Amazoff HashMaps

Gideon Witchel

June 2022

Abstract

The purpose of this paper is to explore profiling tools and enhance programming knowledge. This study analyzed and compared four implementations of a hash map in c++ for use in an online store; the standard library map, the standard library unordered map, a chained map, and an open map. The ideal maximum load factors for read times were 0.7 for open map and 1.9 for chained map. STD Map was very resource inefficient and slow. Chained Map was consistently outperformed by STD Unordered Map, which is a better implementation of the same hash map type. Open map had the best cache performance and the fastest read speeds. STD Unordered Map was a close second in read times, had the fastest write speeds, and had the lowest memory consumption.

Chapter 1

Amazoff HashMaps

1.1 Introduction

Amazoff, Amazon's newest competitor, is looking to open their online store. They have a website mostly up and running, but they don't have a way of storing their inventory. They have dedicated resources to testing different hash map implementations to choose the best one to store their inventory. The purpose of this study is to analyze and compare four implementations of a hash map in c++.

1.2 Design

The first hash map is *std::map*. This is an ordered map and uses a red black tree to store data.

The second hash map is *std::unordered_map*. This uses a hash map with separate chaining to store data. Specifically, *std::unordered_map* contains an array of buckets, each with the head of a linked list.

The third hash map is *ChainedMap*. This uses a hash map with separate chaining. Specifically, *ChainedMap* contains a *std::vector* of buckets that each contain *std::lists*. Each *std::list* contains a list of *KeyValuePair* structs with one key and one value.

The fourth and final hash map is *OpenMap*. This uses a hash map with open addressed chaining. Specifically, *OpenMap* contains a *std::vector* of elements and handles collisions using linear probing, which is the most cache efficient and is guaranteed to pass every item at a given index exactly once. Removal is handled via valid bits in each Key Value Pair in conjunction with a *std::vector* of bools which indicate if a given index in the hash map has been filled in the past.

All of these hash maps support the basic public functions defined in the *HashTable* parent class: *get()*, *put()*, and *remove()*.

Hash map variables such as *mapSize*, *maxLoadFactor*, and *mapType* were passed in through command line. A script called and recorded performance data from a series of different hash maps using *perf*, *std::chrono*, and *getrusage*, aggregating results in a *.csv* format that could be easily analyzed.

1.3 Implementation

1.3.1 Workload and Data Collection

Hash maps tend to be read heavy data structures, so benchmarks were read heavy to simulate a real-world workload. During benchmarks, a hash map was constructed and filled with half of the keys and values. Keys ranged from $0..mapSize \div 2$, increasing by 1 each time, and values were equal to $(key + 1) * (key + 3)$. Then a random key value pair was read from the map $10 * mapSize \div 2$ times. Then the remaining keys were added to the map, ranging from $mapSize \div 2..mapSize$, increasing by 1 each time, with values equal to $(key + 1) * (key + 3)$. Then a random key value pair was read from the map $10 * mapSize$ times. For example, in a map of size 10, 5 elements would be added, then 50 would be read randomly, then 5 more elements would be added, and then 100 would be read randomly.

Performance data was collected from the chrono standard library, the getrusage library, and the perf command line tool. After an initial round of testing, it became clear that the most important data points were *graph type*, *number of items*, *max load factor*, *final load factor*, *milliseconds to fill*, *milliseconds to read*, *system time*, *page faults*, *max resident set size*, *CPU cycles*, *instructions*, *L1 dcache loads*, *L1 dcache load misses*, *LLC loads*, *LLC load misses*, *branch loads*, and *branch load misses*.

1.3.2 Maximum Load Factors

Before comparing map types, it was important to determine the optimal *MaxLoadFactor* for *ChainedMap* and *OpenMap*. The load factor of a given hash map is equal to the number of Key Value Pairs ÷ the number of buckets. In *OpenMap*, it is calculated by the number of vector locations that have contained a value currently or in the past (within the current vector size) instead of just the number of locations that currently contain a value, because the efficiency of *get()* scales with that number. *ChainedMap* and *OpenMap* both start with 4 buckets and double their size whenever their current load factor reaches a given threshold (*MaxLoadFactor*). That threshold must be between 0 and 1 for *OpenMap*, because it cannot over fill.

To benchmark load factors, data was collected on hash maps with 1,000,000 elements. *ChainedMap* was measured at every *MaxLoadFactor* between 0.1 and 10, with increments of 0.1. *OpenMap* was measured at every *MaxLoadFactor* between 0.01 and 0.99, with increments of 0.01.

