6.851: Advanced Data Structures

Implementation and Performance Analysis of Hash Functions and Collision Resolutions

Victor Jakubiuk Sanja Popovic

Massachusetts Institute of Technology
Spring 2012

Contents

1 Introduction

Hash functions are among the most useful functions in computer science as they map large sets of keys to smaller sets of hash values. Hash functions are most commonly used for hash tables, when a fast key-value look up is required. Because they are so ubiquitous, improving their performance has a potential for large impact.

We implemented two hash functions (simple tabulation hashing and multiplication hashing), as well as four collision resolution methods (linear probing, quadratic probing, cuckoo hashing and hopscotch hashing). In total, we have eight combinations of hash functions and collision resolutions. They are compared against three most popular C++ hash libraries: `unordered_map` structure in C++11 (GCC 4.4.5), `unordered_map` in Boost library (version 1.42.0) and `hash_map` in libstdc (GCC C++ extension library, GCC 4.4.5). We analyzed raw performance parameters: running time, memory consumption and cache behavior. We also did a comparison of the number of probes in the table depending on the load factor, as well as dependency of running time and load factor.

2 Theoretical background

We begin with a brief theoretical overview of the hash functions and collisions resolution methods used.

2.1 Hash functions

Formally, a hash function is a function that maps a large universe of keys $U = 0, 1, \ldots, u$ to a smaller set of numbers $T = 0, 1, \ldots, m$. Because a hash functions lies on critical paths in a hash map, it needs to provide low latency. Thus, we only use multiplication hashing and tabulation hashing.

Let p be a large prime number such that $p \geq u$, m be the size of the table, and let a and b be random numbers such that $0 < a < p$ and $0 \leq b < p$. Multiplication function h is defined as:

$$h(x) = [(ax + b) \mod p] \mod m.$$

Simple tabulation hashing treats key x as an array of bytes $x_1, x_2, \ldots x_c$. We pregenerate a matrix of random numbers and use it to calculate the hash function:

$$h(x) = T_{1,x_1} \oplus T_{2,x_2} \oplus \ldots \oplus T_{c,x_c}$$

2.2 Collision resolution

Since a larger set is mapped into smaller one, due to the Pigeon Hole principle, it is unavoidable to have collisions, i.e. two keys mapped to the same bucket. There are two ways to resolve collisions: chaining and open addressing. With chaining, every bucket of the hash table is the head of a linked list. In case of collisions, an element is simply appended to a linked list. While simple to implement, this approach has the worst-case lookup time of $O(n)$ (where n is the number of elements in the hash map), which happens when all elements hash to the same bucket. On average, if all the slots are equally likely to be filled, it has $O(\alpha)$ lookup time where α is a load factor. In case of few collisions, chaining is acceptable because the chains are not very long, so both inserts and lookups run in approximately constant time. However, elements in linked lists are not linearly placed in RAM, which results in many cache-misses and performance degradation.

In open addressing, a hash table is probed in a certain pattern until an empty slot is found. All of the methods we examine are based on open addressing.

2.2.1 Linear and quadratic probing

The most obvious approach to open addressing is linear probing. If h is the original hash bucket, we probe consecutive buckets in linear order: $h, h+1, h+2, \ldots, h+i$. Another approach is quadratic probing, where we pick two constants, c_1 and c_2 (in practice, they are often 1/2) and the probe sequence becomes: $h, h+c_1 \cdot 1+c_2 \cdot 1^2, h+c_1 \cdot 2+c_2 \cdot 2^2, \ldots h+c_1 \cdot i+c_2 \cdot i^2$.

2.2.2 Cuckoo hashing

Cuckoo hashing uses d tables of size m/d each, instead of one of size d. Each hash table is associated with a different hash function. On insert, we try to insert the element in the first table. If the bucket is not empty, we take it out and insert into the next table. We continue this process until we insert an element into an empty slot or until we reach a cycle. In the latter case, we generate new hash functions for each of the tables and rebuild tables accordingly. Table rebuilding is a costly operation, but the probability of it happening is low.

2.2.3 Hopscotch hashing

Hopscotch hashing is an open-addressed hash table with a modified linear probing. It was proposed by Herlihy, Shavit and Tzafrir in 2008 [2]. The idea is to keep items not more than H buckets away (in the H-neighborhood) from their originally assigned buckets. H is a predetermined constant, which in practice is $\log u$, which means 32 or 64. This yields good cache locality, since the entire neighborhood can fit into just a few cache lines (usually, not more than 3).

On inserts, if the probed bucket h is full, we do linear probing until we reach an empty bucket b_i. If it is more than H slots away, we rearrange data so the empty slot gets closer to h. We first find a key from $[b_i - H + 1, b_i]$ which guarantees that moving that key to b_i does not break H-proximity invariant. After the move, we have an empty slot closer to h.

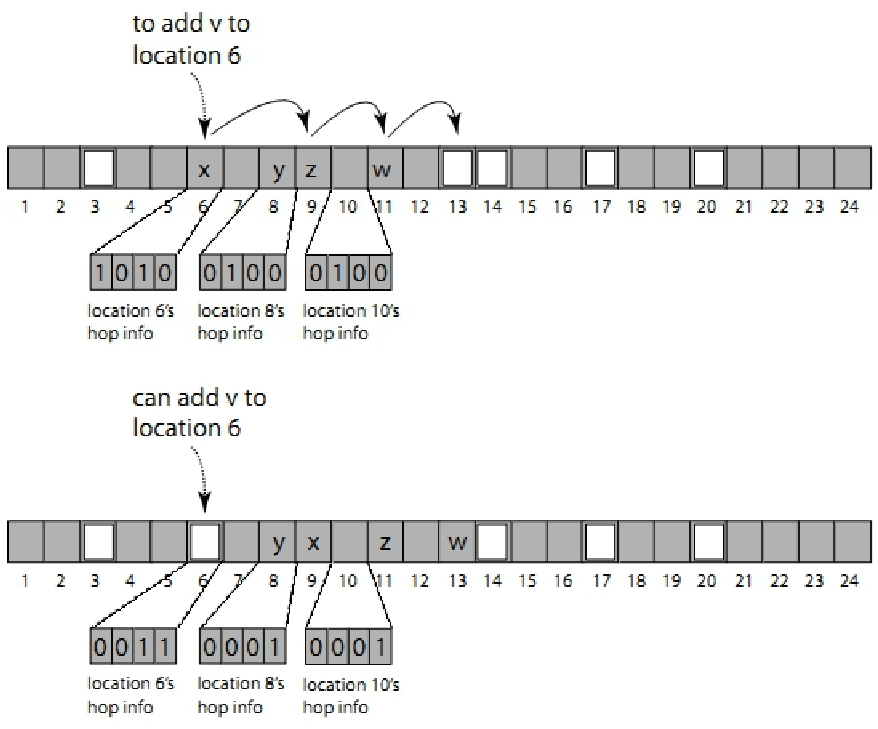


Figure 1: Hopscotch hashing diagram

We repeat this until the empty slot is in $[h, h+H-1]$. The table never needs to be resized because the probability of not being able to create an empty slot within H from the desired bucket is very low and happens only at extremely high loads ($> 95\%$ for $H = 64$).

