addKeyBeginingList(start_bucket, free_bucket, hash, key, data);
10     ++(segment._count);
11     segment._lock.release();
12     return null;
13   }
14   free_bucket=nearestFreeBucket(segment, start_bucket);
15   if(null != free_bucket) {
16     int free_dist =(free_bucket - start_bucket);
17     if(free_dist > _hop_range || free_dist < -_hop_range)
18     free_bucket=displaceFreeRange(segment, start_bucket, free_bucket);
19     if(null != free_bucket) {
20       addKeyEndList(start_bucket, free_max_bucket, hash, key, data, last_bucket);
21       ++(segment._count);
22       segment._lock.release();
23       return null;
24     }
25   }
26   segment._lock.release();
27   resize();
28   return add(key, data);
29 }
```

Fig. 3. add() pseudo-code.

of buckets originating in the key's original bucket. If it could not find such an empty bucket (line 18), it applies a sequence of hopscotch displacements, trying to displace the free-bucket to reside within hop_range of the original bucket of the key. If such a sequence of displacements cannot be performed then the table is resized.

The remove() method in Figure 4 removes a key and tries to optimize the cache line alignment of keys belonging to the neighborhood of the emptied bucket. The idea is to find a key that will, once moved into the emptied bucket, reside in its associated bucket's cache line, thus improving cache performance. In the concurrent version of remove(), while traversing the bucket's list, the method locks segments as it proceeds. The remove() operations will modify the _timestamp field, since concurrent contains() calls need to be failed by concurrent removes.

```
1  virtual _tData remove(const _tKey key) {
2    const int      hash          (_tHash::Calc(key));
3    Segment&       segment       (getSegment(hash));
4    Bucket* const  start_bucket  (getBucket(segment,hash));
5    Bucket*        last_bucket   ( null );
6    Bucket*        curr_bucket   ( start_bucket );
7    segment._lock.aquire ();
8    short  next_delta (curr_bucket-> _first_delta );
9    do {
10     if( _null_delta ==next_delta) {segment._lock.release (); return null ;}
11
12     curr_bucket  += next_delta;
13     if(_tEqual::IsEqual(key,  curr_bucket->_key)) {
14       _tData const found_key_data(curr_bucket->_data);
15       removeKey(segment, start_bucket,  curr_bucket ,  last_bucket ,  hash);
16       if( _is_cacheline_alignment ) cacheLineAlignment(segment, curr_bucket);
17       segment._lock.release ();
18       return found_key_data;
19     }
20     last_bucket  = curr_bucket;
21     next_delta  = curr_bucket->_next_delta;
22   } while(true);
23   return null ;
24 }
```

Fig. 4. remove() pseudo-code.

Notice that it may be the case that once a key x is removed, the preceding key pointing to the x in its associated bucket's list, may find that the item following x in the bucket's list is outside its *hop_range*. In this case, the remove will find the last item y in the list belonging to x's bucket and displace it to x's empty bucket. The new empty bucket of the removal of x will thus be the bucket of the displaced y.

To optimize the key's cache-line alignment (line 16), for each of the buckets in the cache line of the empty bucket, the method traverses the associated lists and moves the last item in the list into the empty bucket. This process can be performed recursively.

The contains() method appears in Figure 5. It traverses the buckets key list, and if a key is not found (line 15) there are two possible cases: (i) the key does not exist or (ii) a concurrent remove() (detected via a changed _timestamp in the associated segment) caused the contains() to miss the key, in which case the method looks for the key again.

```
1  virtual bool contains(const _tKey key) {
2    const int hash          (_tHash::Calc(key));
3    Segment& segment        (getSegment(hash));
4    Bucket*  curr_bucket    (getBucket(segment, hash));
5    int start_timestamp;
6    do {
7      start_timestamp = segment._timestamp[hash & _timestamp_mask];
8      short next_delta(curr_bucket->_first_delta);
9      while (_null_delta != next_delta) {
10       curr_bucket += next_delta;
11       if (_tEqual::IsEqual(key, curr_bucket->_key))
12         return true;
13       next_delta = curr_bucket->_next_delta;
14     }
15   } while(start_timestamp != segment._timestamp[hash & _timestamp_mask]);
16   return false;
17 }
```

Fig. 5. contains() pseudo-code.

4 Analysis

Unsuccessful add() and remove() methods are linearized respectively at the points where their internal contains() method calls are successful in finding the key (for add()) or unsuccessful (for remove()). A successful add() is linearized when it adds the new bucket to the list of buckets that hashed to the same place, either updating by _next_delta or _first_delta , depending on whether the bucket is in the cache line. A successful remove() is linearized when it overwrites the key from the table entry. A successful contains() is linearized when it finds the desired key in the array entry. An unsuccessful contains() is linearized when it reached the end of the list, and found the timestamp unchanged.

