Benchmarking Hashing functions

Nayan Sanjay Bhatia
nbhatia3@ucsc.edu

March 20, 2023

1 Introduction

In the modern world, a vast amount of data is generated due to advancements in storage techniques. To retrieve the data quickly and efficiently, databases often use the concept of hashing. Hashing abstracts the mapping of keys to value in O(1) time for the average case. Due to the O(1), it has fast lookup responses and is used in main memory databases. However, in the world of databases, unlike algorithms, we do care for the constant time as well and work to improve it. There are two types of hashing: cryptographic hash and non-cryptographic functions. Cryptographic hashing functions provide security guarantees such as pre-image attack resistance. However, they can have high overheads. Cryptographic properties are unnecessary for applications such as databases and can be avoided. Hence, we focus on testing the non-cryptographic properties. A hash function has two important properties to satisfy: it should be fast and avoid a collision. For this project, we test various hashing functions such as the standard hash library in C++, Abseil Hash, Robin Hood Hash, xxHash and will use testing suites like SMHasher and hash-table bench. Hashing schemes are typically built for DRAM. However, it can have certain problems regarding persistent memory. DRAM has scaling issues, as seen in the Five minutes Rule. Even though Intel optane was killed, there's still hope of new persistent memory technology coming to the market. The contributions of this work are as follows:

- Testing the hash functions by varying load factor for lookup, insertion, deletion, and iteration operation using randomly generated data set with different seeds using hashtable-bench.

- Calculate MiB/sec, cycle/hash, cycle/map and checkout quality problems in hashing functions using SMHasher. We ignore the cryptography's failures and focus on avalanche tests, machine-specific tests, etc.
- Profiling the hashing schemes on the persistent memory and comparing its performance concerning the traditional architecture. Since I do not have physical access to persistent memory, I used Quartz, a performance simulator for Persistent memory software by HP.
- Discussion on the development experience of the different implementations and the pitfalls.

2 Background

2.1 SMHasher

SMHasher is a testing suite designed to test the non-cryptography hash functions for distribution, collision and performance. It is an open-source project by Austin Appleby in 2011 and provides several testing parameters which cover a range of inputs such as string, integers and binary using a synthetic and real-world dataset.

2.2 Hash Table Benchmark

It represents a collection of tests for various hash functions written in C/C++. It measures lookup, insertion, deletion, iteration, etc., using a randomly generated benchmark on different data distributions. The benchmarks use a variety of datasets, including uniform, not uniform, and real-world examples. The results of the benchmarks can be used to compare the performance of different hash table implementations and configurations and identify the best hash function based on the seed value and the specific workload.

2.3 Quartz: A DRAM-based performance emulator for NVM

Quartz is an emulator for emulating a wide range of NVM latencies and bandwidth characteristics of the application. Since all the hash functions

are typically built for traditional memories, we can test the behaviour of hash functions on persistent memory and check out the difference in the performance.

2.3.1 Issues with Quartz installation

When I was trying to run the experiments on Quartz, it kept giving me a series of errors. I tried to debug them and in the end, was caught at this particular error "ERROR: No supported processor found". I ran the lscpu command and the processor running on the CloudLab architecture was Intel(R) Xeon(R) CPU D-1548 @ 2.00GHz which belongs to the Broadwell family. However, to my peril, Quartz only supports Sandy Bridge, Ivy Bridge, and Haswell family of Xeon Processor. Hence, I had to get a new CloudLab node with the processor Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz processor, which belongs to the Ivy Bridge family.

2.3.2 Potential issue with using the simulator

The accuracy of results we get from the simulator might not be entirely correct. In order to evaluate the accuracy of the results from the simulator, there are benchmarks such as wbbench that can help find the error margins but have been skipped due to a shortage of time. Also, the emulator does not evaluate the write-back information in its evaluation model. However, the persistent memory emulator can give a good enough reference frame even after all the pitfalls.