3 The Main Benchmark

In order to measure the running time, we used a random array containing $900,000$ elements. We inserted them into a hash table of size $1M$ which gave the load factor $\alpha = 0.9$ for our tables (the built-in hash tables automatically resize). We did not resize the hash tables in order to analyze the behavior under high loads. The inserts were followed by $9M$ reads, where we requested random permutations of the inserted arrays. We run the benchmarks on a 12-core Intel Xeon X5650 CPU running at 2.67GHz with a 12MB cache. Figure 2 shows the running times and the number of cache references per query and Figure 3 presents the number of cache misses per query.

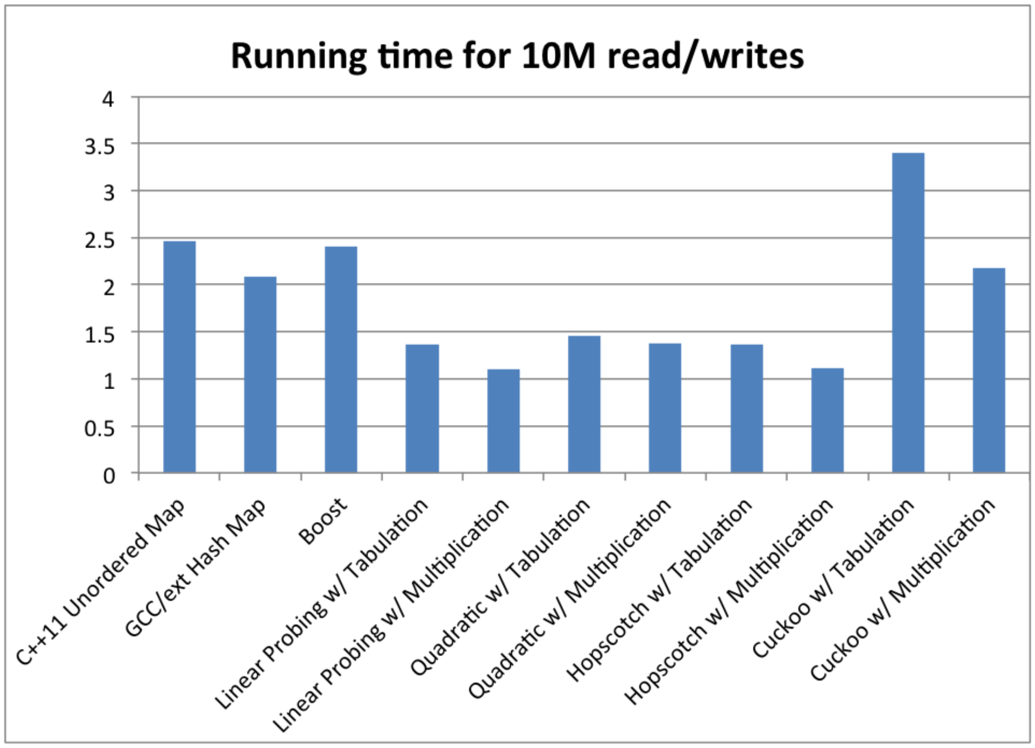
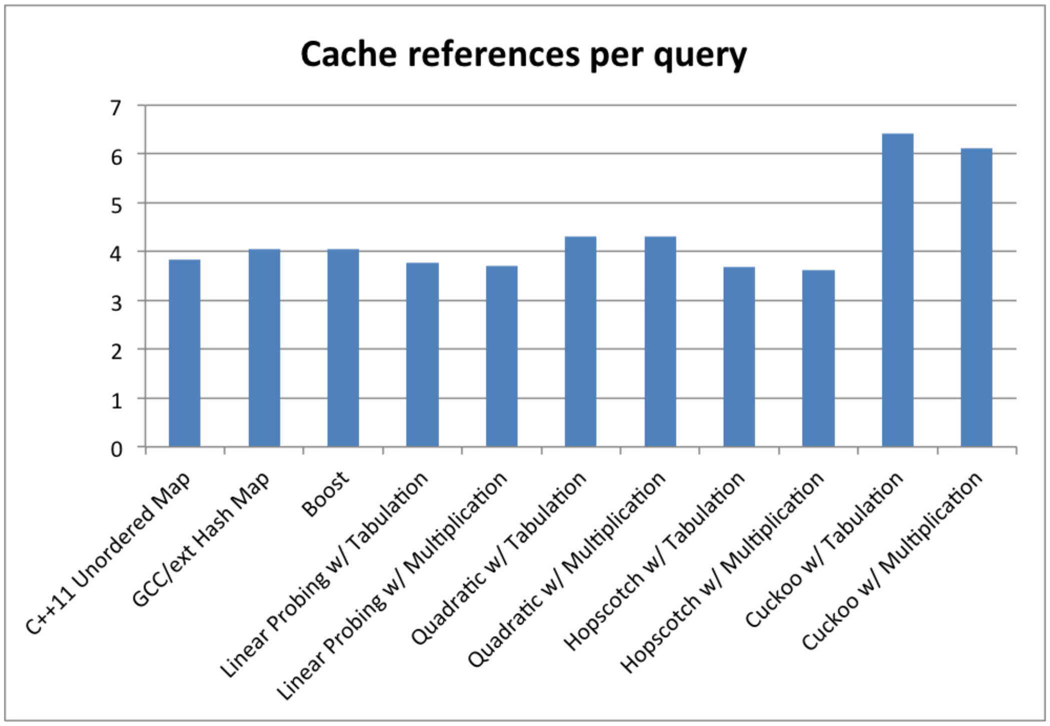

