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 requiring 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 even be used for database indexing. Even lower level computer architectural components like processor caches use ideas from hashing to figure out which cache line to store bits 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 and 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 [1] 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 its 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 and different collision resolutions.

2 Tabulation Hashing

In this section we are going to take a closer look at Tabulation Hashing and see how it fits into the k-independence paradigm of Wegman and Carter [3].

2.1 Simple Tabulation Hashing

The main idea of Tabulation Hashing is to split the key into t r-bit vectors, $x_0, x_1...x_{t-1}$, and use those vectors as lookup indices into prepopulated with random numbers lookup-tables, $T_0, T_1...T_{t-1}$. After all the necessary values from the tables are obtained, the resulting hash value is calculated using a bit-wise XOR operator to combine all of the values.

$$T_0[x_0] \oplus T_1[x_1] \oplus ... \oplus T_{t-1}[x_{t-1}]$$

2.2 Tabulation Hashing Independence

The simple form of Tabulation Hashing, described above, is already 3-independent, as shown in Wegman and Carter in [3]. A function $\vec{t}$ that maps values $x_0, x_1...x_{t-1}$ to the corresponding hash value $T_0[x_0] \oplus T_1[x_1] \oplus ... \oplus T_{t-1}[x_{t-1}]$ is 3-independent as long as T is 3-independent, where $T_1, T_2..T_{t-1} \in T$. However we cannot say the same for 4-independent functions, and since 3-independence is not always enough for some applications, in their 2004 paper [2] Thorup and Zhang describe an efficient way to increase the independency degree of Tabulation Hashing by 1. The basic idea behind 4-independent Tabulation Hashing is to create more bit vectors than there is available from splitting the key, by using the already existing bit vectors. For example, given two existing vectors x_0 and x_1, we can combine them together to derive a new bit-vector $x_2 = x_0 + x_1$. So the hash function using this new method will be as following:

$$T_0[x_0] \oplus T_1[x_1] \oplus T_2[x_0 + x_1]$$

It was later realized and proven in [1], again by Thorup and Zhang, that the same hash functions used for 4-independence can also be used to make Tabulation Hashing 5-independent. They show that to get 5-independent hashing from the old methods, it is only necessary to make the pre-computed table to be 5-independent. That doesn't affect the performance at all, since the everything else stays the same and the tables are precomputed ahead of time.

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.

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

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 total of 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	$2^8 \times 2 \times 4$ 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

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

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

T0	$2^8 \times (1 + 1 + 0.5) \times 8$ bytes
T1	$2^8 \times (1 + 1 + 0.5) \times 8$ bytes
T2	$2^8 \times (1 + 1 + 0.5) \times 8$ bytes
T3	$2^8 \times (1 + 1 + 0.5) \times 8$ bytes
T4	$2^8 \times (1 + 1 + 0.5) \times 8$ bytes
T5	$2^8 \times (1 + 1 + 0.5) \times 8$ bytes
T6	$2^8 \times (1 + 1 + 0.5) \times 8$ bytes
T7	$2^8 \times (1 + 1 + 0.5) \times 8$ 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	$\approx$ 32 megabytes

Table 4: Space utilized by tables for Char64

3.2.6 UnivString

In this function we use the multiplicative Univ2 hash function mentioned above to hash strings into 32-bit values. The strings are hashed recursively, as suggested in [1]. If the string is less than or equal to 4 characters (32 bis), then we just use Univ2, otherwise we divide the string into 2 substrings and call the function recursively. We use XOR to combine the results of the recursive calls.

3.2.7 CharString

This function is very similar to UnivString described above. The only difference is that we Char32 hash function in order to obtain the hash values of strings less than or equal to 4 characters.

3.3 Collision Resolution

For our project, we implemented three different type of collision resolution for comparison of performance. One is the basic linear probing, which just checks sequential array indices if the one a key is hashed to is already occupied. The quadratic probing looks at the hashed index plus the square of the number of collisions thus far. Lastly the Chaining has a linked-list of values at array index. New links can always be appended at the end of the linked-list. For both linear and quadratic probing, we store the actual value of the number we hashed in the array. The hash table for chaining stores the pointer to the first element in the linked-list.