3 Evaluations

To run the benchmarks, the experiments were performed on CloudLab [1], an NSF-funded bare-metal as a service infrastructure. I used the "m054" node for running the Quartz simulator from CloudLab, which has the Intel(R) Xeon(R) CPU $E5-2660v2$ @ 2.20GHz, CPU family: 6, Model: 62, 256GB DRAM, L1d cache: 640 KiB, L1i cache: 640 KiB, L2 cache:5 MiB, L3 cache: 50 MiB, and a 10 GbE network interface. g++ 9.3.0 with -O3 -march=native. I used the "m843" node for running the non-Quartz simulator from CloudLab, which has Intel(R) Xeon(R) CPU D-1548 @ 2.00GHz, CPU family: 6, Model: 86, 64GB DRAM, L1d cache: 256 KiB, L1i cache: 256 KiB, L2 cache:2 MiB, L3 cache:12 MiB and a 10GbE network interface.

g++ 9.3.0 with -O3 -march=native. Ideally, both the evaluations should have been on the "m054" node, however running the simulation takes almost 14-15 hours and I figured out the issue with quartz a day before the report submission. I will run the simulations on the same server and again showcase the evaluation.

3.1 Performance Comparison

```
<mid_string_max_56,uint64_t>,avg_hit_find_without_rehash
<mask_high_bits_uint64_t,uint64_t>,avg_erase_insert_time
<mid_string_max_56,uint64_t>,avg_50%_hit_with_rehash
<mask_high_bits_uint64_t,uint64_t>,avg_miss_find_without_rehash
<mid_string_max_56,uint64_t>,avg_insert_time_with_reserve
<mask_high_bits_uint64_t,uint64_t>,avg_miss_find_with_rehash
<mask_high_bits_uint64_t,uint64_t>,avg_50%_hit_without_rehash
<mid_string_max_56,uint64_t>,avg_iterate
<mask_high_bits_uint64_t,uint64_t>,avg_insert_time_without_reserve
<mask_high_bits_uint64_t,uint64_t>,avg_hit_find_with_rehash
<mid_string_max_56,uint64_t>,avg_hit_find_with_rehash
<mask_high_bits_uint64_t,uint64_t>,avg_50%_hit_with_rehash
<mask_high_bits_uint64_t,uint64_t>,avg_hit_find_without_rehash
<mid_string_max_56,uint64_t>,avg_erase_insert_time
<mask_high_bits_uint64_t,uint64_t>,avg_iterate
<mid_string_max_56,uint64_t>,avg_insert_time_without_reserve
<mid_string_max_56,uint64_t>,avg_construct_time_with_final_rehash
<mid_string_max_56,uint64_t>,avg_50%_hit_without_rehash
<mask_high_bits_uint64_t,uint64_t>,avg_insert_time_with_reserve
<mid_string_max_56,uint64_t>,avg_miss_find_without_rehash
<mask_high_bits_uint64_t,uint64_t>,avg_construct_time_with_final_rehash
<mid_string_max_56,uint64_t>,avg_miss_find_with_rehash
```

Results: link

For most of the test cases, I got the following results:

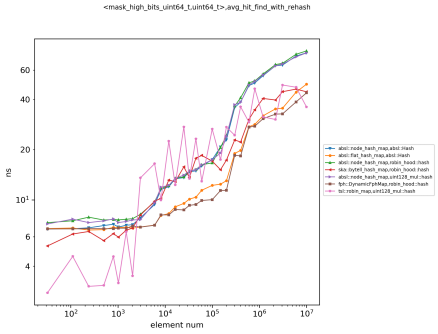
However, for Average erase time comparison, Quartz seemed to be faster in order of magnitude for Robin hood hashing.

3.2 Inference

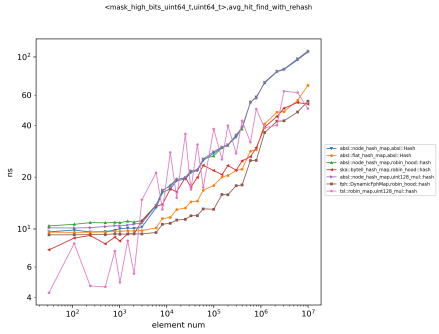
As shown in Section 3.1, the benchmarks without quartz were faster for almost all of the results, even though they were run on a less powerful "m843" node. The "m054" node running the simulator was comparatively slower for the test cases. We get the same pattern for all the test cases. My hypothesis