Running time for 10M read/writes
4
3.5
3
2.5
2
1.5
1
0.5
0
C++11 Unordered Map
GCC/ext Hash Map
Boost
Linear Probing w/ Tabulation
Linear Probing w/ Multiplication
Quadratic w/ Tabulation
Quadratic w/ Multiplication
Hopscotch w/ Tabulation
Hopscotch w/ Multiplication
Cuckoo w/ Tabulation
Cuckoo w/ Multiplication

Cache references per query
7
6
5
4
3
2
1
0
C++11 Unordered Map
GCC/ext Hash Map
Boost
Linear Probing w/ Tabulation
Linear Probing w/ Multiplication
Quadratic w/ Tabulation
Quadratic w/ Multiplication
Hopscotch w/ Tabulation
Hopscotch w/ Multiplication
Cuckoo w/ Tabulation
Cuckoo w/ Multiplication

Figure 2

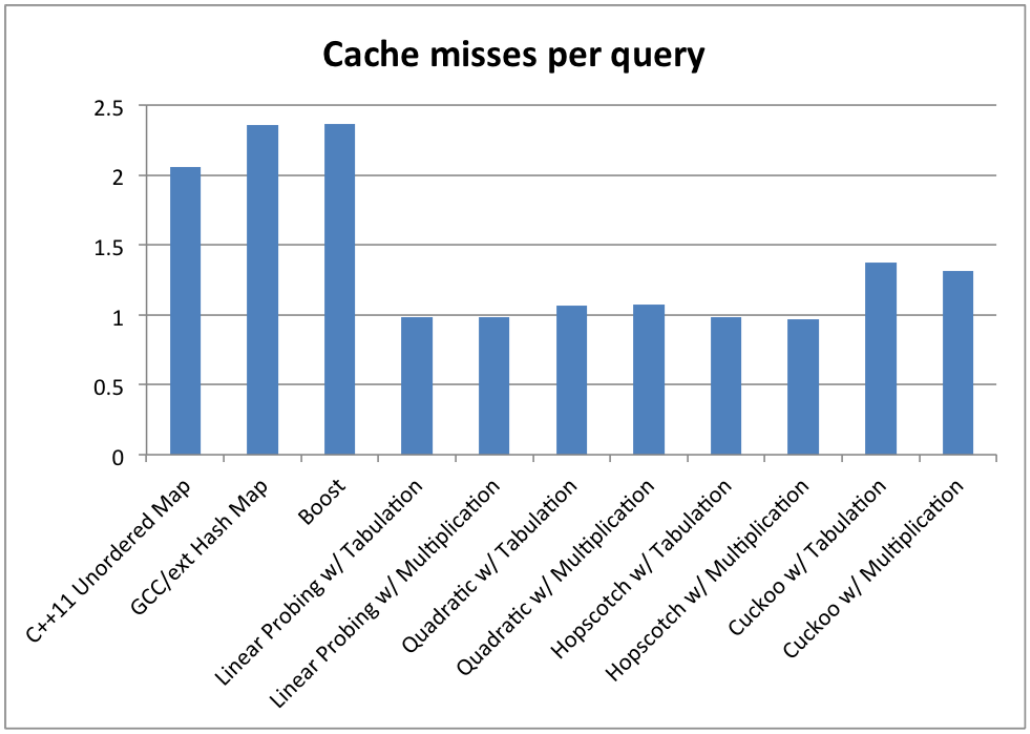


Figure 3

4 Discussion

Our experiments show that theoretical analysis is not sufficient to predict algorithm's behavior in practice. In particular, standard running time analysis does not take into account cache behavior, which has significant impact on performance.

The first thing we noticed is that the choice of a hash function does not significantly affect the number of probes or the running time - multiplication hashing is about 10% faster than tabulation hashing. This is expected because tabulation hashing for 64-bit keys requires 8 iterations of a loop to xor all intermediate hashes. Although bitwise operations used for dividing the key and *xor*-ing are fast, repeating them numerous times is still slower than multiplication hashing.

Another interesting observation is the overall running time. Our functions have almost exclusively outperfomed the standard hashing implementations. One of the reasons is that the standard libraries are templatized and optimized to work well in a general case, while we specifically hash integers so our functions avoid certain templates overhead. Although the C++ standard does not explicitily specify the implementation of hash maps, we believe that chaining is the most popular implementation.

Let us analyze the collision resolution methods one by one. Graph 4 and 5 show the dependency between the running time and number of probes versus the load. As expected, linear probing needs to do more probes than the other functions due to long, continuous sequences of filled buckets. However, the running time of linear hashing is among the best out of all the algorithms. This is because consecutive probes lie on the same cache line. Cache misses are very expensive so minimizing them has significantly improved the running time.

Quadratic probing performs similarly linear probing. Expectedly, it has fewer probes, but its running time is about the same. On one side, the sequence in which we probe makes it possible to avoid long series. On the other side, more cache lines are needed. Since we do not have many probes before we find an empty slot, this is still a relatively small offset with respect to h, and we need just a few lines of cache.

Cuckoo hashing has shown a very poor performance. Cuckoo with tabulation had by far the worst running time, while cuckoo with multiplication was on average 2-3 times slower than the rest of our implementations. It did have a very small number of probes per operation, but it was very unstable in a sense that its performance was highly dependant on a "good" choice of a hash function (notice the re-hashing jumps around the load factor of 10% and 75% on the figure 5). Also, jumping between the tables means large jumps in memory. Therefore, there are significantly more cache references (and misses) than for the any other algorithm.

In our experiments, the hopscotch hashing was about as good as the linear probing. Because all the probes are localized, cache is heavily utilized. Timewise, this makes up for the higher number of probes.

We tested all of our implementations without resizing the hash table, so we can test them at high loads. We could achieve good performance up to $\alpha = 90\%$ where α is the load factor. Going beyond that point is almost impossible. For hopscotch, if the table is almost completely full, there are buckets that do not have an empty slots anywhere nearby, so it is not possible to shift empty buckets into initial neighborhoods. For cuckoo hashing, once the load gets above 90%, cycles happen very often during rehashing, so the performance drops to almost zero.

