



Hopscotch Hashing

A Hash Map Algorithm for Years to Come

Extreme Scalability at Very High Table Densities

Maurice Herlihy, Nir Shavit, and Moran Tzafrir

Tel-Aviv University & Brown University



Introduction

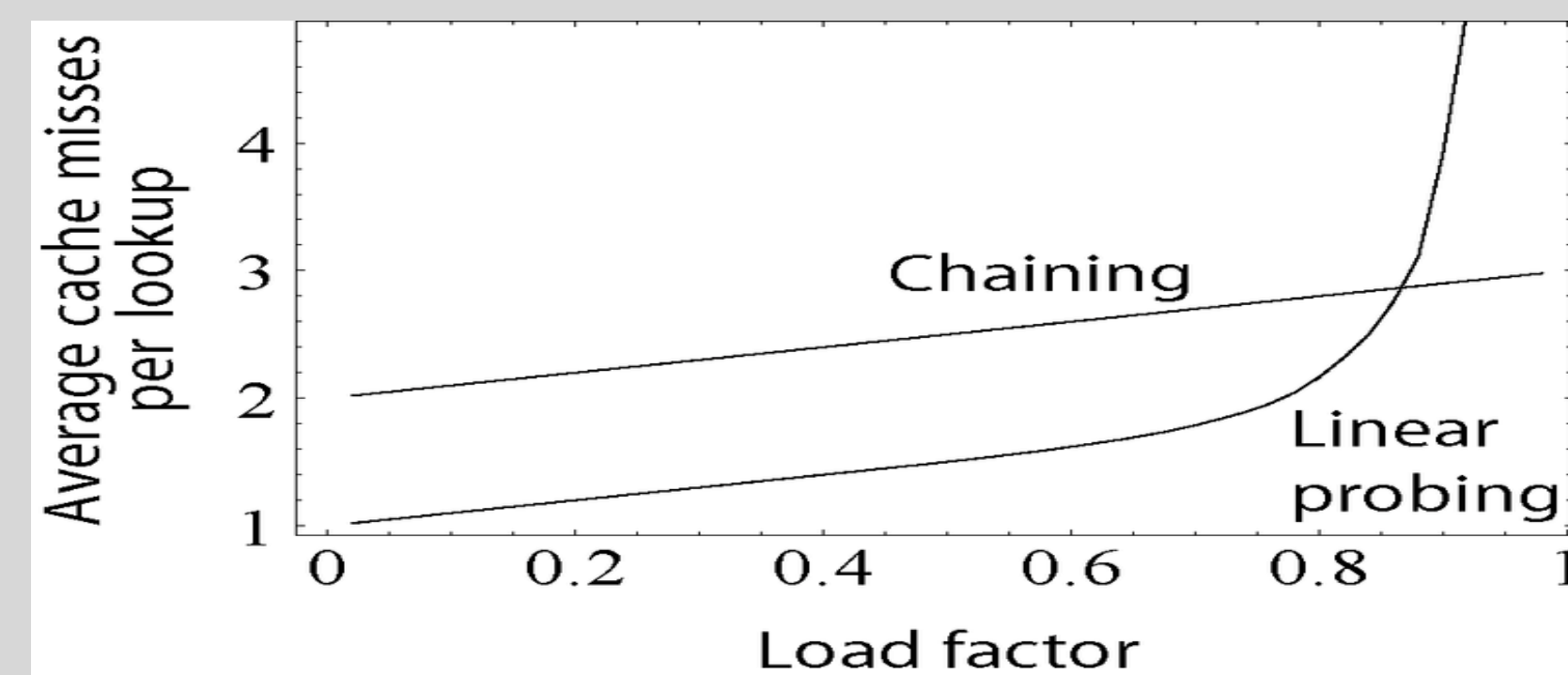
We present a new resizable **sequential & concurrent open-addressed hash-map** algorithm, utilizing our novel **hopscotch** multi-phased *probing* and *displacement* technique. The new algorithm offers a unique combination of:

1. more than **90%** memory utilization,
2. almost **linear** scalability,
3. **deterministic constant** lookup-time, and
4. a **high level of immunity** to increases in *table-density*, i.e. *good performance when the table density is above 90%*.

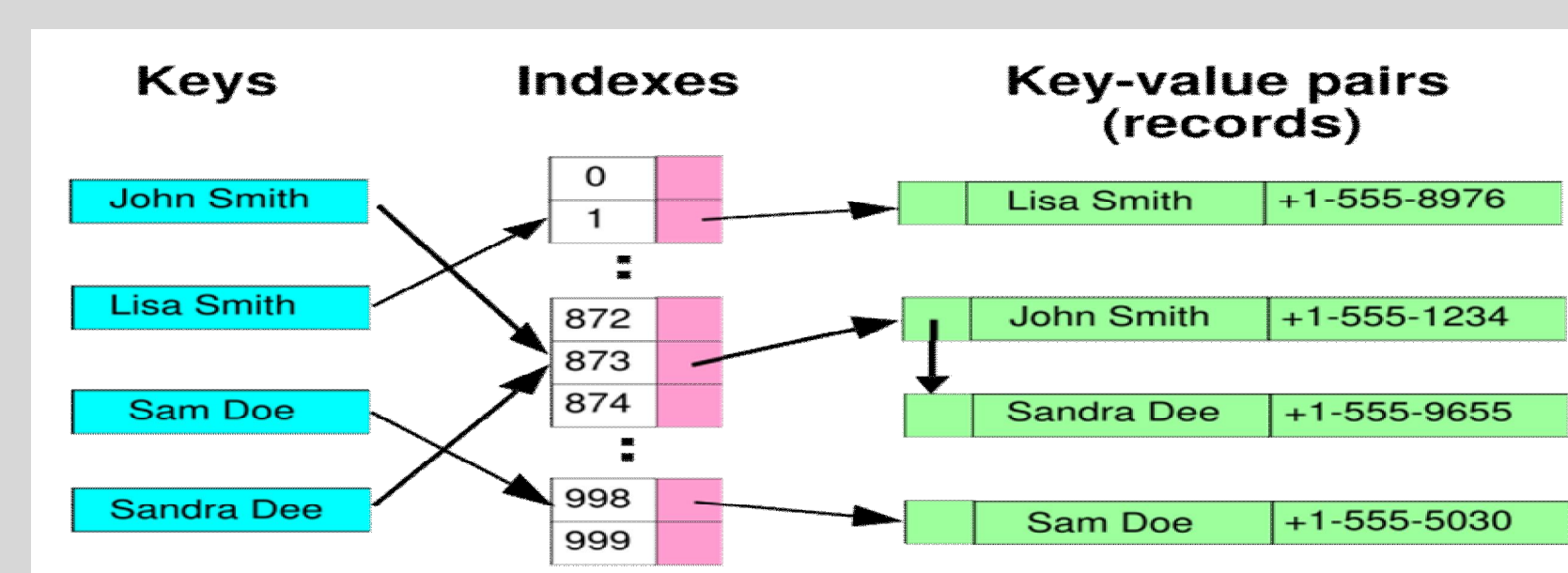
To the best of our knowledge, no prior algorithm has all these properties, and indeed, hopscotch hashing outperforms all known hash table algorithms on both uni- and multi-core machines.