1.3.3 Comparing Map Types

To compare the different map types, data was collected for all four map types at different sizes ranging from 100,000 to 10,000,000, with increments of 100,000 (400 total tests). After data collection, the L1 Miss %, LLC Miss %, Branch miss %, and instructions per cycle were calculated for each test.

1.4 Evaluation

1.4.1 Spikes and Dips

Both Chained Map and Open Map resized when the number of key value pairs ÷ the number of buckets reached a certain value (*MaxLoadFactor*). This doubling in size caused "doubling

points" in the data for chained maps and open addressed maps, where there was a sharp spike or dip in data. The spikes and dips in these saw-shaped trends fell exactly when $NumberOfItems \div 2^x$, where x is any integer, $= MaxLoadFactor$. This is because 2^x represents the size of the map at any point.

When studying the effects of max load factors on performance, these "doubling points" are actually "halving points". As the max load factor increases, the number of buckets gets smaller, because the map will have doubled one less time. When studying the effects of map elements on performance, the doubling points indicate actual doubling in the number of buckets.

Interestingly, STD Map did not have any spikes or dips, and STD Unordered Map only had small ones occasionally. STD Map's lack of sudden changes can be attributed to its implementation as a tree. Trees do not have buckets, so they increase in proportion to new items, without sudden jumps or dips.

1.4.2 Chained Map Max Load Factor

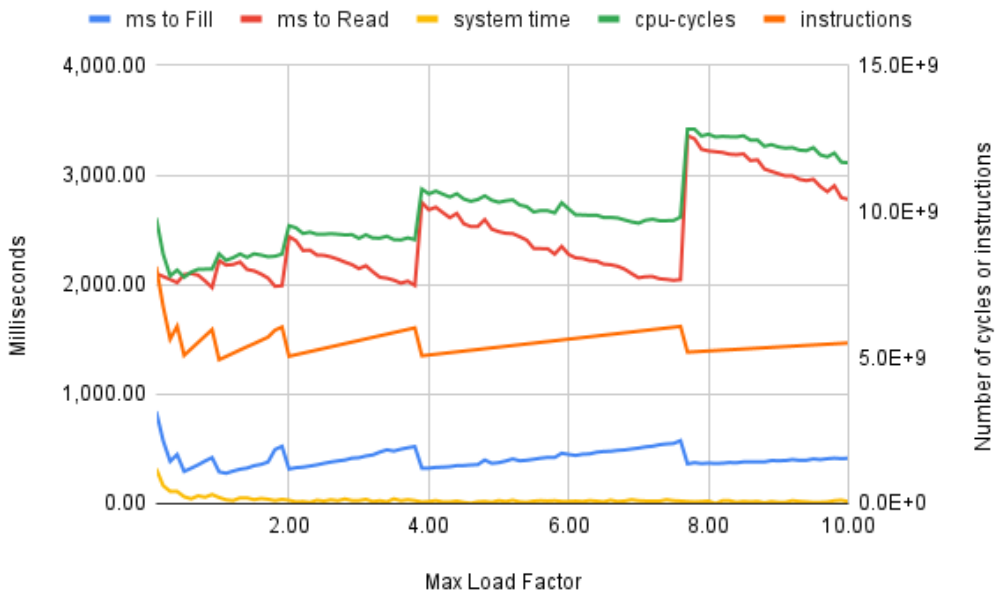

Figure 1.1: Chained Map Max Load Factors vs Milliseconds to Read and Fill, System Time, CPU Cycles (right), and Instructions (right)

Chained maps had read speeds that generally moved downward with increasing max load size, but sharp increases at halving points at max load factors of 0.9, 1.9, 3.8, and 7.6 meant that a higher max load factor led to higher read times overall (Figure 1.1). The sharp increases in read time at halving points occurred because there were significantly fewer buckets, leading to longer linear search times through linked lists. The general decreasing direction was unexpected. The max load factors between halving points ended with the same number of buckets, so it is unclear why the read times would be any different, as the

distribution of items should be the same. This may be due to the read operations halfway through writing.

Write speeds generally increased with increasing max load factor, but dips the same halving points led to a relatively unchanging speeds (Figure 1.1). Halving points led to temporarily faster write times because the lower number of buckets meant easier array access; however, it is unclear why the longer linear searches would not cause a spike in write times. The generally increasing trend was not caused by the final state of the map (which would have the same number of buckets between halving points); rather, a higher max load size means the map would wait to double its size until it was more populated, leading to more linear search time during writes.