5 Conclusion

With all the analysis, we can conclude that hopscotch hashing is the best and safest choice for a practical implementation of a hash map. It certainly outperforms cuckoo hashing and quadratic probing, mostly due to high cache hit-rate. Linear probing, although a tempting choice, carries a risk of a slowdown in long-term real-world test scenarios, due to deletions and contamination. Hopscotch is also relatively straightforward to implement (on par with linear probing) and has a potential to perform well in multithreaded environment [2].

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.

[2] M. Herlihy, N. Shavit, M. Tzafir, *Hopscotch Hashing*, in Proceedings of the 22nd international symposium on Distributed Computing, pp. 350–364, 2008.

[3] E. Demaine, *6.851 Lecture notes*, 2012.

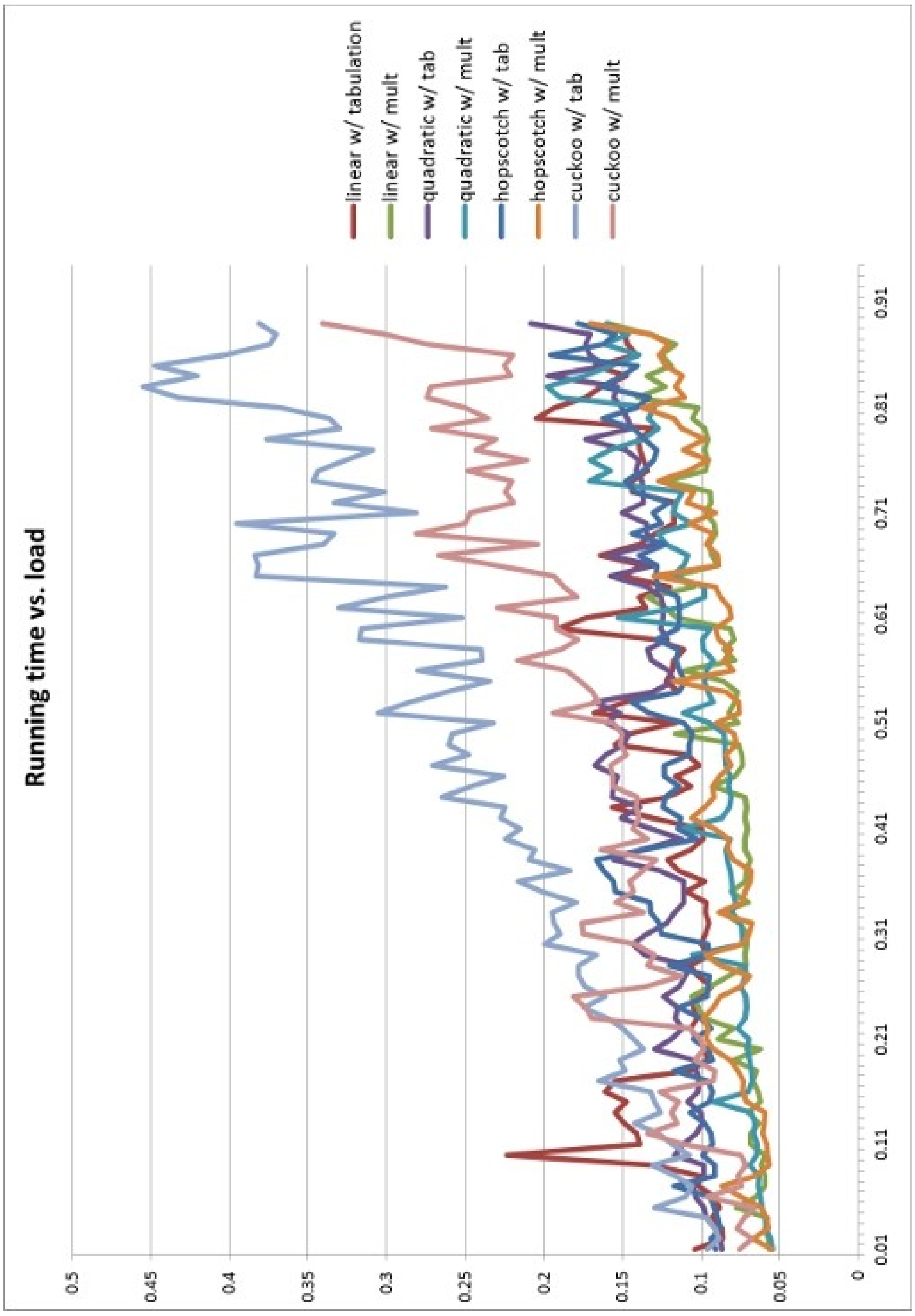

Running time vs. load
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.01
0.11
0.21
0.31
0.41
0.51
0.61
0.71
0.81
0.91
linear w/ tabulation
linear w/ mult
quadratic w/ tab
quadratic w/ mult
hopscotch w/ tab
hopscotch w/ mult
cuckoo w/ tab
cuckoo w/ mult

