6.851 Final Project
Tabulation Hashing Performance Benchmark

Maksim Stephenako
Yuzhi Zheng

May 2012

1 Introduction

Hashing is one of the most basic computer science concept. It allows elements to be reliably stored and retrieved from a limited number of slots, without dedicated slot of every possible variation of the element. While basic, hashing is used everywhere. Hashing is used in associative arrays, sometimes also known as dictionaries, in languages like PHP, Perl, and Python. Hashing can event be used for database indexing. Even lower level computer architectural components like processor caches use ideas from hashing to figure out which line to store value from a particular memory address. Hashing can also be used to keep track of sets or make sure certain data representations are unique. Even the famous MapReduce framework uses hashing to help shard inputs to be processed on different machines.

From a theoretical standpoint, hashing takes $O(1)$ time, which means it takes a constant amount of time. That is essentially as fast as it gets. However, big-O notations can not accurately depict the size of the constant factor. These constant factors sometimes have a significant but real influence on the performance of any algorithm. Since hashing is used so often, it is important to keep that constant factor as low as possible, and finding improvements whenever possible.

One of the most basic hashing function is the multiplicative hashing. Thorup and Zhang showed that a different type of hashing, tabulation hashing, could potentially be a good alternative to the more basic multiplicative hashing in their paper from 2010. More specifically, they looked at the performance of tabulation hashing used in conjunction with linear probing and found the performance to be competitive with other hash functions on dense tables.

This report takes a closer look at tabulation hashing and it's performance against the basic multiplicative hashing. Instead of only looking at linear probing, we expanded our collision resolution techniques to quadratic probing and also chaining. We plan to do some benchmark testing as well as analyzing the possible pros and cons of each type of hash functions as well as the different collision resolutions.

2 Tabulation Hashing

overall idea of tabulation hashing

make table
look stuff up etc
- 3 independence
- 4 independence
- 5 independence

3 Implementation

We implemented this project in C, hoping the result will be fast and efficiently. We enjoyed knowing exactly where certain arrays and variables are going to be laid out in memory. In the end, we have approximately 1.5k lines of code, including the hash functions, table generation, collision detection, and test code.

Fortunately for us, Thorup and Zhang included the code for tabulation hashing in their 2010 paper on 5-independent tabulation hashing. We were able to model most of our code based on what was included in the paper. We kept the logic behind how the hashes are generated, but made some changes on how the structures are stored in the code. Storing fewer pointers, hoping that will use less memory space and have higher performance.

3.1 Random Numbers

Tabulation hashing requires tables and tables of random numbers in ordering to function correctly. The C language's standard `rand()` function only guarantees up to 15 bits of random bits. However, we needed at least 32-bit or 64-bit for each entry in our random number tables. Thus, we recreated our own version of random number generator by calling the `rand()` function and number of times and shifting the randomly generated bits. Even though the `rand()` function is only a pseudorandom number generator, we thought it should be good enough for our purpose. We made sure to seed the `rand()` function each time we run our program.

3.2 Hash Functions

We had a 5 hash functions. One is a basic multiplicative function and the four other ones are some variation of the tabulation hash function.

3.2.1 Univ2

This is the basic multiplicative hashing. It takes a value to hash, multiply it by a number and then adds another number to generate a 32-bit hash.

3.2.2 Short32

This is a tabulation hashing function. It divides up the 32-bits into 16-bit (`short`) chunks. It has a look up table for each chunk, as well as the sum of the chunks. This requires a total of 3 random number tables.

T0	$2^{16} \times 4$ bytes
T1	$2^{16} \times 4$ bytes
T2	$2^{17} \times 4$ bytes
Total	1 megabyte

Table 1: Space utilized by tables for Short32

T0	$2^{8} \times 2 \times 4$ bytes
T1	$2^{8} \times 2 \times 4$ bytes
T2	$2^{8} \times 2 \times 4$ bytes
T3	$2^{8} \times 2 \times 4$ bytes
T4	$2^{10} \times 4$ bytes
T5	$2^{10} \times 4$ bytes
T6	$2^{11} \times 4$ bytes
Total	32 kilobytes

Table 2: Space utilized by tables for Char32

3.2.3 Char32

This is also a tabulation hashing function. It divides up the 32-bits into four 8-bit (`char`) chunks. There is a look up table for each of the chunks and a few extra table for additional generated characters. This requires a total of 7 random number tables and 7 table look-ups. Some look-ups uses more than 1 random number from the table.

3.2.4 Short64

Short64 is a hash function that divides a 64-bit key into 4 chunks of 16-bits. The actual algorithm is similar to Char32, except this function has much larger tables, even though it has the same number of tables.

3.2.5 Char64

This is the most complicated tabulation hash function we have. It requires 15 lookup tables and also the most number of table accesses. However, since each chunk is only 8-bits the

T0	$2^{16} \times 2 \times 8$ bytes
T1	$2^{16} \times 2 \times 8$ bytes
T2	$2^{16} \times 2 \times 8$ bytes
T3	$2^{16} \times 2 \times 8$ bytes
T4	$2^{21} \times 8$ bytes
T5	$2^{21} \times 8$ bytes
T6	$2^{22} \times 8$ bytes
Total	68 megabytes

Table 3: Space utilized by tables for Short64

T0	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T1	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T2	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T3	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T4	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T5	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T6	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T7	$2^{16} \times (1 + 1 + 0.5) \times 8$ bytes
T8	$2^{11} \times 8$ bytes
T9	$2^{11} \times 8$ bytes
T10	$2^{11} \times 8$ bytes
T11	$2^{11} \times 8$ bytes
T12	$2^{21} \times 8$ bytes
T13	$2^{11} \times 8$ bytes
T14	$2^{21} \times 8$ bytes
Total	≈ 32 megabytes

Table 4: Space utilized by tables for Char64

total size of the lookup tables is actually much smaller than that of short64.

3.3 Collision Resolution

problems we ran into
    counting collisions instead of absolute time (or maybe we can do both)

3.4 Small Improvements

4 Benchmark Results

Compare pure hashing vs tabulation hashing
    compare linear probing
    compare quadratic probing
    compare chaining
    compare between the three
    some analysis on memory access and mention pros ad cons of each

5 Conclusion

summrize what we wanted to find out
    what we did
    and the results we found

References

[1] M. Thorup, Y. Zhang *Tabulation Based 5-Universal Hashing and Linear Probing*, 2010

[2] M. Thorup and Y. Zhang *Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation*, Proc. 15th SODA:608-617 2004.