3.4 Small Improvements

We kept performance in mind as we coded our project. One measurable improvement we were able to make is to change the memory access pattern for hash functions that sometimes use a pair of random numbers from one table for each index. For example the Short64 and the Char32 both have two random numbers associated to each chunk of data. One way is to have two tables for the chunk of bits to index into. However, that requires two memory look ups. Since the tables are fairly large, it is impossible for those two numbers be in the same cache line. The other way is to have a single look-up table that is twice the size of the number of index and just use `index*2` and `index*2+1`. Those two index in a continuous array is almost guaranteed to be on the same cache line, thus reducing the number of times we have to actually go out to memory to retrieve values. This small changed showed a 20% performance improvement for the short64, which we believe is significant and important to watch out for.

4 Benchmark Results

After programming all the functions, we were finally able to start looking at what interested us in the first place. We were careful in making sure our code would work on different machines and hoped to be able to do benchmark test on various computers. Unfortunately, we were low on time to get access to faster machines and to collect data from multiple machines, especially since the process of collecting data and generating graphs can be tedious and quite time consuming.

In the end, we only tested all combination of our hash functions on a 3-year old MacBookPro. This machine has a 2.53 GHz Intel Core 2 Duo processor. This processor has a L2 Cache of 3MB. This small cache size can limit the performance of the tabulation hashing, especially for functions that require a larger table size. It also has a 8GB 1067 MHz DDR3 RAM and which should be more than enough to fit all tables without paging to disk.

For the analysis of the hash functions, we decided to look at both the number of collisions and the overall time taken to better understand the behavior of the hash tables at different load factors. The benchmark tests done below are all measured with attempting to fill a hash table of size 1000 with randomly generated numbers.

4.1 Collisions

First will looking at the performance of each of the hash functions with the different collision resolution. Then we will compare the performance of the collision resolutions with the tabulation hash functions. We also collected data on both the average collision count as well as the running maximum number of collisions as elements are added into the table. For the average number of collisions, we took the average of 5 runs for each combination of hash functions and collision resolution.

It is important to remember the number of collisions do not necessarily represent the performance, because each hash function uses a different number of operations and memory accesses.

4.1.1 Linear Probing

From Figure 1 shows collision data for creating hash tables. There is no visible difference between the different hash functions in terms of number of collisions. This show that all the hash functions are random enough for the 1000 elements.

4.1.2 Quadratic Probing

Figure 2 also shows that there is no significant between the different hash functions.

4.1.3 Chaining

While the number of collisions for the hash tables with chaining is significantly lower, Figure 3 shows a similar behavior between the different hash functions.

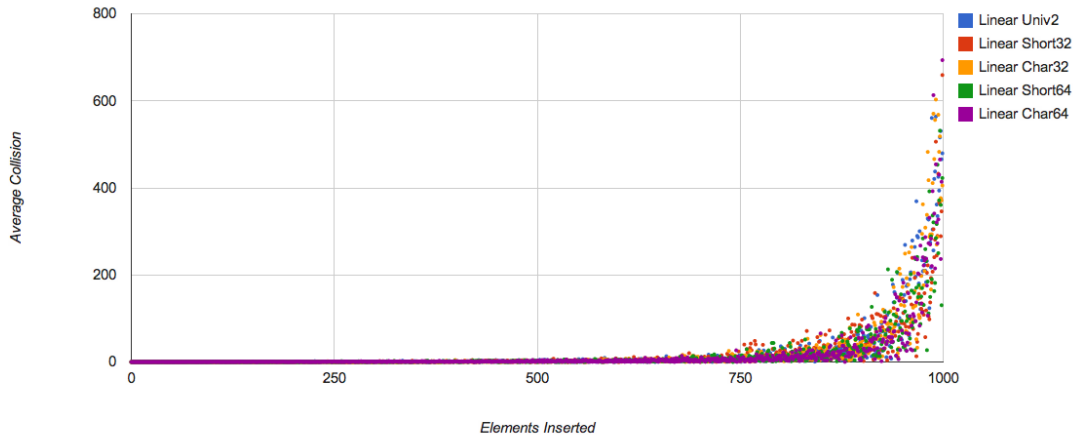
4.1.4 Overall

The comparison between the different collision resolution is much more interesting. The results are shown in Figure 4. Throughout the benchmark test, chaining seems for perform much better than the other two ways of collision resolution. It grows at much more linear rate. Aside from chaining, quadratic probing seems to be a little bit more efficient than linear problem. This is probably due to the fact that quadratic probing can be more resilient against clusters of occupied slots, by being able to jump over the clusters more quickly. However, the average collision count spikes up to drastically at a load factor of over 0.95. This is caused by the quadratic nature of the probing. Unlike linear probing, quadratic probing can not guarantee to find an empty slot with number of probes less than the table size. In order for the program to not stall, we had to cap the number of probes for quadratic probing to twice the size of the table.