Methodology

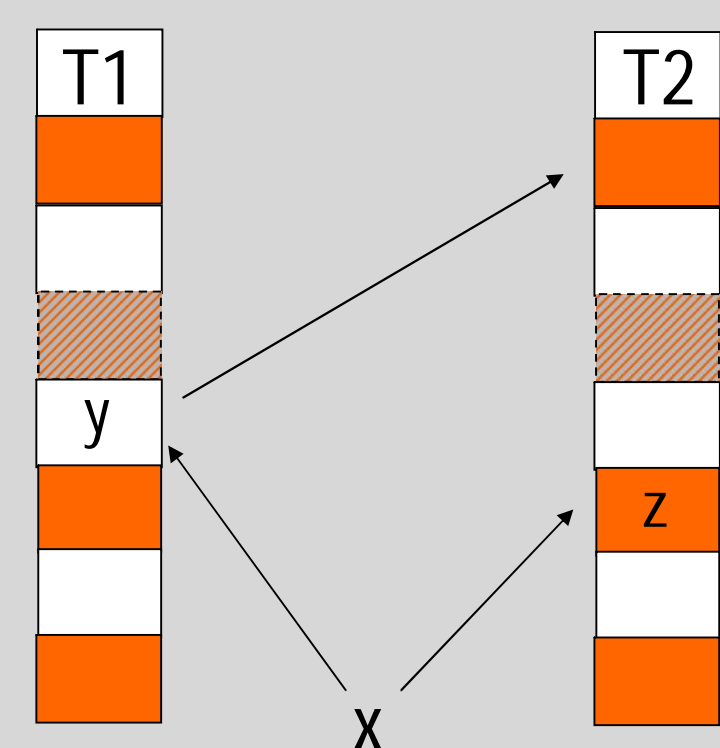
Linear-Probing tends to show better cache behavior, because linear-probes access the memory sequentially. This good cache behavior translates to good **scalability** of the algorithm (from Wikipedia)



Chained-hashing exhibits bad non-sequential memory access patterns. But at high table-density outperforms linear-probing which encounters long probing sequences due to the formation of key clusters.



Cuckoo-Hashing, an open addressed hashmap, uses two hash functions and series of displacements while inserting new keys, to ensure the lookup takes a deterministic constant number of probes, in contrast to *linear-probing*. But the price is (1) the need to use expensive hash functions, and (2) a maximal table density of about 50%.