<mask_high_bits_uint64_t,uint64_t>,avg_hit_find_with_rehash.png

Without Quartz

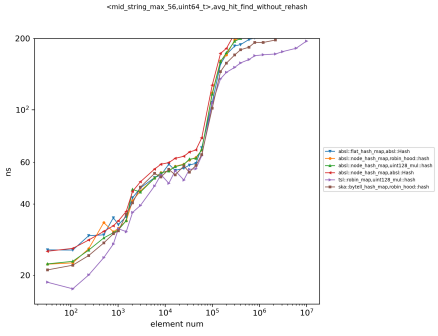


With Quartz

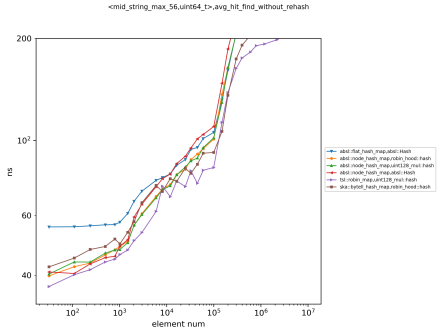


<mid_string_max_56,uint64_t>,avg_hit_find_without_rehash.png

Without Quartz

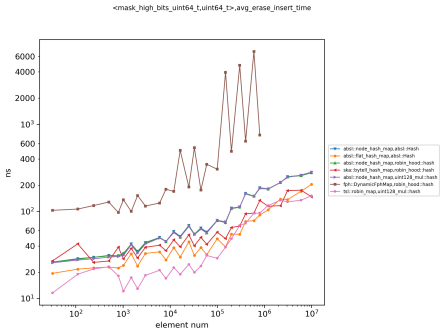


With Quartz

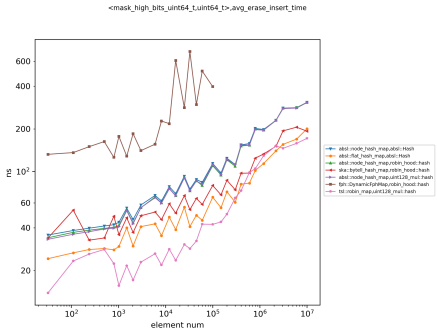


<mask_high_bits_uint64_t,uint64_t>,avg_erase_insert_time.png

Without Quartz



With Quartz



for the slowdown is that there's a chance that the simulator is the issue, and this slowdown is maybe irrelevant if we are using the actual persistent memory. The problem with the test cases is that the ram is big enough, and I have not run big enough simulations to exhaust all the memory. Some slight differences might be because of the different cache sizes between the two nodes. To see a noticeable change or speed up for quartz, I need to run test benchmarks big enough to fetch the table from the disk. However, if you check the Average erase time comparison, quartz was significantly faster than the non-quartz version for the Robin Hood hash. My hypothesis is since the workload requires frequent read and write operations with large amounts of data, persistent memory is causing potentially provides faster access times than traditional memory.

4 Future Work

I will choose an open-source database system like Postgres, replace its hash function with different hashing algorithms like XXHash3 and CityHash, and benchmark its performance with existing benchmarks, such as TPC-C. After replacing the hashing scheme works, I will try to test the same on the Quartz persistent memory simulator and compare the difference. Run persistent memory benchmarks on the Quartz simulator to calculate the error margin

and adjust the readings accordingly.

5 Conclusion

Hashing function for non-cryptography use cases should ideally be fast and avoid the collision. As seen in the evaluation, different hash functions can be used depending on the workload. We see a similar graph pattern when using traditional memory and Quartz. However, in one test case, like average erase insert time, where it repeatedly does one erase after one insert, keeping the map size constant, we see Quartz scale faster significantly. Even though this is only one result and the rest of the results are similar or slower for Quartz, I hypothesise that if we increase the hash table size, there should be a reasonable difference in the performance. Whether persistent memory is faster than traditional memory in a specific scenario depends on the workload and the particular use case. It is essential to evaluate the benefits and limitations of both types of memory to determine which is the best fit for the intended use case; hence, more thorough testing is needed.

References

[1] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.