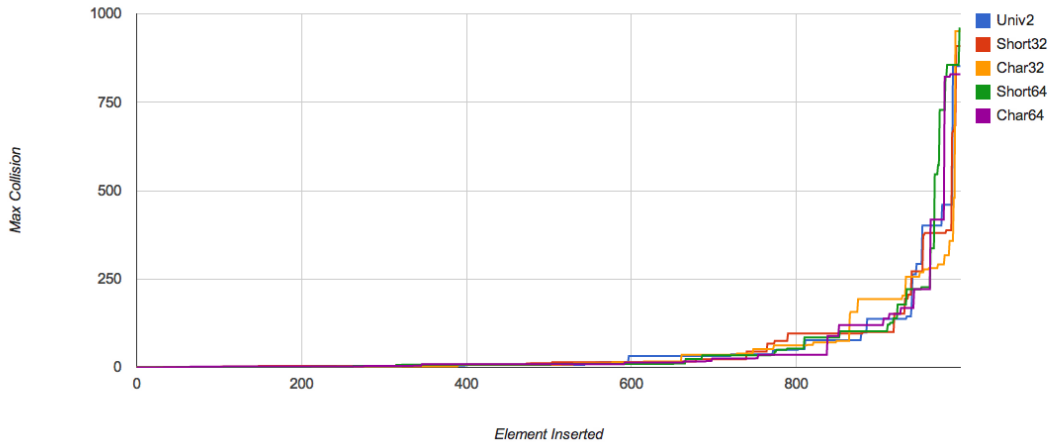
4.2 Time

We measured the amount of time it took the hash functions to fill a hash table size of 1000 with random numbers. We ran that 100 times each and plotted the cumulative time distribution. The timing is done with the `clock()` function so we can accurately measure the number of cycles used. Furthermore, we commented out any unnecessary calculations and branching statements that were used for measuring the other metrics in order to have accurate timing data.

Figure 6 shows the distribution of the timing data we collected.