Figure 4

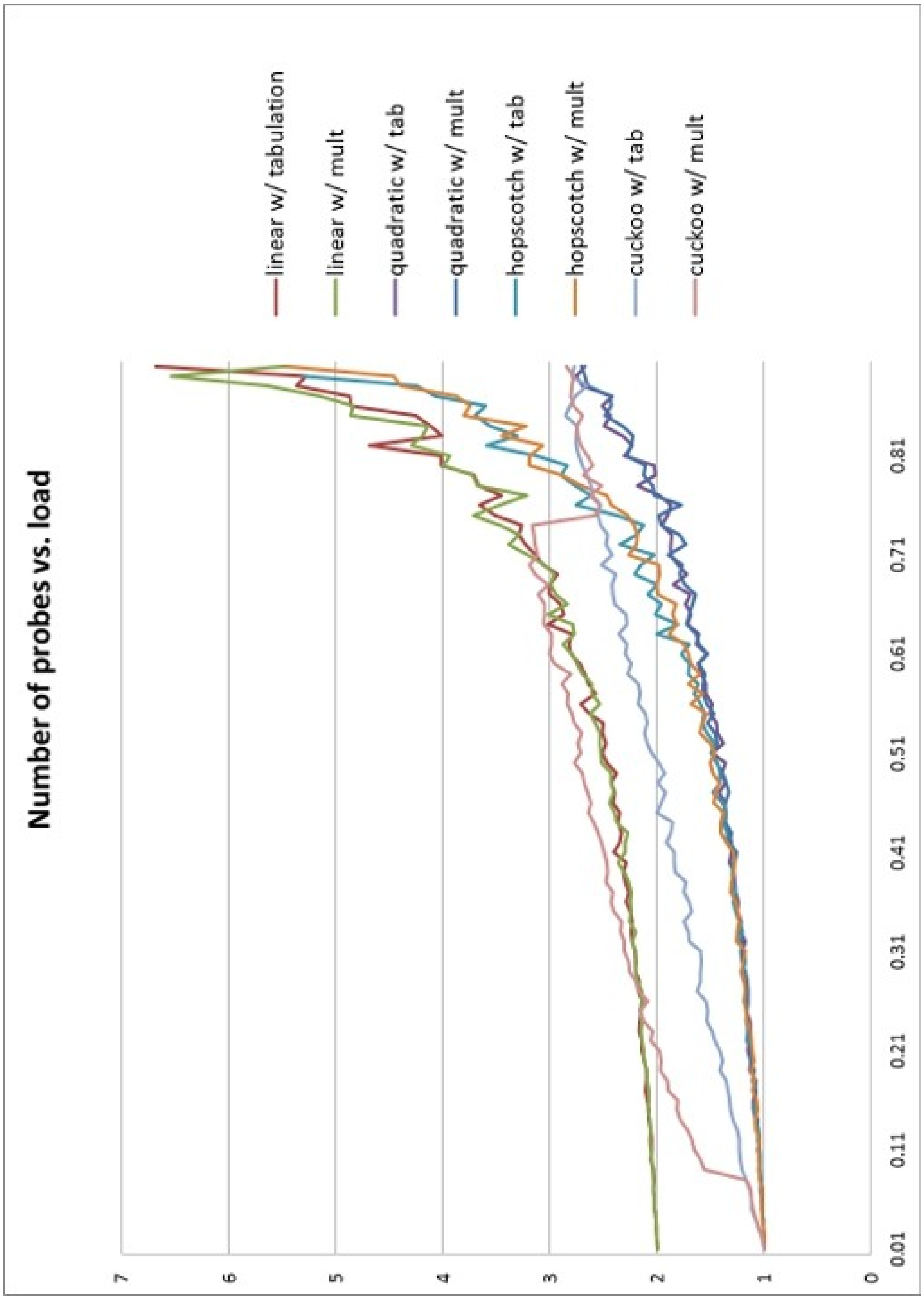
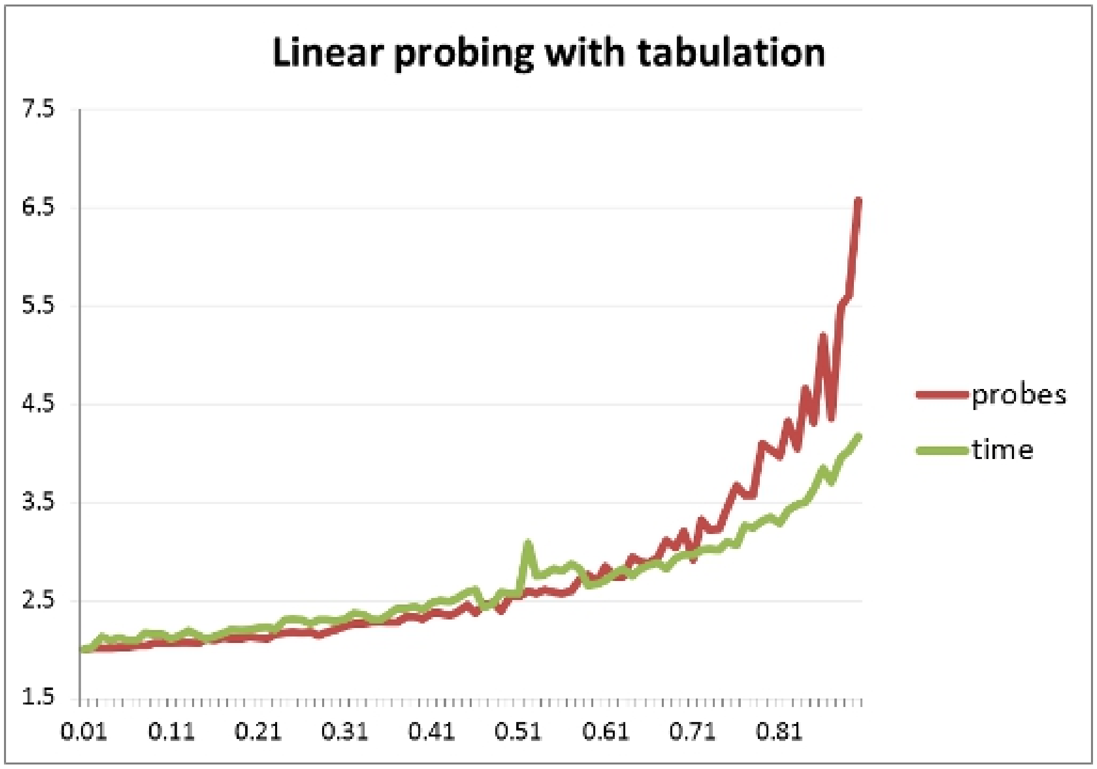

Number of probes vs. load
0
1
2
3
4
5
6
7
0.01
0.11
0.21
0.31
0.41
0.51
0.61
0.71
0.81
linear w/ tabulation
linear w/ mult
quadratic w/ tab
quadratic w/ mult
hopscotch w/ tab
hopscotch w/ mult
cuckoo w/ tab
cuckoo w/ mult

