Benchmarking Hashing functions

Nayan Sanjay Bhatia

Goals

1.Testing hashfunctions for MiB/sec, cycl./hash , cycl./map, size and any quality problems with the hashing functions using SMHasher and hashtable-bench.

2. Profiling the hashing schemes on the persistent memory and comparing its performance concerning the traditional architecture using Quartz.

Background: SMHasher

- Smhasher is a set of test suites and tools developed to help software developers evaluate the quality of their hash functions.
- The tests are designed to detect various types of hash function weaknesses, including collisions, distribution, and avalanche properties

Background: Hash Table Benchmark

This is yet another benchmark for hash tables(hash maps) with different hash functions in C++, attempting to evaluate the performance of the lookup, insertion, deletion, iteration, etc. on different data as comprehensively as possible.

Index	Test items	Notes
1	Insert with reserve	Call map.reserve(n) before insert n elements
2	Insert without reserve	Insert n elements without prior reserve
3	Erase and insert	Repeatedly do one erase after one insert, keep the map size constant
4	Look up keys in the map	Repeatedly look up the elements that are in the map
5	Look up keys that are not in the map	Repeatedly look up the elements that are not in the map
6	Look up keys with 50% probability in the map	Repeatedly look up the elements that have a 50% probability in the map
7	Look up keys in the map with larger max_load_factor	Same as Test Item 4 except that the map is set a max_load_factor of 0.9 and rehashed before the lookup operations
8	Look up keys that are not in the map with larger max_load_factor	Same as Test Item 5 except that the map is set a max_load_factor of 0.9 and rehashed before the lookup operations
9	Look up keys with 50% probability in the map with larger max_load_factor	Same as Test Item 6 except that the map is set a max_load_factor of 0.9 and rehashed before the lookup operations
10	Iterate the table	Iterate the whole table several times
11	Insert and rehash time with larger max_load_factor	The average time used in insert and rehash to construct the time in Test Item 7,8,9

Background: Quartz

- Emulate DRAM-based performance emulator for NVM.
- Quartz can be used to evaluate the performance of various NVM-based technologies and to optimize software algorithms and data structures for NVM-based systems.

Issues with Quartz installation

- Quartz only supports Sandy Bridge, Ivy Bridge, and Haswell family of Xeon Processor.
- My previous architecture was Broadwell.
- Potential issue with using the simulator.

Benchmarking Setup

- **Infrastructure**:
- CloudLab m054 nodes, Intel(R) Xeon(R) CPU E5 – 2660v2 @ 2.20GHz, 256 GB RAM, 256GB NVME, 40GbE ethernet, CPU family: 6, Model: 62, 256GB DRAM, L1d cache: 640 KiB, L1i cache: 640 KiB, L2 cache:5 MiB, L3 cache: 50 MiB

For running Quartz simulator

- CloudLab m843 nodes, IIntel(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.