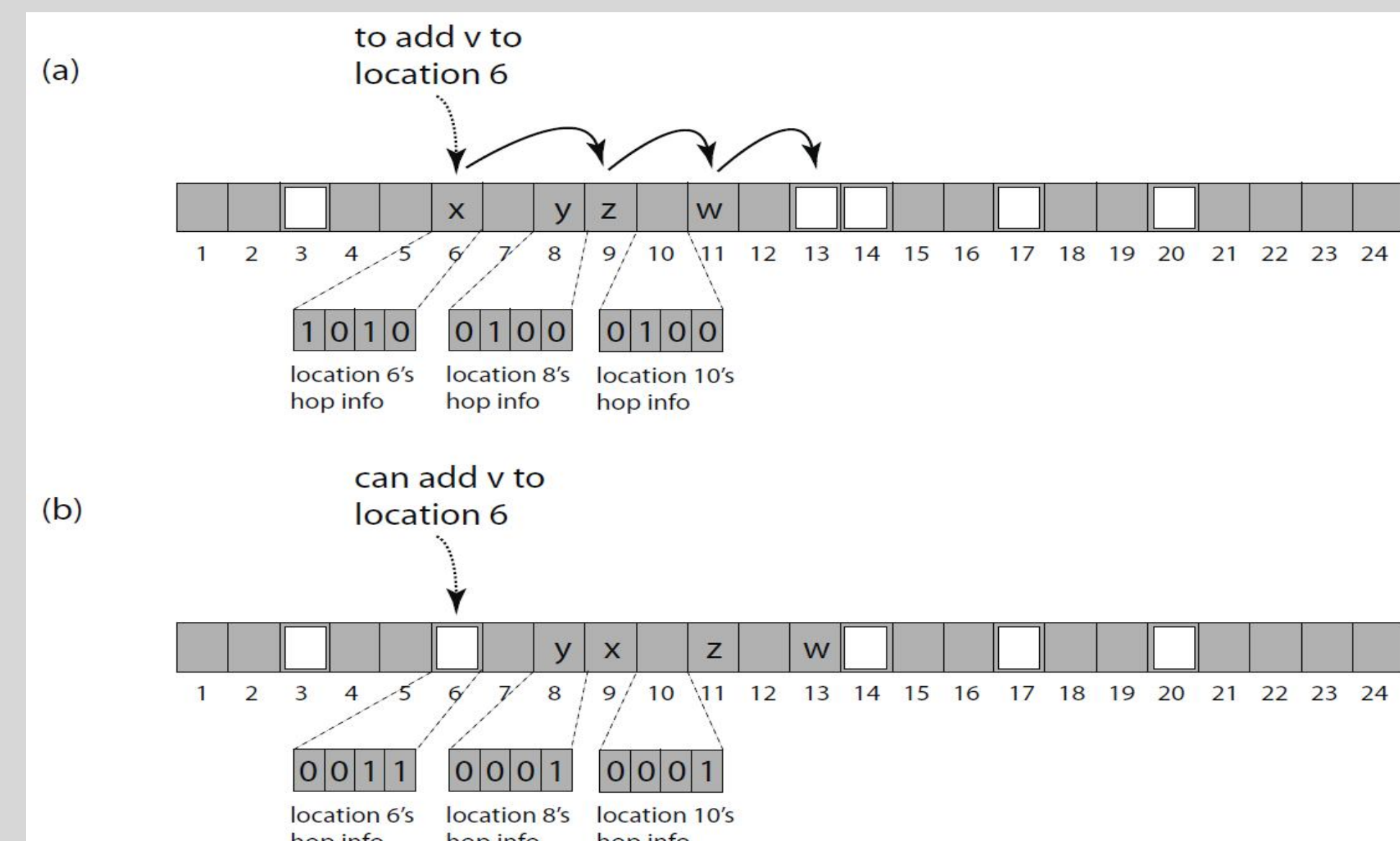


The New Hopscotch Algorithm

Our new hopscotch hashing algorithm is based on a novel multi-phased probing and displacement technique that combines elements of the *cuckoo-hashing* and *linear-probing* algorithms described above, but overcomes their drawbacks. Hopscotch hashing combines these two approaches as follows. 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). 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. In this way, an item can be found quickly (in exactly two steps) and deterministically.

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 slot 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, resize and rehash the table.



Above is an execution of the algorithm. 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 .

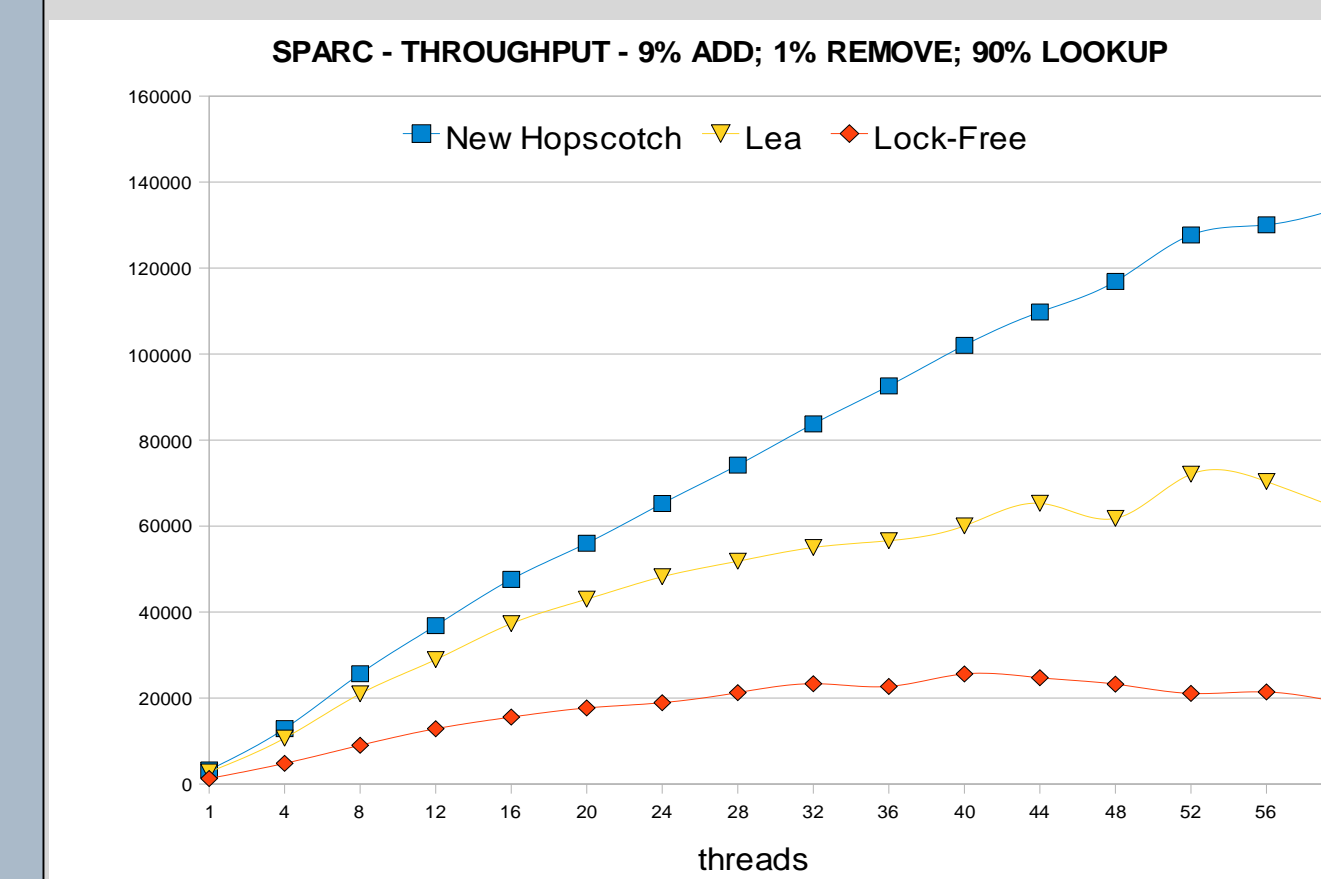
The Key Insight

In cuckoo hashing, the 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 half 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 table densities.