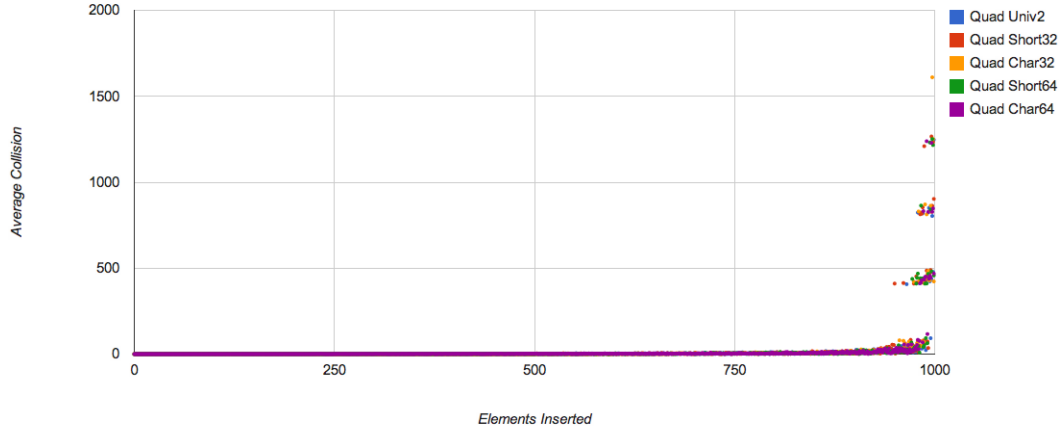


(a) The average of number of collisions for different hash functions

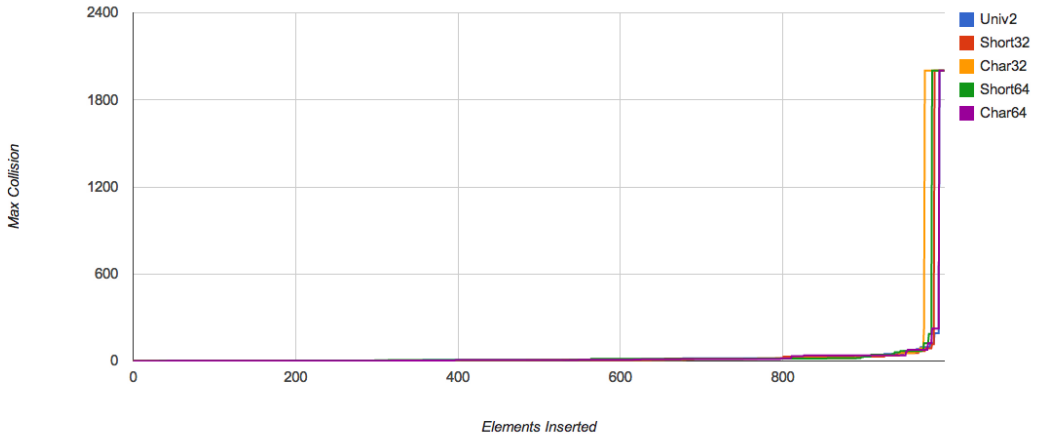


(b) The running maximum number of collisions for different hash functions

Figure 1: Collision graphs for hash table with linear probing

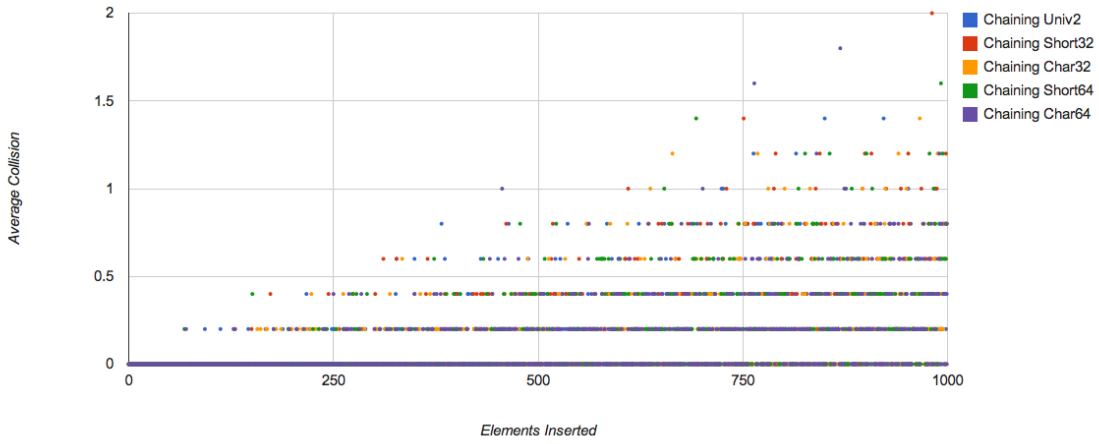


(a) The average of number of collisions for different hash functions

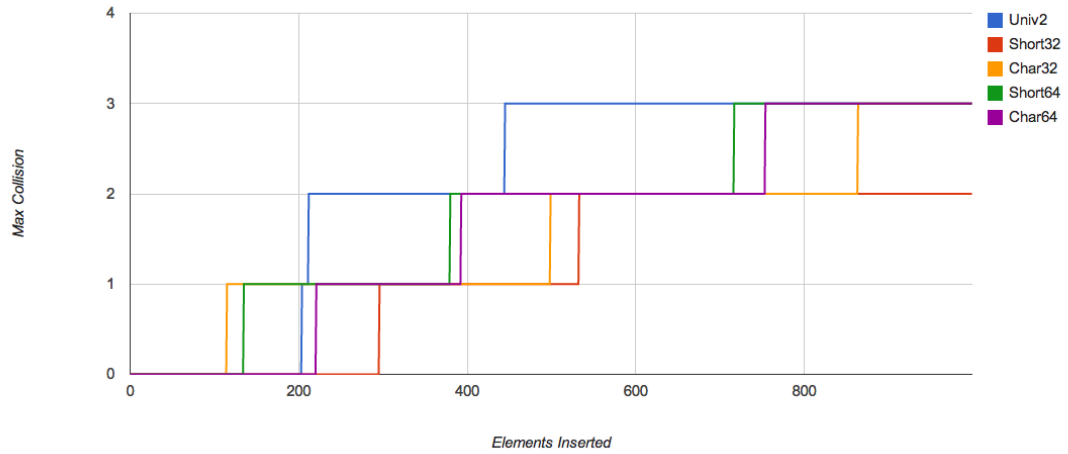