  For non- Quartz simulator

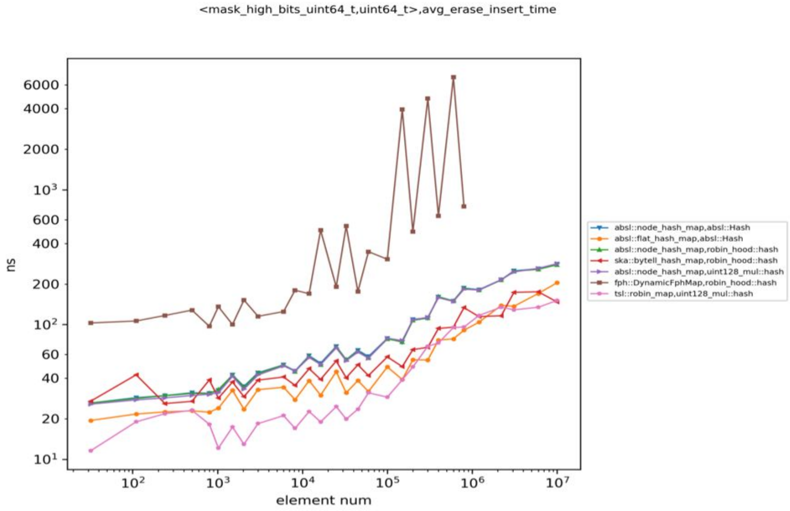
Index	Key Type	Value Type	Notes
1	uint64_t with several split bits masked	uint64_t	The keys have such characteristics: only some bits may be 1, and all other bits are 0. For test data of size n, at most ceil[log2(n)] fixed bits may be 1. e.g. If the key type is uint8_t (it is uint64_t in reality) and the test size is 7, the keys will be generated with the method `rng() & 0b10010001`. The distribution characteristics of such bits can relatively comprehensively examine whether hash tables and hash functions can handle keys that only have effective information in specific bit positions.
2	uint64_t, uniformly distributed in [0, UINT64_MAX]	uint64_t	The keys follow a uniform distribution in the range [0, UINT64_MAX].
3	uint64_t, bits in high position are masked out	uint64_t	The bits in the high position are set to 0. For test data of size n, at most ceil[log2(n)] fixed bits may be 1. For example, if the key type is uint8_t (uint64_t in reality) and the test size is 7, the keys will be generated with the method `rng() & 0b00000111`
4	uint64_t, bits in low position are masked out	uint64_t	The bits in the low position are set to 0. For test data of size n, at most ceil[log2(n)] fixed bits may be 1. For example, if the key type is uint8_t (uint64_t in reality) and the test size is 7, the keys will be generated with the method `rng() & 0b11100000`
5	uint64_t with several bits masked	56 bytes struct	The keys are the same as the distribution of the data 1. The payload is a 56 bytes long struct, which makes the `sizeof(std::pair<key, value>)==64`
6	Small string with a max length of 12	uint64_t	The key type is a string with a maximum length of 12. Both length and characters are randomly generated. The Small String Optimization(SSO) technique may be taken by the compiler.
7	Small string with a fixed length of 12	uint64_t	The key type is a string with a fixed length of 12. The characters are randomly generated. The Small String Optimization(SSO) technique may be taken by the compiler.
8	Mid string with a max length of 56	uint64_t	The key type is a string with a maximum length of 56. Both length and characters are randomly generated.
9	Mid string with a fixed length of 56	uint64_t	The key type is a string with a fixed length of 56. The characters are randomly generated.