The trends in instructions in Figure 1.1 could indicate many things; that read operations took significantly fewer instructions than write operations, that doubling in size was very instruction expensive, and/or that linear searching was not instruction expensive. The trends in CPU cycles likely match the time to read simply because that was what took up the most time; CPU cycles and time are linked.

The max resident set size spiked downwards at halving points because the removal of buckets freed space from memory. Max load factors closer to 0 caused doubling exponentially more often than max load factors further from 0, leading to skyrocketing memory usage at max load factors close to 0. The max resident set size gradually increased between halving points, but the cause of this is unclear. The max resident set size should be representative of the final state of the map, which should not change between halving points.

L1 and LLC loads stayed relatively constant across max load factors, with a general trend upwards but sudden spikes down at halving points. Cache loads increase with increasing linear search times, as each pointer must be loaded into cache individually. Increasing the max load factor causes more linear searches during writing, which would explain the general upwards trend. The dips at halving points does not make sense; fewer buckets should mean more linear searches and therefore more cache loads. L1 and LLC misses increased as the max load factor increased, with a general downward direction but sudden upwards spikes at halving points. Cache misses generally trend with cache loads, so it is unclear why these misses are inverted.

From these results, it appears that the optimal max load factor for read times would be directly before a doubling point, or directly after if optimizing for write times. However, any value lower than 1 would be too memory inefficient, and any value above 8 would be too cache inefficient. As this use case is read heavy, the ideal load factors would either be around 1.9 or 3.9. Keeping in mind that these points will change with the number of elements, it seemed that the smaller value (1.9) would be most consistently efficient in comparative tests, especially considering the test data would not go above 10 million elements (so the extra memory taken up was not a concern).

1.4.3 Open Map Max Load Factor

Open addressed maps showed similar saw-shaped trends but had more flattened data. Read and write speeds, CPU cycles, instructions, and system speeds were generally constant across central max load factors. Low max load factors caused excessive doubling in size, leading to massive empty sections of array which decreased efficiency. Very large max load factors caused excessive collisions, leading to long linear probing times.

The max resident set size was flat between halving points, but spiked downwards at halving points at 0.06, 0.11, 0.24, 0.48, and 0.95 max load factor. This is a direct representation

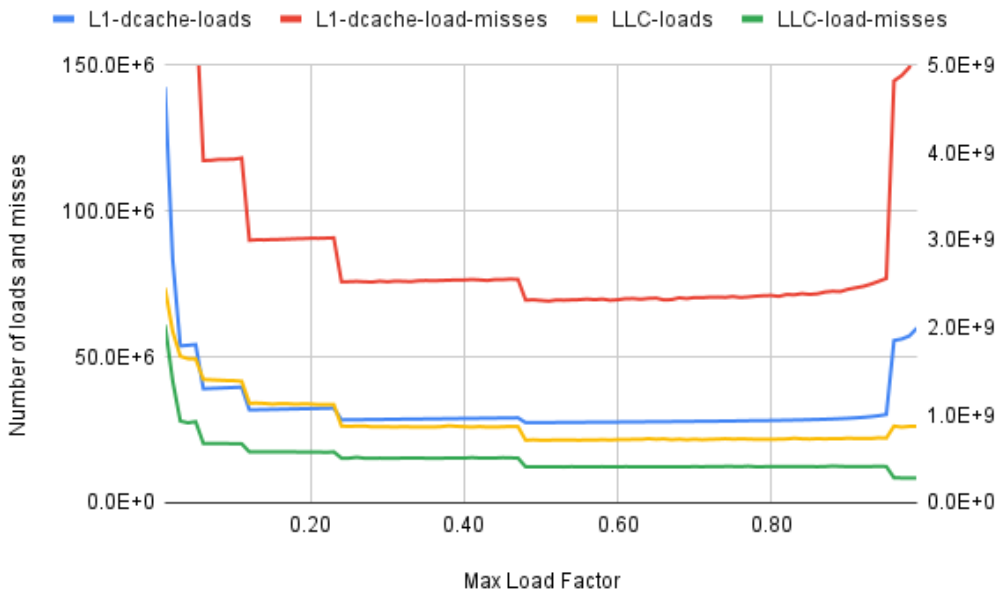

Figure 1.2: Open Map Max Load Factors vs L1 dcache Loads (right), L1 dcache load Misses, LLC Loads, and LLC Load Misses

of the size of array that stores data for open addressed maps; when halving the map size, the amount of memory taken halves. There was no change between halving points. Max load factors closer to zero cause doubling exponentially more quickly, leading to skyrocketing memory usage close to zero max load factor.