(b) The running maximum number of collisions for different hash functions

Figure 2: Collision graphs for hash table with quadratic probing



(a) The average of number of collisions for different hash functions



(b) The running maximum number of collisions for different hash functions

Figure 3: Collision graphs for hash table with chaining

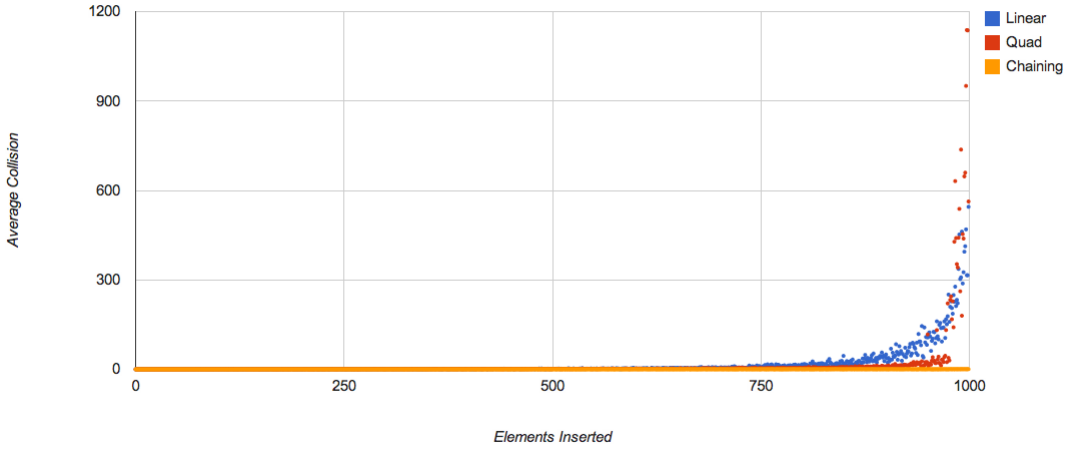


Figure 4: The average number of collision with different collision resolution.