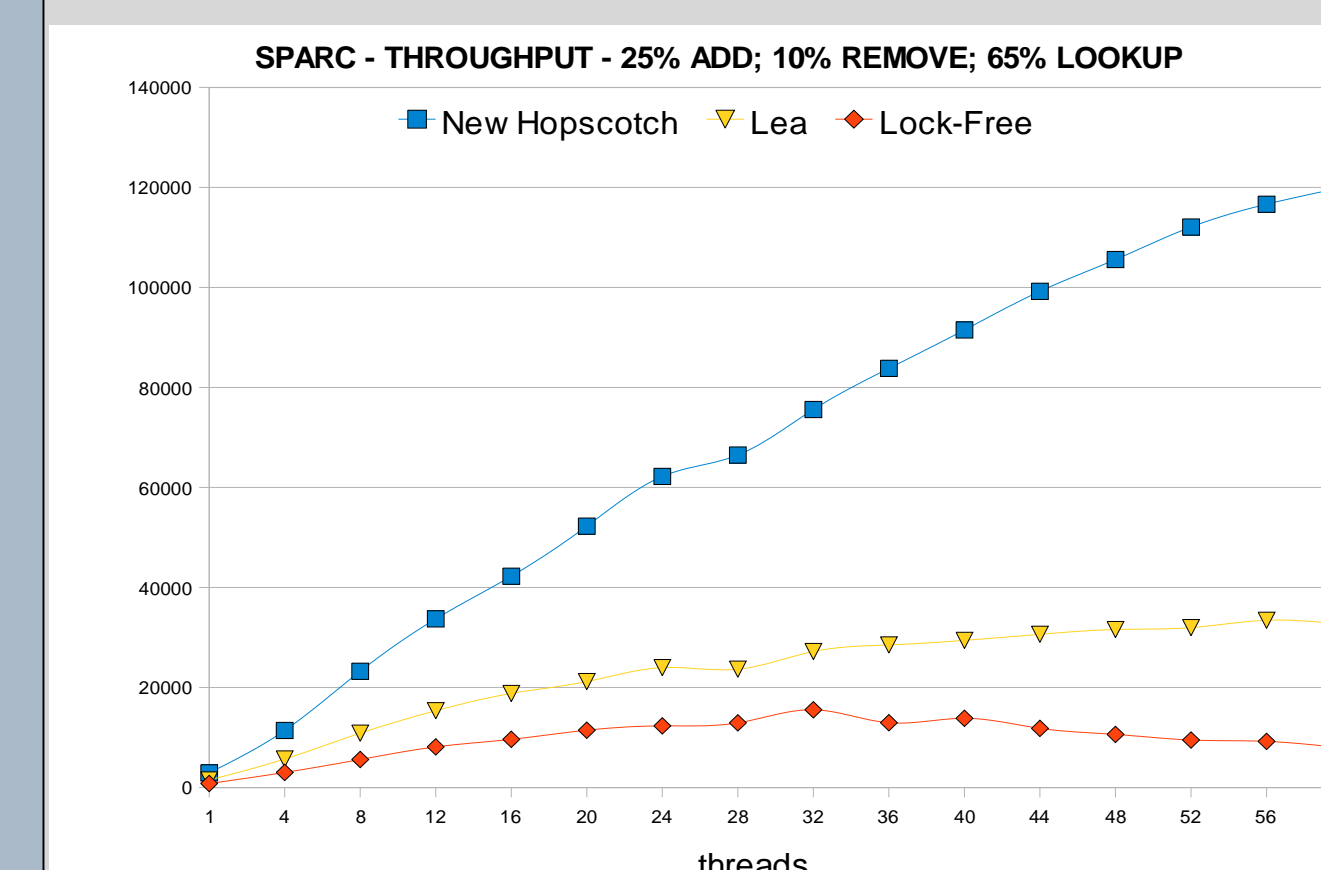
Concurrent - Results

We ran a series of benchmarks on a 64-way Sun UltraSPARC T2TM, a multi-core machine based on the Niagara II architecture that has 8 cores, each supporting 4 multiplexed hardware threads. We compared our algorithm to the two leading concurrent hash maps: Lea's ConcurrentHashMap from the Java concurrency package and Shalev and Shavit's lock-free Hash-Map.