The add() and remove() methods use lockqs. They are deadlock-free but not livelock-free. The contains() method is obstruction-free.

We next analyze the complexity of the sequential and concurrent versions of the hopscotch algorithm. The most important property of a hash table is its expected constant time performance. We will 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, the constant H is the maximal number of keys a bucket can hold, n is the number of keys in the table, m is the table size, and $\alpha = n/m$ is the density or *load factor* of the table.

Lemma 1. *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 [3].

In hopscotch hashing, as in chained hashing, in the common case there are very few items in a bucket.

Lemma 2. *The maximal expected number of items in a bucket is constant.*

Proof. Again, following Knuth [3], the function $f(m, n)$ is increasing, and the maximal value for n, the number of items, is m, so that the function's value tends 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.

Lemma 3. *Calls to* remove() *and* contains() *complete in constant expected time when running in isolation.*

Proof. A complete proof will appear in the final paper. Suppose that there r threads calling remove(), a threads calling add(), and c threads calling contains(). Since the number of segments n is at least the number of threads, $(a+c+r) <= n$. The probability of having at least one segment with remove() and contains() assigned to it is $n\left(\frac{n^{c-1}n^a n^{r-1}}{n^n}\right) = \frac{1}{n}$. By Lemma 2, each iteration in contains() takes at most a constant expected number of steps. The expected number of iteration is:

$$
\begin{aligned}
1 + \sum \frac{i}{n^i} = \frac{\frac{k(1+k)}{2}}{\frac{1-n^k}{1-n}} &= 1 + \frac{k(1+k)(1-n)}{2(1-n^k)} \\
&= 1 + \frac{k(1+k)(1-n)}{2(1-n)\sum 1^t n^{k-1-t}} \\
&= 1 + \frac{k(1+k)}{2\sum 1^t n^{k-1-t}} \\
&\to 1
\end{aligned}
$$

It is known that the worst case number of items in a bucket, even when using a universal hash function, is not constant [3]. So what are the chances of a resize () due to overflow? It turns out that chances are low.

Lemma 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 [5] it follows that

$$\begin{aligned}\Pr\{\text{H items in bucket}\} &= \binom{n}{H}\frac{(m-1)^{n-H}}{m^n} \\ &= \frac{n!}{H!(n-H)!}\frac{(m-1)^{n-H}}{m^n} \\ &= \frac{1}{H!}n(n-1)\dots(n-H+1)\frac{(m-1)^{n-H}}{m^{n-H}m^H} \\ &\leq \frac{1}{H!}\end{aligned}$$

The last inequality follows by substituting m for n, where m is a monotonically increasing function's maximal value.

With uniform hashing, the probability of resizing due to having more than $H = 32$ items in a bucket is $1/32!$.

Lemma 5. *The probability of probing for an empty slot, with length bigger then H is α^{H-1}.*

The result follows from Theorem 11.6 of [1]. For $\alpha = 0.8$, $Pr(H = 32) \approx 0.000990$, $Pr(H = 64) \approx 7.846377 \cdot 10^{-7}$, and for $\alpha = 0.9$ $Pr(H = 128) \approx 1.390084 \cdot 10^{-6}$, $Pr(H = 256) \approx 7.846377 \cdot 10^{-7}$. This is true for universal hash functions. The experimental results show that using the *displacement-algorithm* the probability of failing to find an empty slot in range H is smaller. For example, for $\alpha = 0.8$ and $H = 32$, no resize occurred. The analysis of the probably to fail to find an empty slot in range H, when using the displacement algorithm is left for the full paper.

Lemma 6. *Calls to* `add`() *complete within expected constant time.*

Proof. From Knuth [3] for 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 $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}^2\right)$ 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 7. *Calls to* `resize`() *complete within $O(n)$ time.*

5 Performance Evaluation

This section compares the performance of hopscotch hashing to the most effective prior algorithms in both concurrent (multicore) and sequential (uniprocessor) settings. We use micro-benchmarks similar to those used by recent papers in the area [4, 7], but with significantly higher table densities (up to 99%).

We sampled each test point 10 times, and plotted the average. To make sure that the table does not fit into a cache-line, we use a table size of approximately 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 ensure that memory-management costs do not skew our numbers, we show results for the closed-address hash-maps both with the mtmalloc multi-threaded malloc library, and with pre-allocated memory.