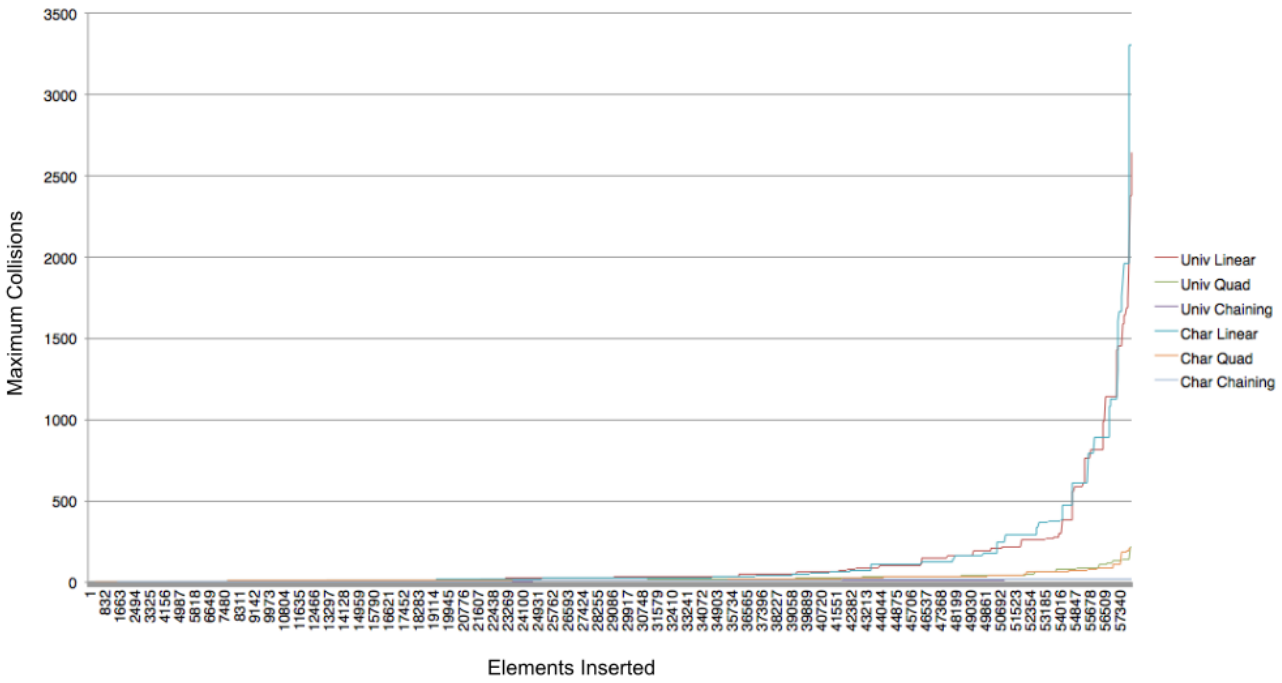


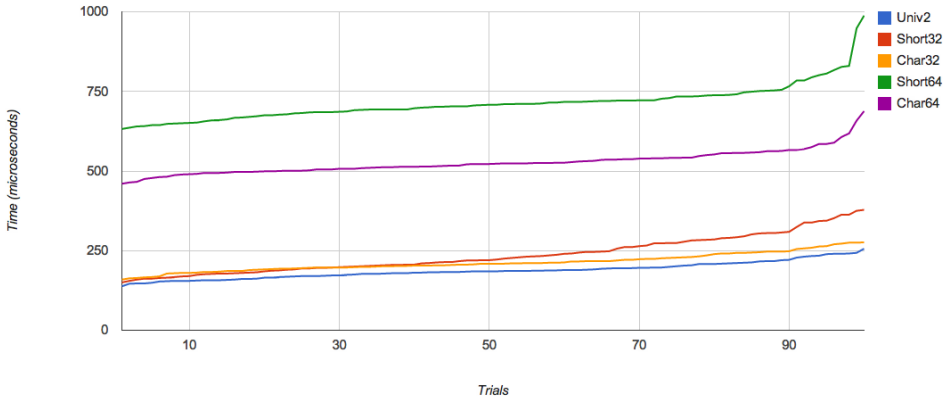
Figure 5: The running maximum for hashing a list of dictionary words.

We noticed a interesting trend for these graphs. The hash functions that use 16-bit chunks seem to have a wider range of timing distribution. This is not as obvious for the 64-bit version of the function. However, we can see the effect more clearing in the 32-bit versions of the functions. The lower bound of the `short32` is lower than that of `char32` but the upper bound is also higher than `char32`. Intuitively, `short32` should be faster because it only uses 3 memory accesses, while `char32` uses 7 memory accesses. This trend makes more sense if we look at size of the tables. The size of all the tables for `short32` takes 1 megabyte according to Table 1. The space needed for `char32` only 32 kilobyte. While both of them will fit comfortably in L2 cache, the `char32` can fit a larger percentage of its tables in L1 cache, which is much faster. Despite needing fewer table accesses, `short32` is more likely to need something outside the L1 cache and thus taking a longer time to hash everything. However, if we had a bigger L1 cache, the performance of `short32` maybe actually be better than that of `char32`.