Figure 5



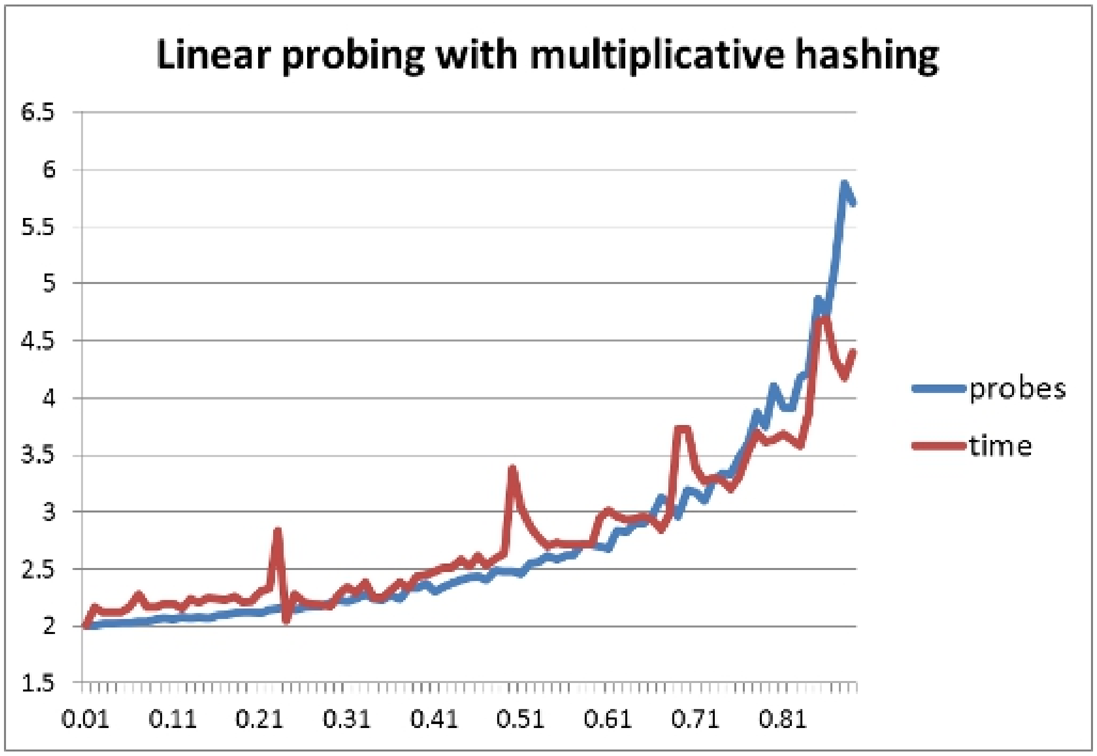
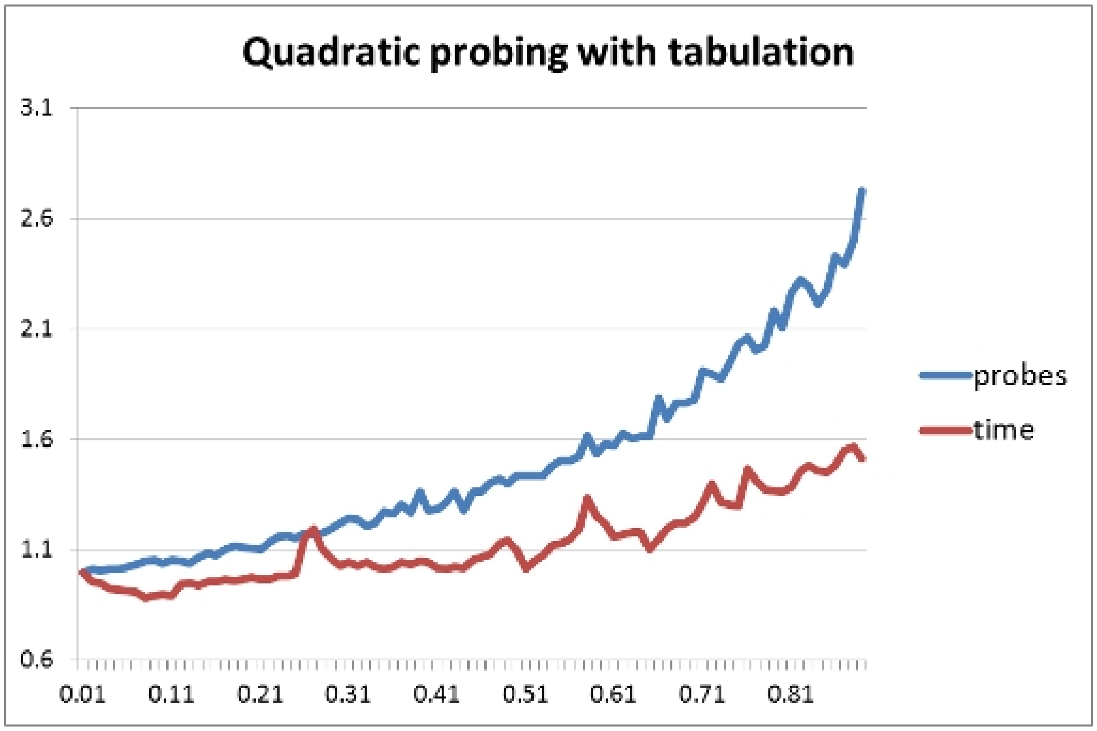