L1 and LLC loads and misses remained relatively straight between different max load factors between halving points, spiking down slightly at halving points and then spiking up at 0.95 max load factor (Figure 1.2). The spikes down were caused by decreasing array size; a smaller array can be loaded into cache with fewer loads and therefore with fewer misses. The spike up after 0.95 max load factor was due to general inefficiency with many collisions and longer linear searches.

From these results, it appears that the optimal max load factor for any application would be somewhere between 0.6 and 0.8. There are not significant changes in performance between those values, so 0.7 seemed like an appropriate value to use in comparative tests.

1.4.4 Comparing Maps

As in the data from max load factors, there were doubling points that corresponded to sudden spikes or dips in saw-shaped trends in chained and open addressed maps. Chained maps' doubling points were at 900,000, 1,900,000, 3,900,000, and 7,900,000 items in this data, corresponding with its max load factor of 1.9. Open maps' doubling points were at 700,000, 1,500,000, 30,000,000, and 5,800,000 items for this data, corresponding with its max load factor of 0.7.

Open Map consistently had the highest read performance due to superior cache locality.

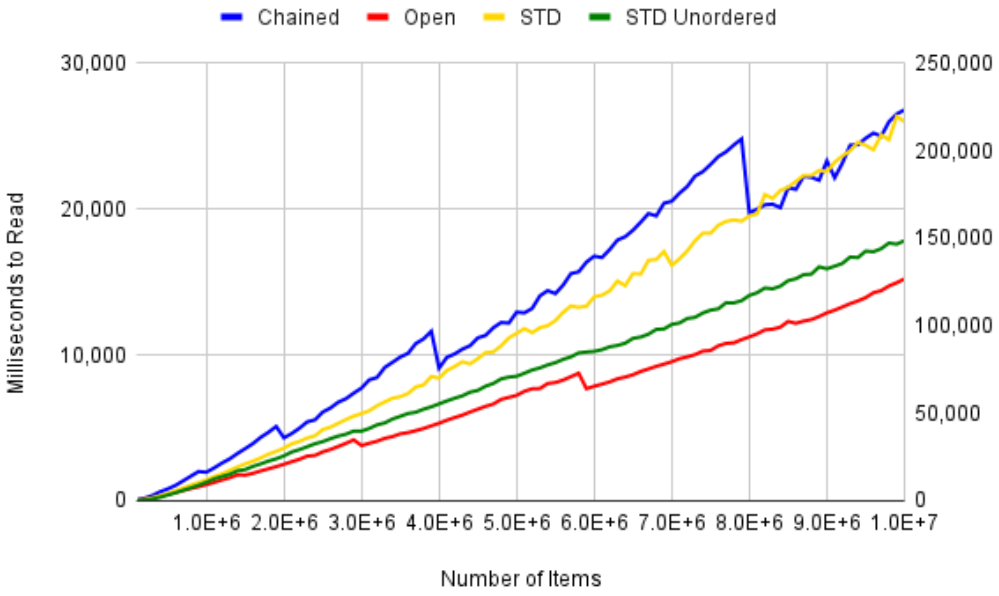

Figure 1.3: Number of Items vs Milliseconds to Read in Chained, Open, STD (right), and STD Unordered

Notably, when the map size doubles, we see a performance improvement due to fewer linear searches. We see a similar improvement in read latency when doubling Chained Map's size due to decreased linear probing; however, Chained Map's overall performance was worse due to the poor spatial locality of linked lists. STD Map had the worst read performance, around 10 times slower than every other map implementation (Figure 1.3). This was likely because STD Map is an ordered map and is implemented as a red black tree, which is not designed for unordered operations. It is unclear why STD Unordered Map does not have any spikes or dips in read latency.