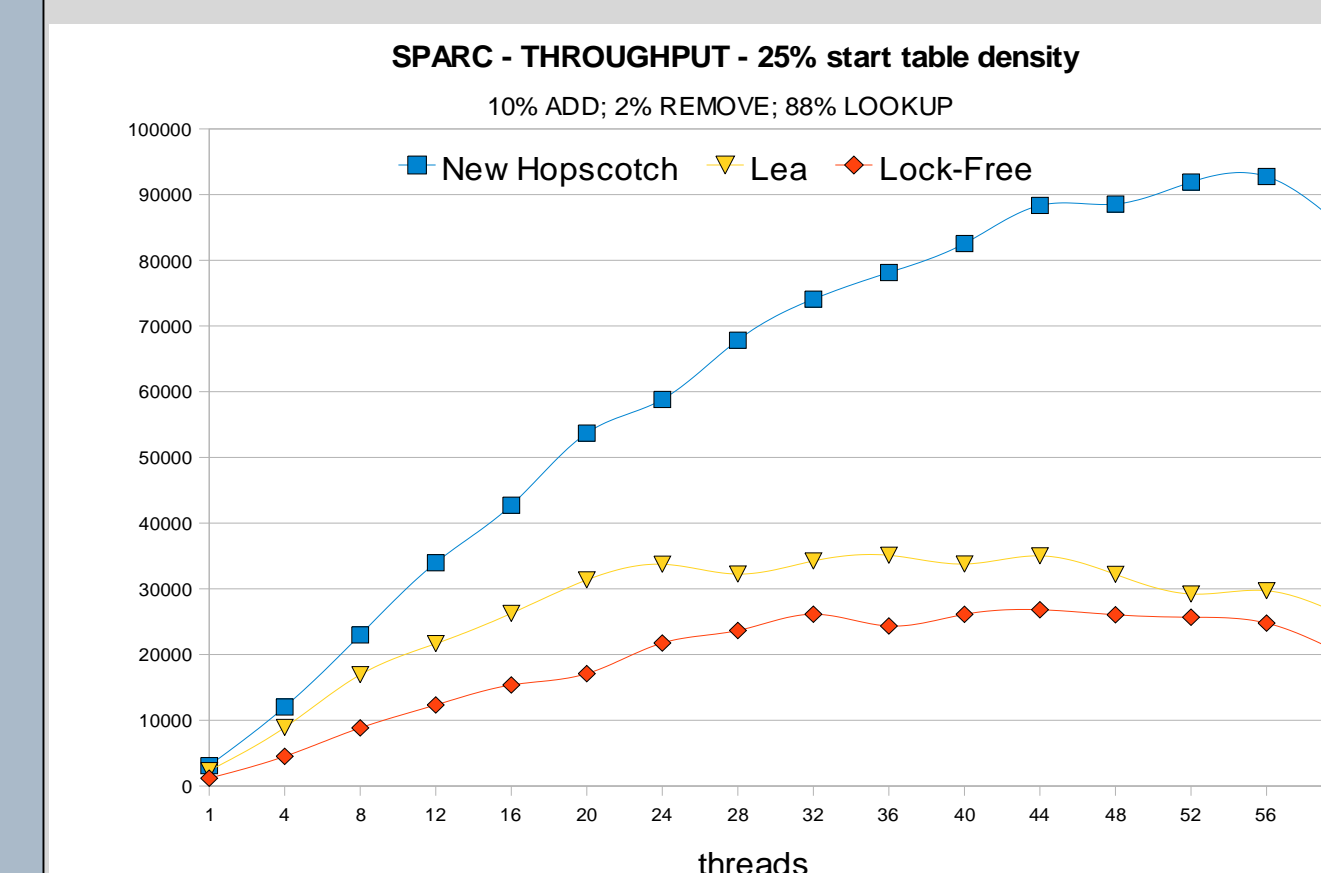
In our benchmarks we measured the change in throughput as a function of concurrency under various synthetic distributions of method calls.



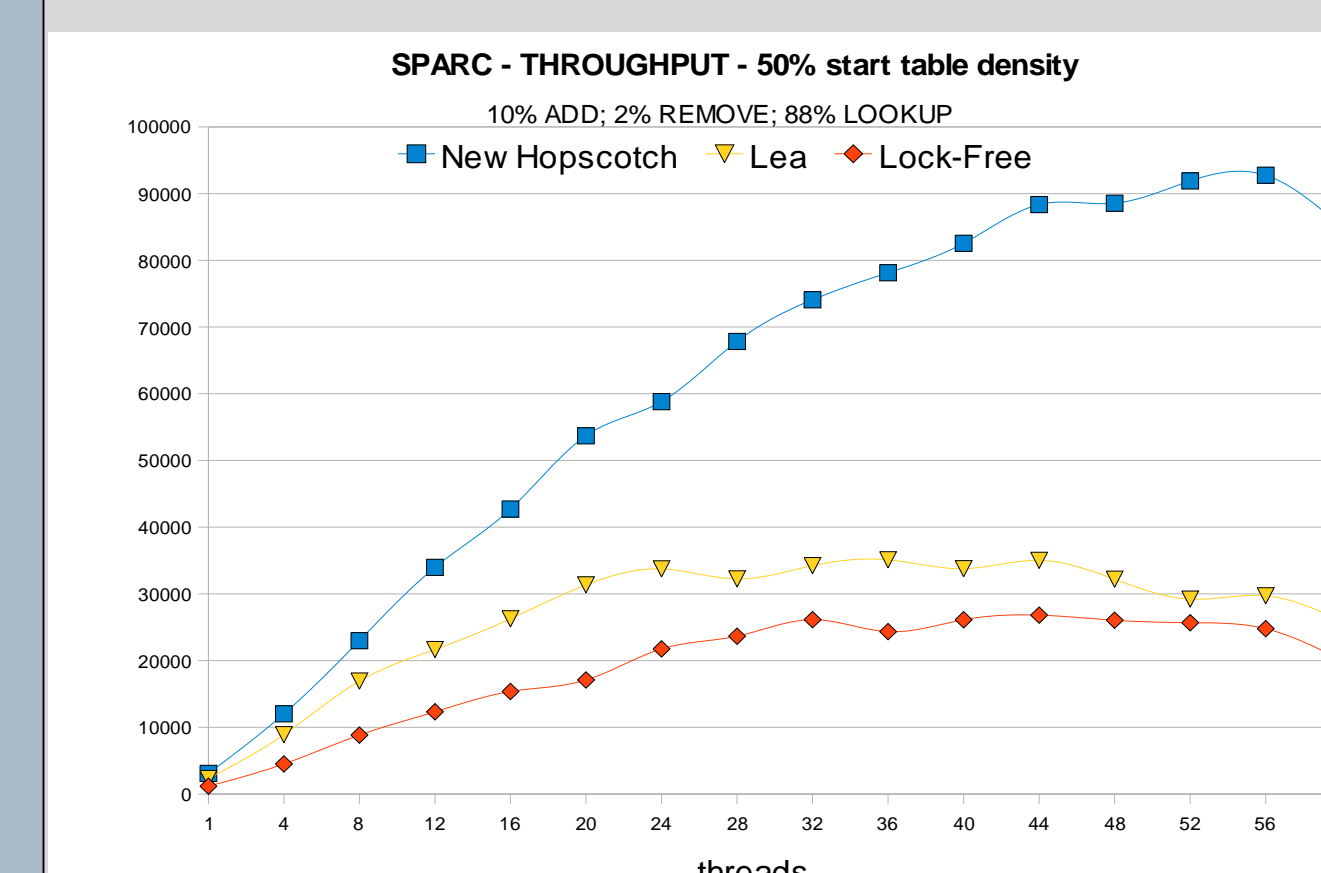
In this first benchmark, we started with an empty table. We used a mix that consisted of 90% contains(), 9% add()s and 1% remove()s. As can be seen the hopscotch scales better, notice that it is almost perfect linear scale.