In all algorithms, each bucket encompasses pointers to the key and data (satellite key and data). This scheme is thus a general hash-map.

5.1 Concurrent Hash-Maps on Multicores

We compared, on a 64-way Sun UltraSPARC^{TM} T2 multicore machine, two versions of the concurrent hopscotch hashing to the *Lock-based Chained* algorithm of Lea [6]: *Chained_PRE* is the pre-allocated memory version, and *Chained_MTM* is the mtmalloc library version, and the *New Hopscotch* algorithm: a variant of hopscotch that uses a an 16bit representation of the pointers in the *hop-information* word, providing an effectively unbounded range of 32767 locations. To neutralize any effects of the choice of locks, all lock-based concurrent algorithms use the same locks, one per memory segment.The number of segments is the concurrency level (number of threads). We also use the same hash function for all the algorithms (except *Cuckoo-Hashing*).

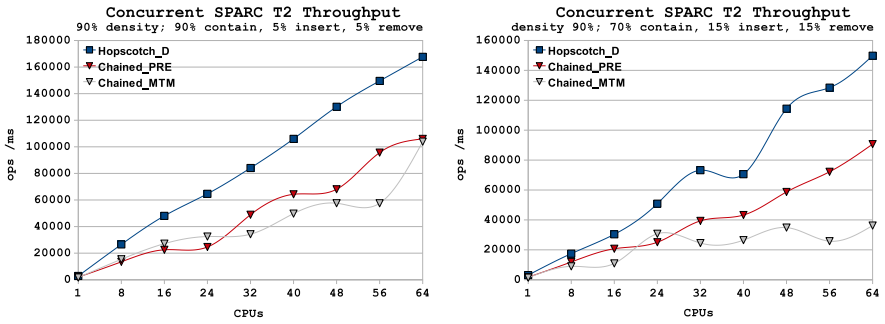

Fig. 6. Concurrent performance under high-loads.

Figure 6 illustrates how the algorithms scale under high loads and a table density of 90%. As can be seen, hopscotch is about twice as fast as *Chained_PRE* and more than three times as fast as *Chained_MTM*.

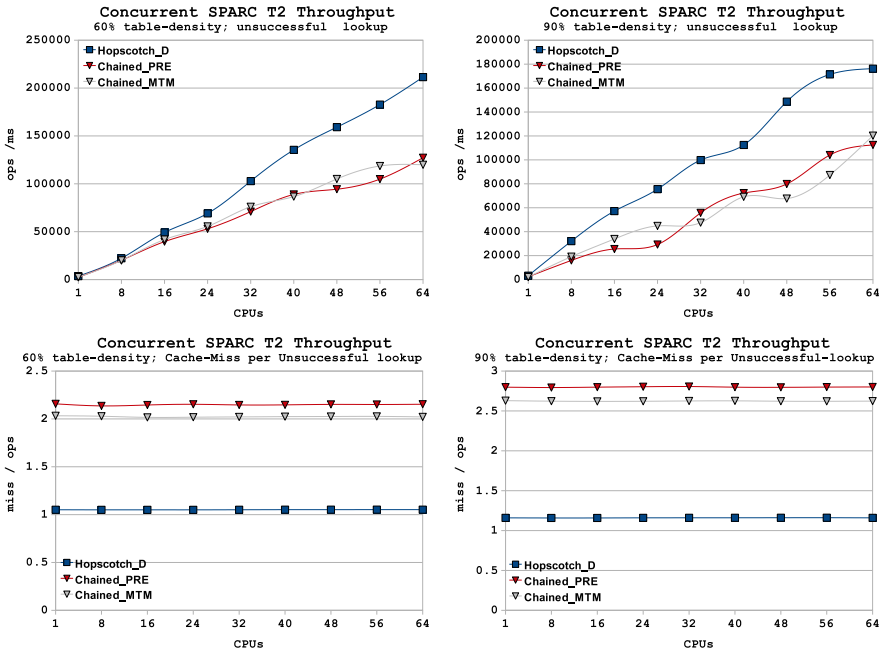

Fig. 7. Concurrent Benchmark: unsuccessful contains() throughput and cache-miss counts.

Figure 7 compares the performance of the hash table's unsuccessful contains() calls which are more trying for all algorithms than successful ones. As can be seen, the hopscotch algorithm retains its advantage when only contains() operations are counted, implying that the difference in performance among the various algorithms is not due to any effect of the locking policy. The lower graphs explain the results: hopscotch suffers fewer cache misses per operation than the other algorithms. The benefit of using an open addressed table is further exemplified by the gap it opens from the chained algorithm once memory allocation costs are not discounted. Though we do not show it, the advantage in performance continues even as the table density reaches 99%, though with a smaller gap.