Figure 6: Linear probing



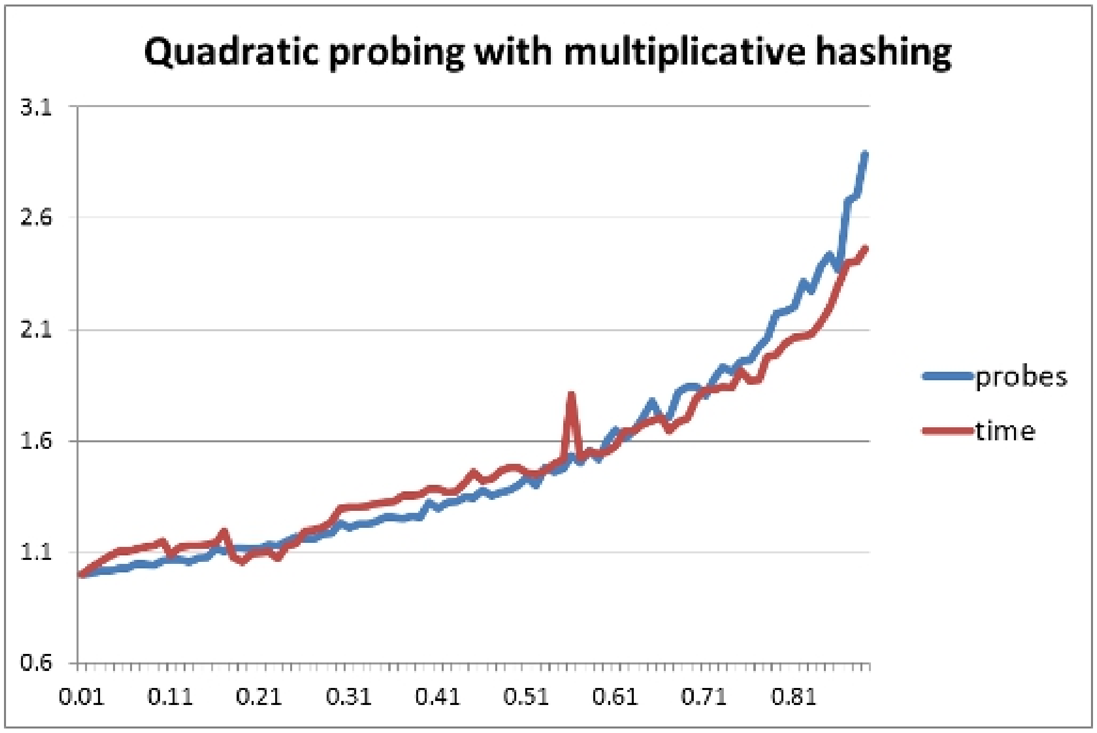
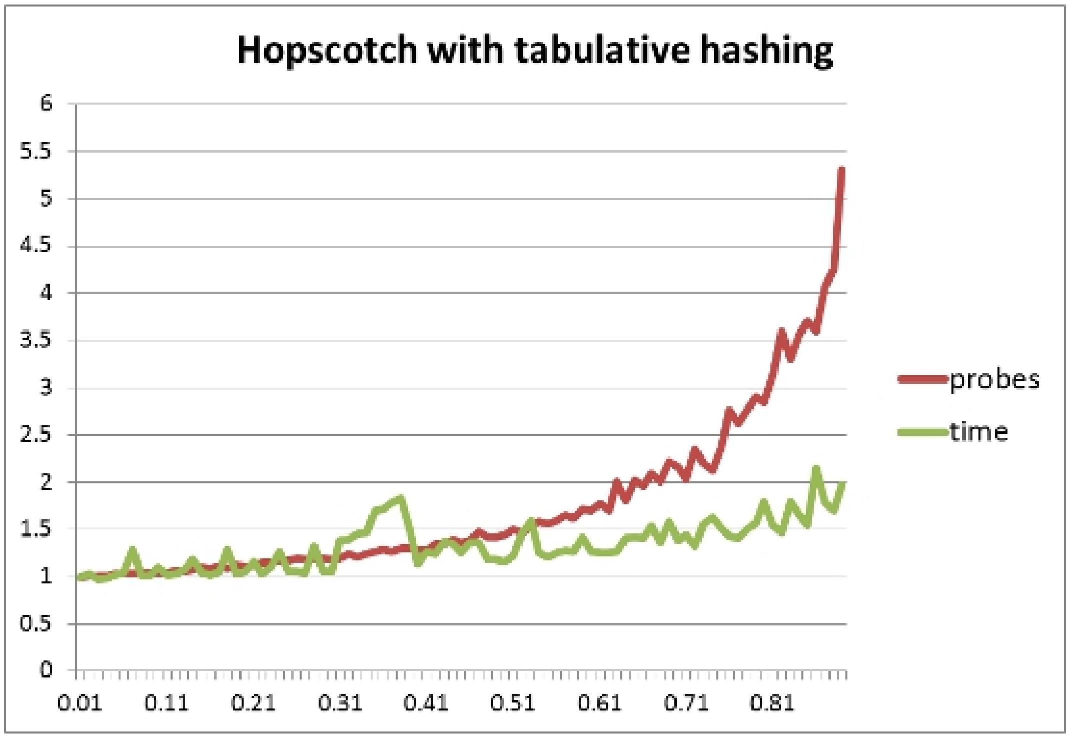