This benchmark also starts with empty table. But now the emphasize is on insert & remove, mix that consisted of 65% contains(), 25% add()s and 10% remove()s. As can be seen the hopscotch scales much better, and we can deduce that insert & remove scale much better too. Again notice that it is almost perfect linear scale.



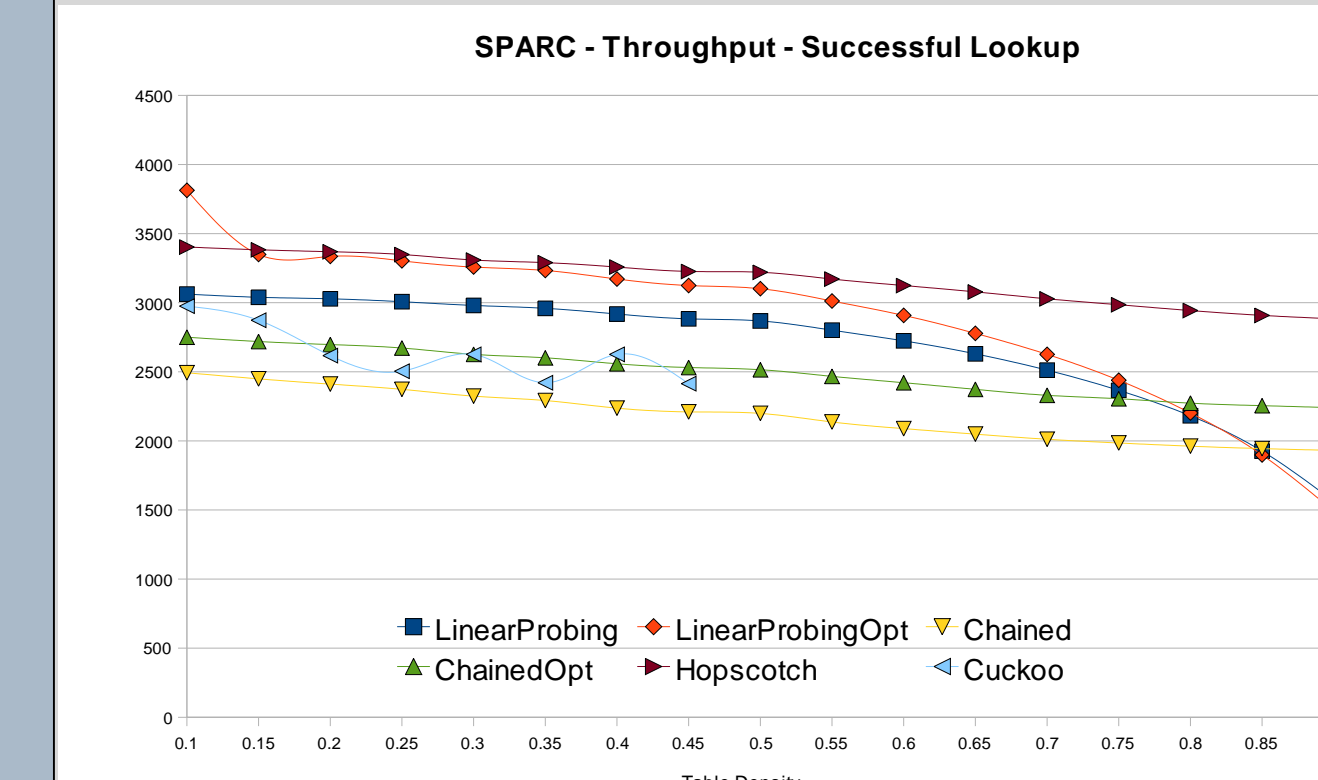
Now we test the how the table-density affects the performance. We used a mix that consisted of 88% contains(), 10% add()s and 2% remove()s. Notice that the hopscotch almost not affected by the 25% table-density, and still manage to keep the linear scalability behavior, in contrast with the other hash-maps.