5.2 Sequential Hash-Maps

We selected the most effective known sequential hash-maps.

– *Linear-Probing*: we used an optimized version [3], that stores the maximal probe length in each bucket.
– *Chained*: We used an optimized version of [3] that stores the first element of each linked-list directly in the hash table.
– *Cuckoo*: Thanks to the kindness of the authors of [4], we obtained the original cuckoo hash map code.
– *New Hopscotch*: we used the sequential version. *Hopscotch_D* is a variant of hopscotch algorithm, that displaces keys to improve cache-line alignment. We contrast it with *Hopscotch_ND* that does not perform displacements to allow cache-line alignment.

We ran a series of benchmarks on two different uniprocessor architectures. A single core of a Sun UltraSPARC^{TM} T1 running at 1.20GHz, and an Intel^{R} Core^{TM} Duo CPU 3.50GHz.

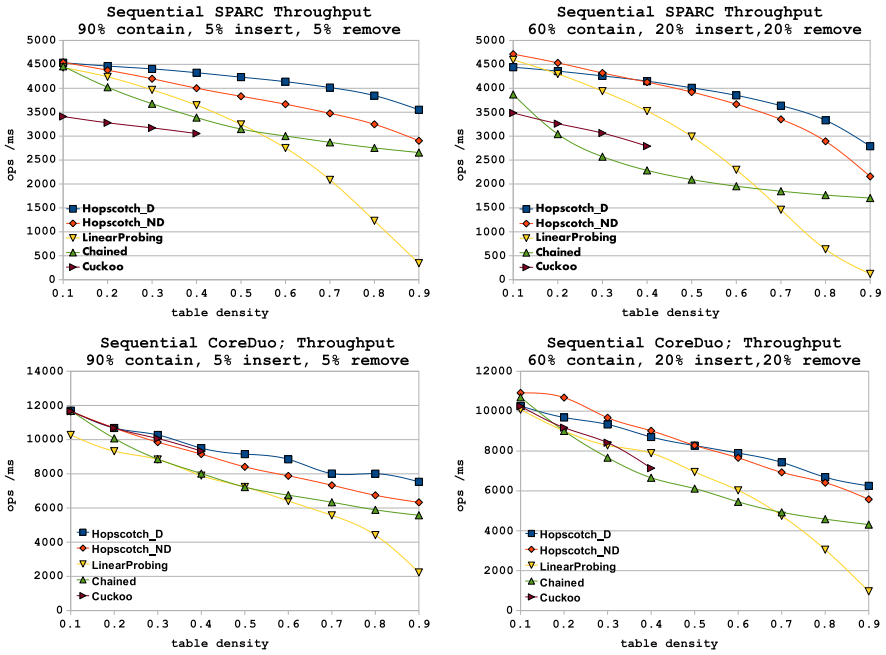

Fig. 8. Sequential high-load benchmark.

The graphs in this section show throughput as function of the average table density. Tests cover high loads, extremely high table density, and contains() throughput. First, we contaminated the tables (e.g ran a long sequence of add() and calls). This preparation is important for a realistic analysis of open-address hash tables such as hopscotch or linear probing [3].

As can be seen, the performance of all the hash-maps diminishes with density. Nevertheless, the results show that the overhead associated with hopscotch's key-cache alignment and displacement does not diminish relative performance at high loads. As in the concurrent case, to ensure that memory allocation overhead does not distort the results, we use neutralized versions of the dynamic memory algorithms where all memory needed is preallocated.

Figure 9 shows the extremely high table density benchmark. As table density increases, so does the average number of keys per bucket, implying both longer CPU time for all hash-map operations, and the likelihood of more cache misses. As expected, cuckoo hashing does not carry beyond 50% table densities due to cycles in the displacement sequences. The open-addressed linear-probing does poorly compared to hopscotch. Note that the Hopscotch_D overhead of ensuring key's cache-line alignment and displacement proved beneficial even in this extreme setting.