Figure 7: Quadratic probing



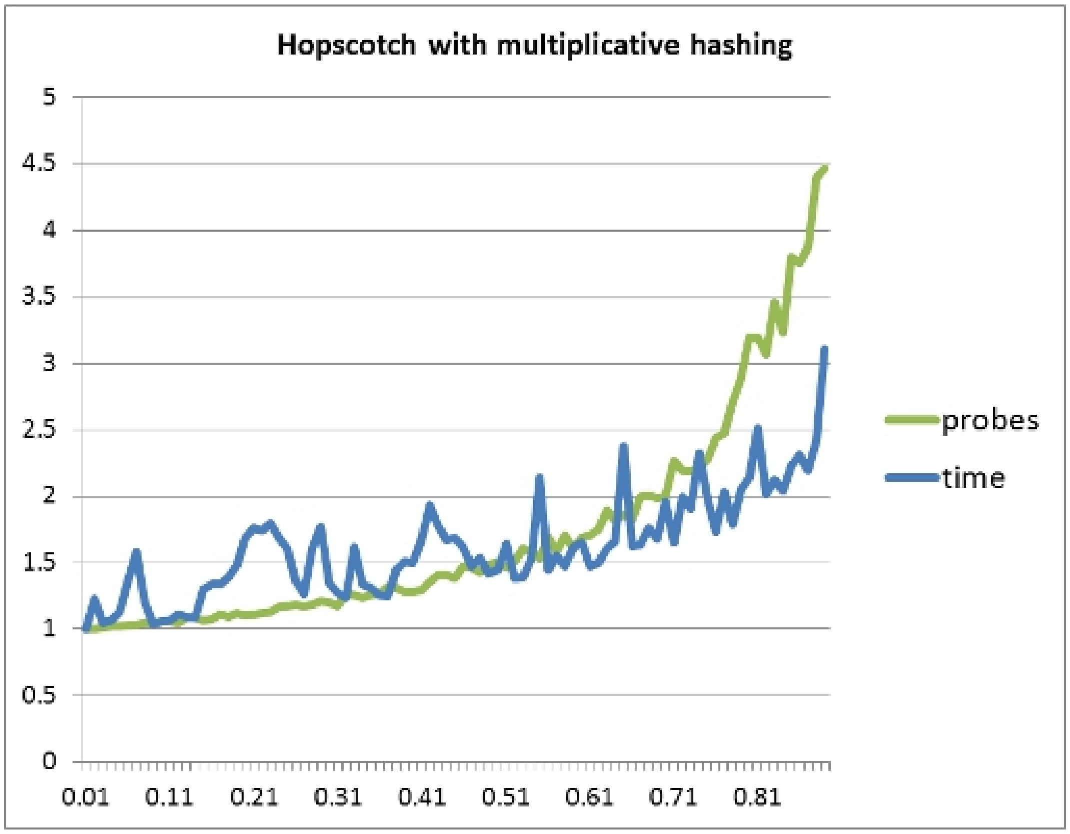
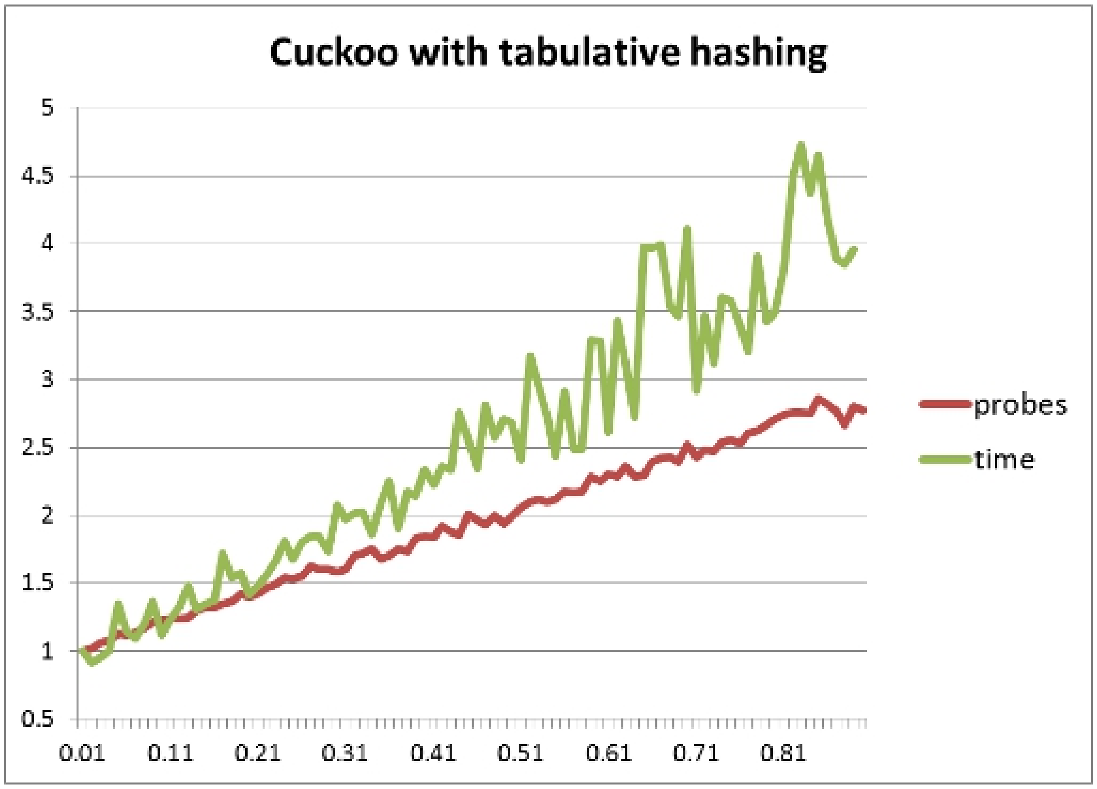


Figure 8: Hopscotch hashing



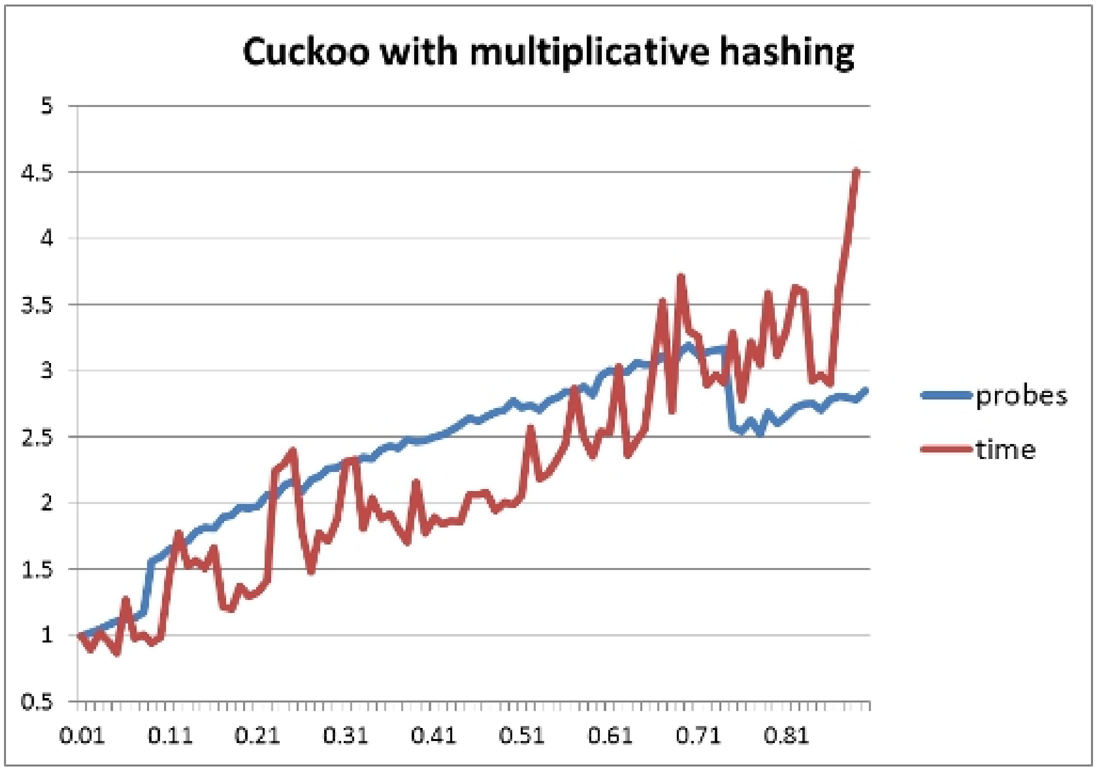


Figure 9: Cuckoo hashing