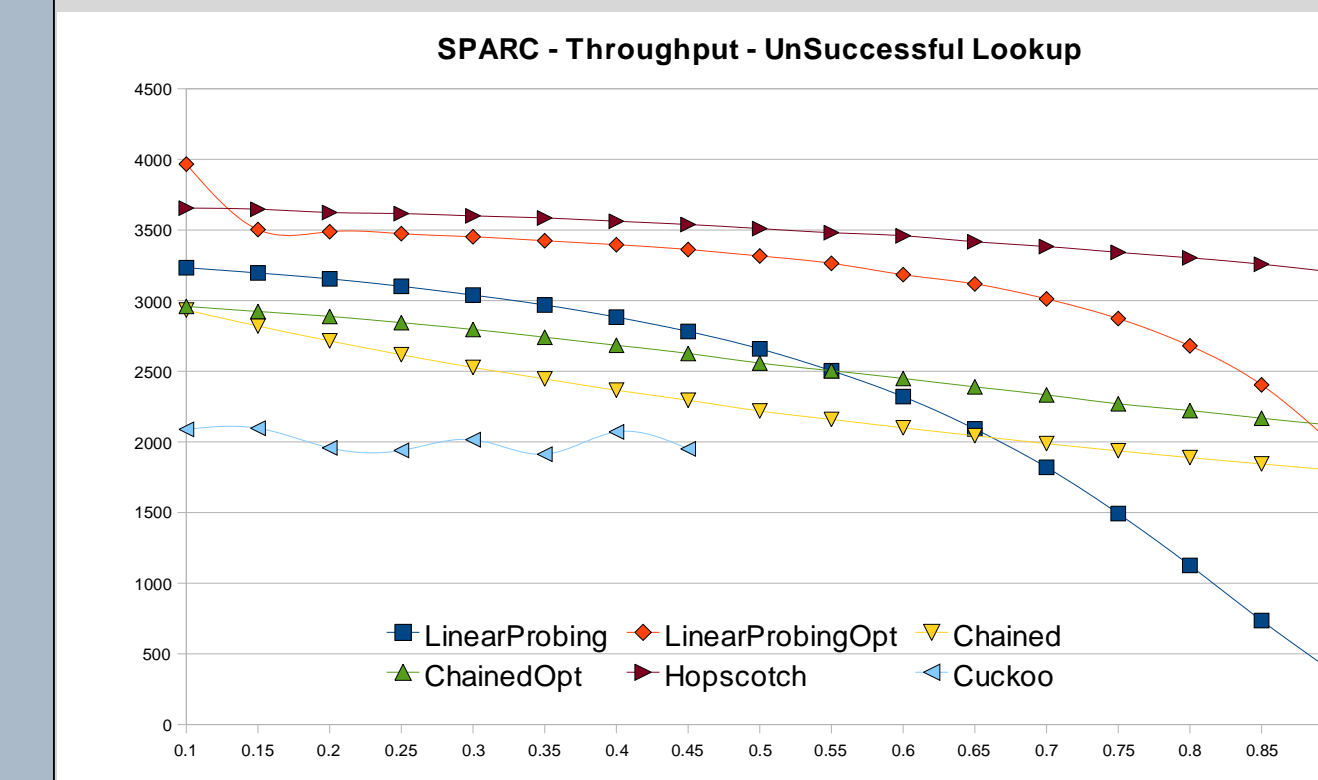
To emphasize even better the resistance of the hopscotch to the table-density, we now start with 50% table density. Again we used a mix that consisted of 88% contains(), 10% add()s and 2% remove()s.

Sequential Results

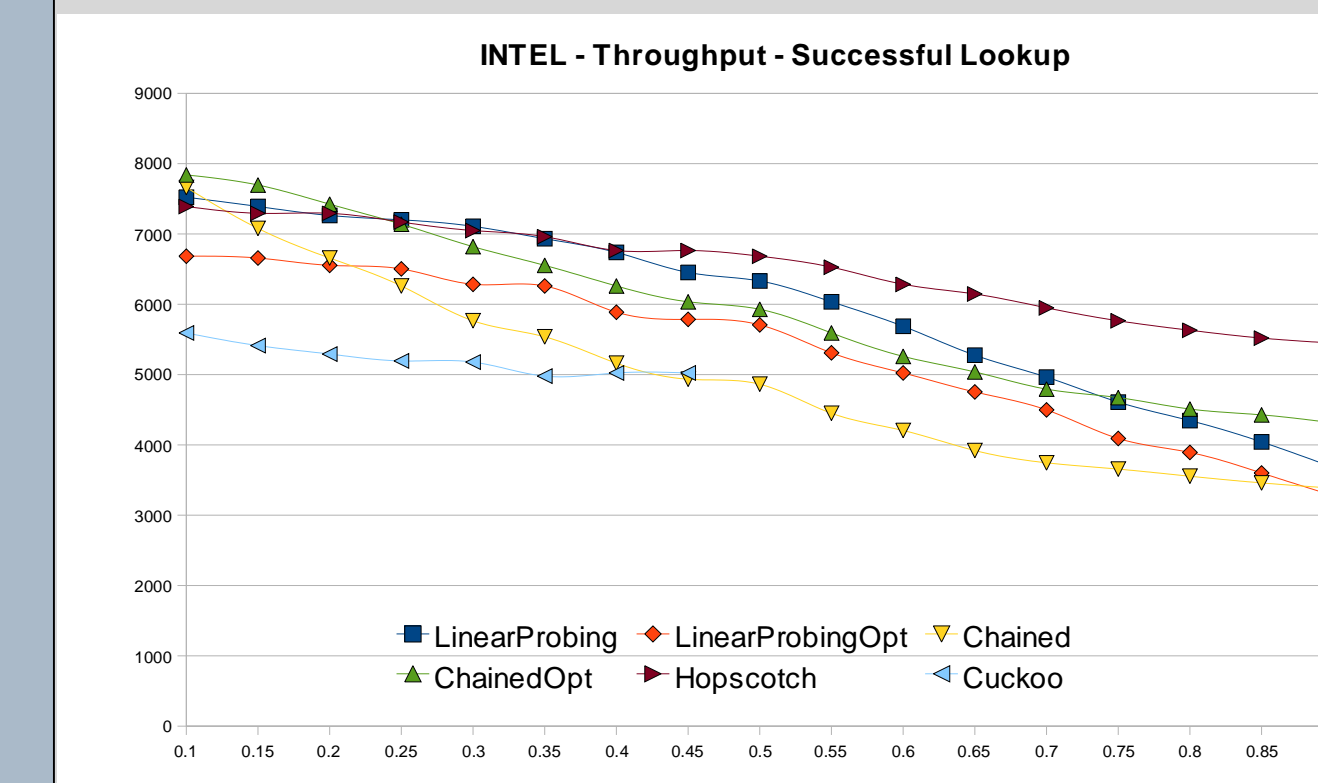
The sequential benchmarks, compare our new hopscotch to other classic sequential hash-maps: (1) Linear-Probing is the classic algorithm. (2) Linear-Probing-Opt is an optimized version that keep track of the probing-bound to limit the probing length. (3) Chained is the classic chained hash-map. (4) Chained-Opt is an optimized version that use the list-heads to store key & data. (5) Cuckoo - is the classic cuckoo hash-map. We measured the change in throughput as a function of table-density.