Both chained and open addressed maps' max resident set size stayed flat when increasing the number of items, increasing in spikes at doubling points (Figure 1.4). This is expected from open addressed maps, as the size of the array is constant between doubling points, but is unexpected from chained maps; as collisions occur and items are pushed to the back of linked lists between doubling points the resident set size should increase. Memory performance in STD and STD unordered maps was better, with STD Unordered Map taking up the least memory. STD Map's resident set size increased linearly in relation to new items (adding items to trees increases their size immediately rather than in chunks), while STD Unordered Map had a combination of gradual increases and spikes when it increased the number of buckets. This was the expected behavior of a chained hash map. The average memory efficiency was about 20 items per set size for STD Unordered Map, 16 items per set size for STD Map, 9 items per set size for Open Map, and 7 items per set size for Chained Map (Figure 1.4).

Open addressed maps and STD unordered maps had very similar L1 and LLC load performance, with loads increasing linearly with added items, because more items means

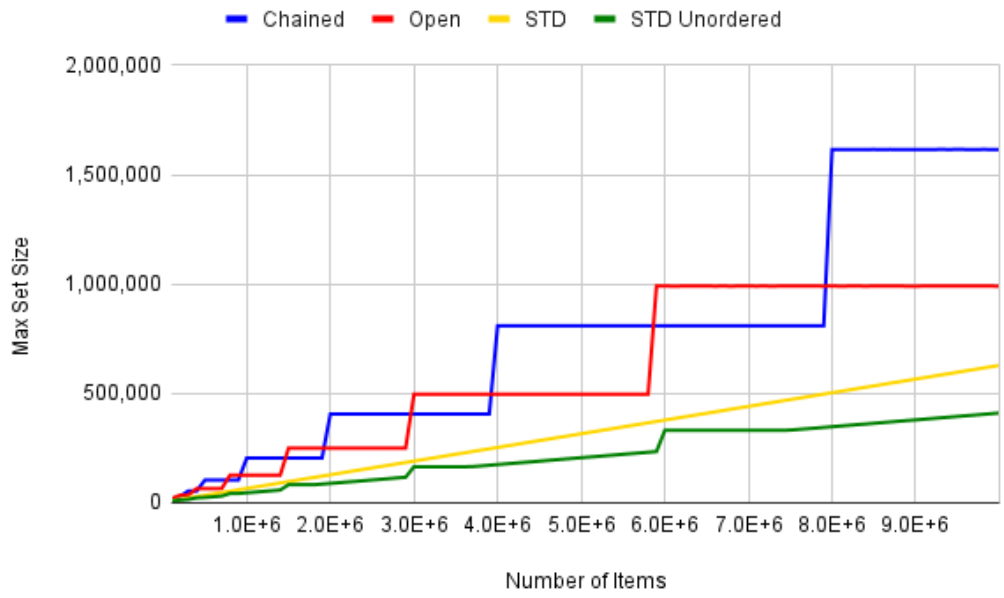

Figure 1.4: Number of Items vs Max Resident Set Size in Chained, Open, STD, and STD Unordered

more things to read into cache. STD unordered maps had fewer L1 loads while open maps had fewer LLC Loads. This may be because STD Unordered Map never had to load the entire map into L1 (unlike Open Map, which loaded the entire map into L1), but Open Map only ever had to load one vector into cache, while STD Unordered Map had to load individual linked lists into LLC for every operation. Chained Map had more L1 loads than Open and STD Unordered Map, increasing linearly and in spikes at doubling points, because more data means more values to load and more buckets means more buckets to load. STD Map had the most L1 loads, about twice as many as open addressed an STD unordered maps, increasing linearly with more values. Chained Map had slightly more LLC loads than Open and STD Unordered Map, increasing with more items but decreasing in spikes at doubling points (fewer linear searches). STD Map had around four times as many LLC loads as every other map and held a 40% L1 Miss % compared to every other map's 10%. This poor L1 performance is likely due to the pointer structure of trees, which requires loading new lines into cache many times in a single read or write. All the maps had a high LLC Miss %, ranging from Open Map's 50% to STD Unordered Map's 80%.

Minimizing branch misses may be how STD Unordered Map maintains efficiency with a chained implementation. While STD Map had a 15% branch miss rate and Open Map and Chained Map had a 2% branch miss rate, STD Unordered Map consistently had a 0.11% branch miss rate.