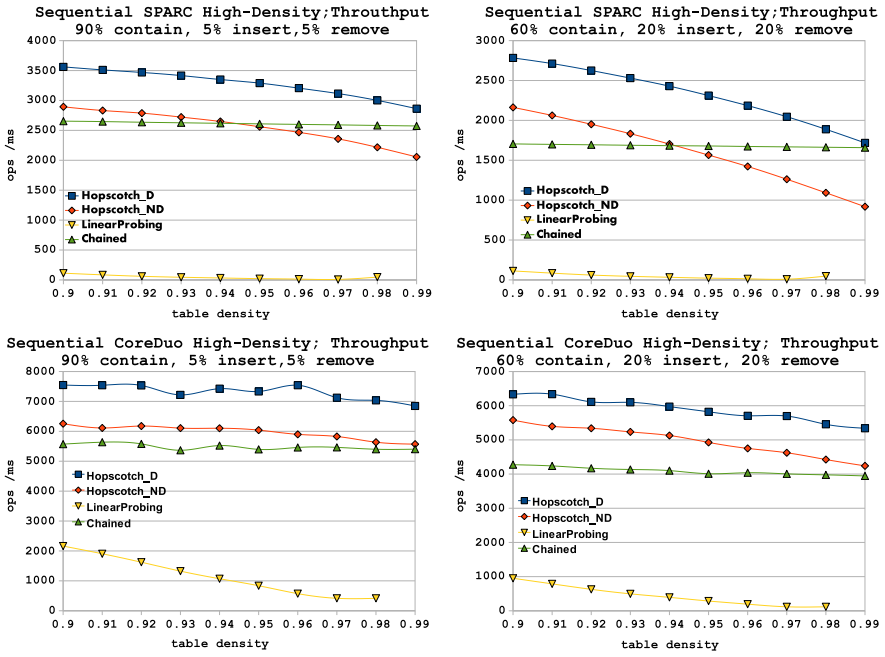

Fig. 9. Sequential extremely high table density benchmark under 60% load.

Figure 10 shows the throughput of the unsuccessful contains() method calls, a measure dominated by memory access times. The lower graphs show how performance is completely correlated with cache miss rates and how the cache-line alignment of the hopscotch algorithm serves to allow it to dominate the other

algorithms. Perhaps quite amazingly, hopscotch delivers less than one cache miss on average on the Intel architecture!

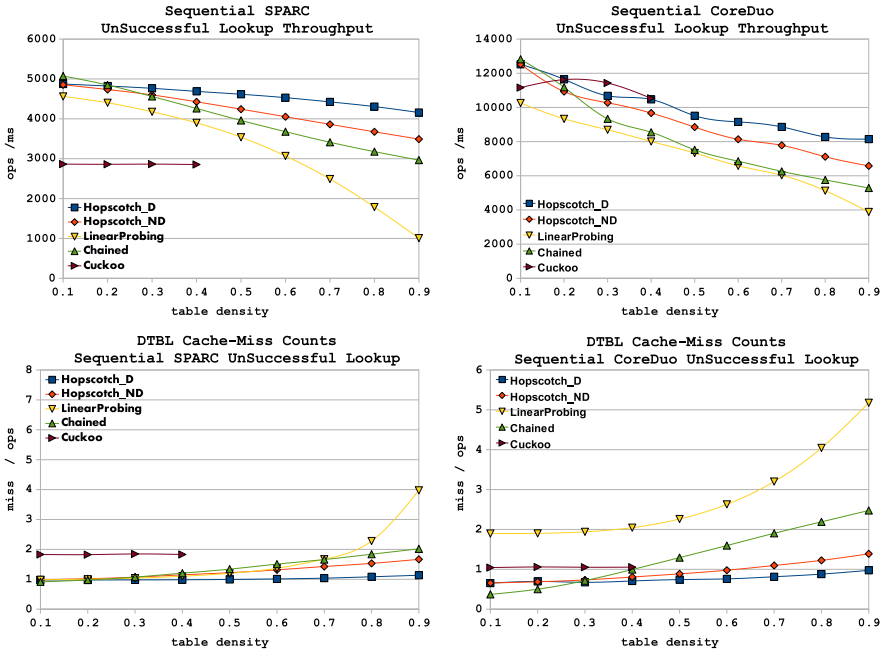

Fig. 10. Sequential contains() throughput with related cache miss counts below.

6 Acknowledgments

We thank Dave Dice and Doug Lea for their help throughout the writing of this paper.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. MIT Press, Cambridge, Massachusetts (2001)
2. Lea, D.: Personal communication (January 2003)
3. 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)
4. Pagh, R., Rodler, F.F.: Cuckoo hashing. Journal of Algorithms **51**(2) (2004) 122–144

5. Gonnet, G.H., 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)
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. Shalev, O., Shavit, N.: Split-ordered lists: Lock-free extensible hash tables. Journal of the ACM **53**(3) (2006) 379–405
8. Purcell, C., 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
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, NY, USA (2008)
10. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2003) 522