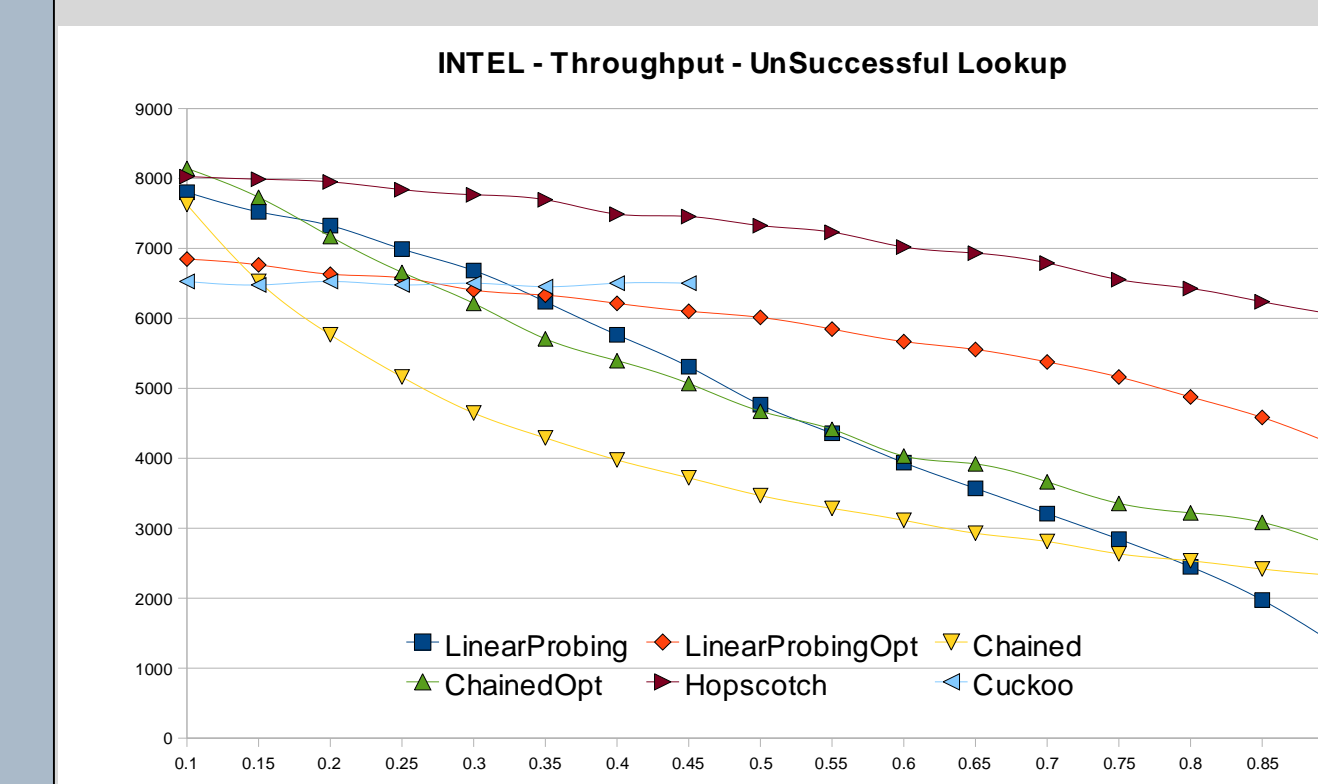
This benchmark tests the successful-lookup throughput on a SPARC machine. Its quite clear that the new hopscotch algorithm outperforms the other hash-maps, but the important point is the *high level of immunity* to increases in *table-density*



Now we test the Unsuccessful-lookup throughput on a SPARC machine. Again the same findings. Notice the drastic decrease in performance, exposed by linear-probing & cuckoo hash-map.



The benchmarks on the INTEL architecture exposed the same behavior as above.



The INTEL Unsuccessful-lookup benchmark results, similar to the benchmark on the SPARC machine. An exception is the cuckoo hash-map, that enjoys the arithmetical qualities of the INTEL architecture .

Indicating that our findings are not platform specific.

Conclusions

The new hopscotch combines the good qualities of closed-addressing and open-addressing, e.g. good cache behavior and immunity to increases in table-density. The design emphasize on cache efficiency, proved itself and provided extreme scalability. Our conclusion from our empirical testing is that the new hopscotch hash algorithm maintains a performance advantage over other leading hash-tables while maintaining a high memory utilization.

For further information

- Visit the site <http://groups.google.com/group/hopscotch-hashing> for further documentation, references, and source-code.
- Nir Shavit: shanir@cs.tau.ac.il
- Moran Tzafrir: morantza@gmail.com