While cycles are mostly representative of time, instructions per cycle can reveal hardware efficiency. STD maps consistently had the lowest number of instructions per cycle, around 0.1, which is indicative of its poor performance. Both chained maps and STD unordered maps hovered around 0.5 instructions per cycle, although chained showed spiked increases

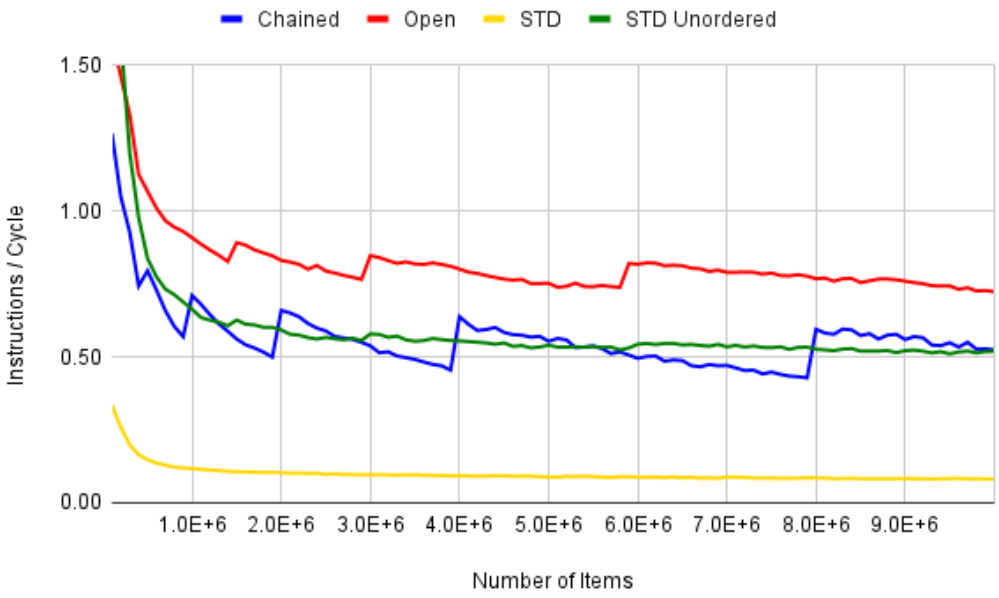

Figure 1.5: Number of Items vs Instructions per Cycle in Chained, Open, STD, and STD Unordered

in instruction efficiency at doubling points, likely due to decreased linear searching. Open maps had the highest number of instructions per cycle, around 0.78, spiking up in efficiency at doubling points, likely due to decreased linear probing. All of the programs seemed to lose instruction efficiency as the size of the hash map increased, because more items causes less locality which decreases performance (Figure 1.5).

1.5 Conclusion

1.5.1 Future Research

If this study were to be repeated or expanded upon in the future, it could be improved by exploring and standardizing in various ways. Analyzing the effects of different key types and value types, different hash functions, different probing strategies, different resizing strategies, and alternate data structures such as binary trees would give a more complete understanding of the ideal hash map. Standardizing or accounting for hardware such as CPU architecture and memory size and speed, software such as the operating system, background processes, and compiler optimizations would lead to more precise results. Separating read data from write data and exploring the effects of removing elements would give a more nuanced understanding of real-world applications. It would also be useful to create worst, best, and typical case tests that involve background processes, interruptions, high collision input sets, and other stress tests.

1.5.2 Recommendations for Company Use

STD Map should not be used. It is cache inefficient, has a high number of branch misses, has a low number of instructions per cycle, and takes at least 10 times as long as the alternative map types to read (and 4 times as long to write) data.

Chained Map has some value for smaller data sets due to its superior statistical analysis tools, but is consistently outperformed by STD Unordered Map, which is a better implementation of the same hash map type.

The main two options would be STD Unordered Map and Open Map. If the only consideration is speed to read, Open Map will be the best choice. It has the best LLC performance and the lowest L1 dcache misses and miss %, although it does load into L1 slightly more often than STD Unordered Map. Most importantly, it has the highest instructions per cycle and the fastest read speeds. However, if a more balanced approach with resource consumption is preferred, then STD Unordered Map will be the best choice. Its fill speeds are at least four times as fast as Open Map, it takes up between half and a quarter the memory that Open Map takes up, and blows every other option out of the water in terms of branch misses. Its lower memory usage also means it will be able to store around four times as much data before crashing when compared to Open Map.

Even with further research and optimization, there will always be significant limitations with any hash map for this use case. They are limited to memory size; both open and chained map implementations crash when approaching 100 million key value pairs, while std map crashes around 250 million key value pairs and std unordered map crashes around 400 million key value pairs on a machine with 16 gigabytes of RAM. Memory is also volatile; any power disruption or process malfunction would cause a catastrophic loss of data. These drawbacks indicate that a better solution might be a database, which operates in long term storage.