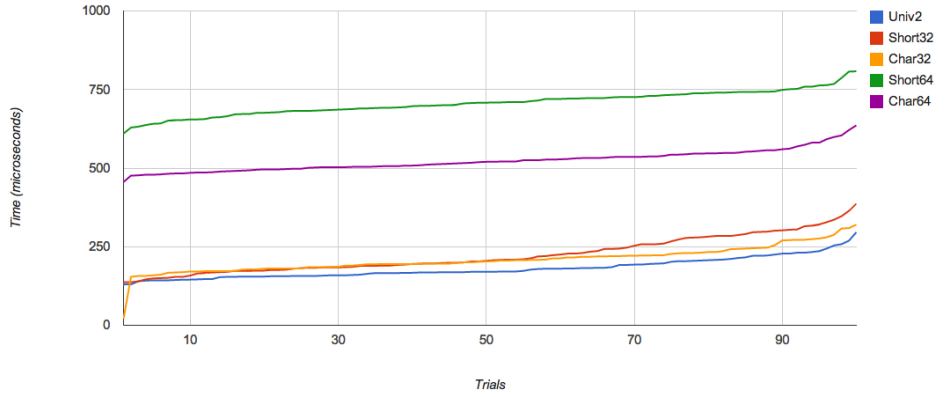
We also plotted the median times for the different hash function and collision resolution, as shown in Figure 7. The three 32-bit functions (Univ2, Short32, and Char32) has fairly similar performance. The `univ2` being slightly faster, but not by very much. This is probably because this function does not need to access any memory other than the hash table, which fits in small caches fairly well. On the other hand, the two 64-bit functions are much slower. Both of those functions take a large about of memory for its tables. According to Table 3, `short64` takes 68 megabytes, and `char64` takes about 32 megabytes according to Table 4. Both of them are no where close to fitting in the small L2 cache on the process we are using. Thus, aside from having more memory accesses, the cache performance is also very poor.

For all the hash functions chaining performs better than both linear and quadratic hashing. However, chaining does have a higher space requirement, since it needs an additional space to store a the value and pointer to the next link in the linked list. If the space is not a constraint, chaining might be the more effective collision resolution. Excluding chaining, quadratic probing seem to perform better than linear probing. Even though quadratic probing does not have the benefit of checking continuous locations in memory, the reduction in number of collisions has made it more efficient. Keep in mind these timed intervals also include the spike of collisions when the load factor is above 0.95. Thus, if the load factor was kept below that, we might see a bigger gap between linear and quadratic probing.

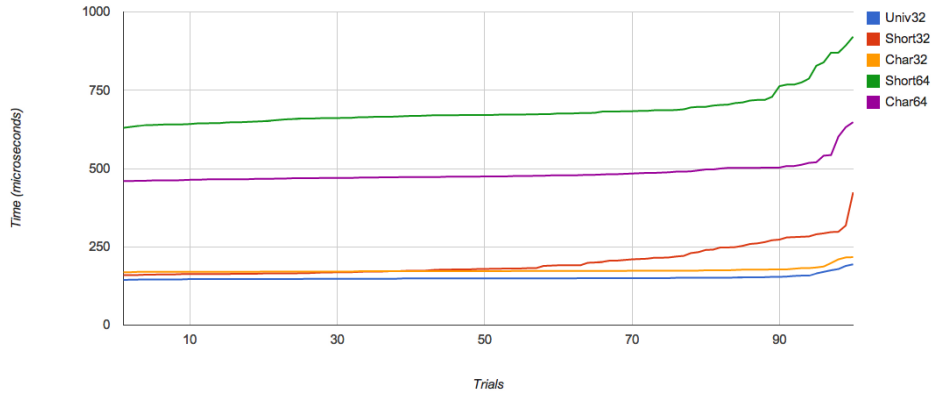
To summarize, the 32-bit functions are much more performant on personal computer. While the 64-bit might function almost just as well on a more powerful machine that has much larger cache size, the requires too much space to sufficiently fit in the cache of a normal processor. Furthermore, we did not see an clear reduction in collision rate while filling up hash tables of 1000 between the 64-bit and 32-bit functions. Thus, unless more than the hash table size is greater than 2^{32}, the 32-bit functions might be sufficient. Between the collisions, we found the chaining to be most performant. However, quadratic probing is a



(a) Linear Probing



(b) Quadratic Probing



(c) Chaining

Figure 6: Distribution of time for filling a hash table for different collision resolutions and hash functions

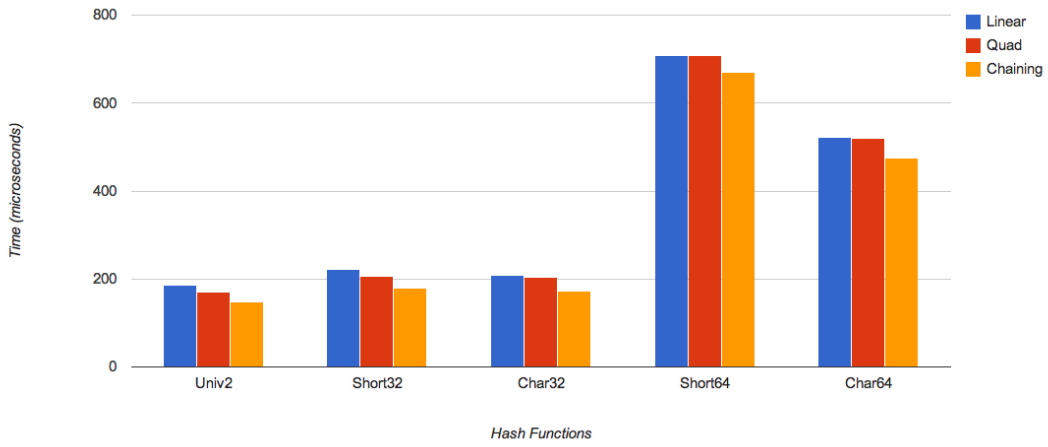


Figure 7: The median time span used to fill hash table.

better option if space is limited.

Unfortunately, we were unable to reproduce Thorup and Zhang's results [1] that tabulation hashing sometimes had better performance than universal hash functions with the tests we've done. The universal hash functions seemed to always have slightly better performance in our tests. However, results could be very different on a processor with better caches.

5 Conclusion

Thorup and Zhang analyzed in [1] how 5-independent Tabulation hashing's performance compared to the most basic Multiplication Based hashing method. In their paper they only used one type of collision resolution methods: linear probing. Following their results, we wanted to see how their results and claims project onto a larger set of experiments. We decided to compare the performance of Tabulation hashing described in their paper with the multiplication based hashing using 3 different collision resolution methods: linear probing, quadratic probing, and chaining. In the paper it is also mentioned that the same functions can be used to hash arrays of characters, or strings. We also decided to analyze how string hashing using 5-independent Tabulation hashing compares to hashing using multiplicative hashing. We implemented and ran a number of experiments to be able to compare all the methods.

what we might do next -maybe look at more specific timing data for when the table is really full -also if we had time, we could have further optimized chaining by allocting many links in started instead of calling malloc 1000 times, which could be fairly slow.

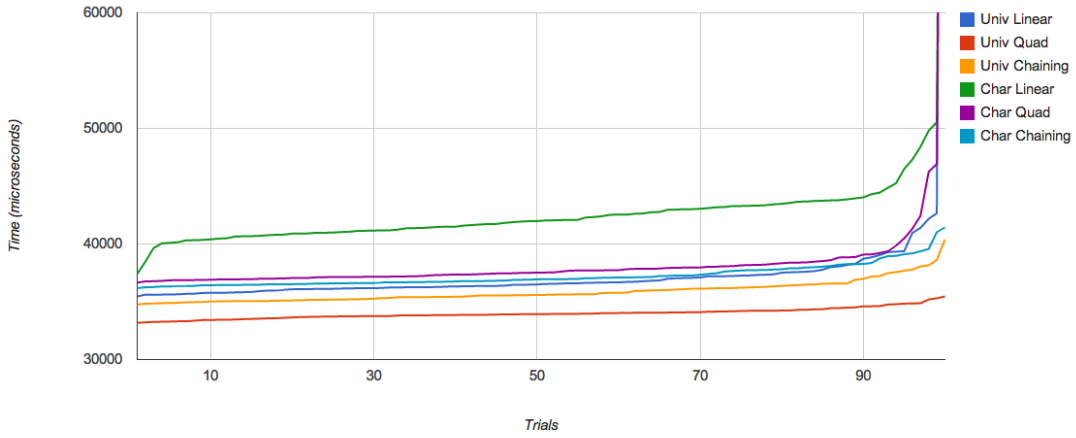


Figure 8: The median time span used to hash all the words.

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.

[3] Mark N. Wegman and Larry Carter *New classes and applications of hash functions.*, Journal of Computer and System Sciences, 22(3):265 279, 1981. See also FOCS'79.