Performance Comparison

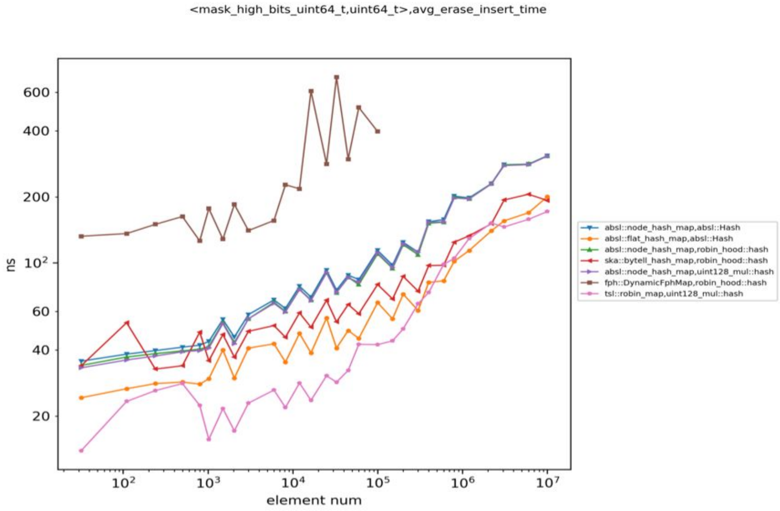
Lower is better

<mask_high_bits_uint64_t,uint64_t>,avg_erase_insert_time.png

Without Quartz



With Quartz

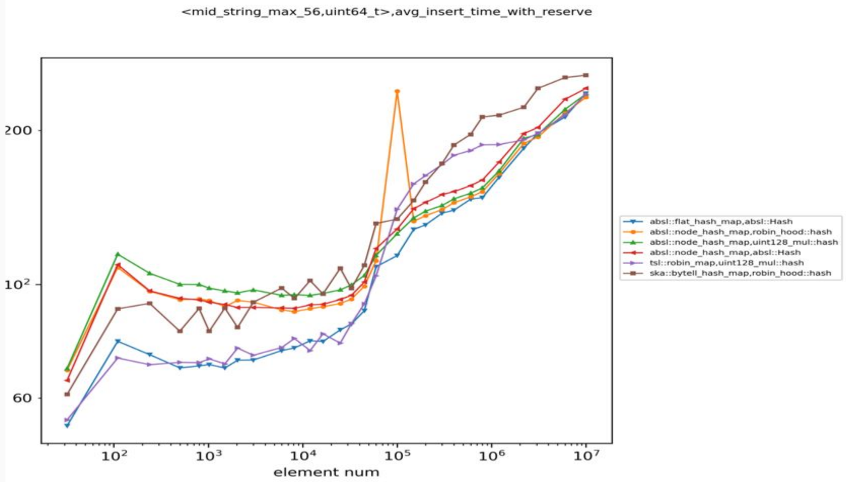


Performance Comparison

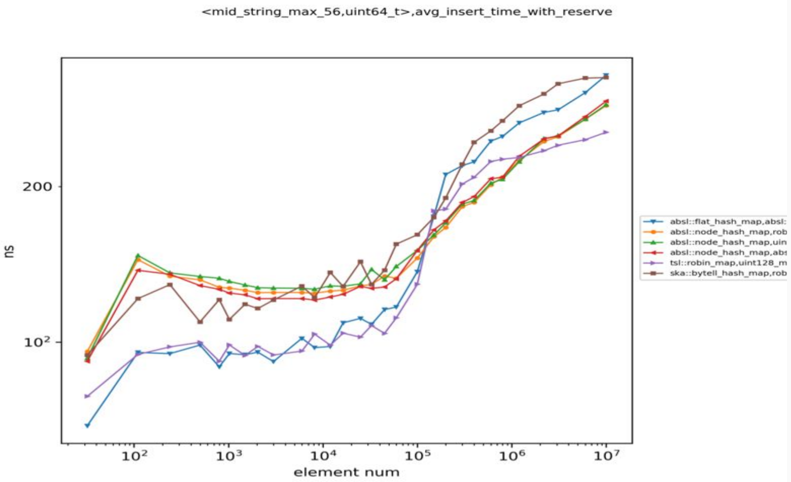
Lower is better

<mid_string_max_56,uint64_t>,avg_insert_time_with_reserve.png

Without Quartz



With Quartz

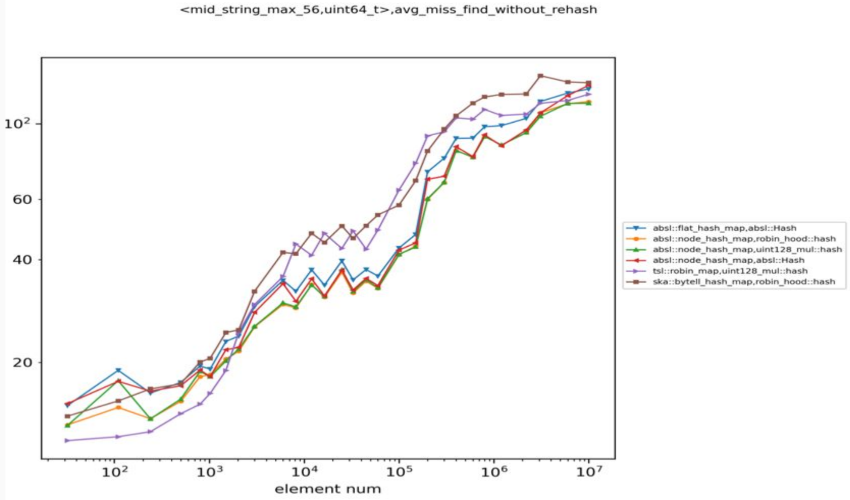


Performance Comparison

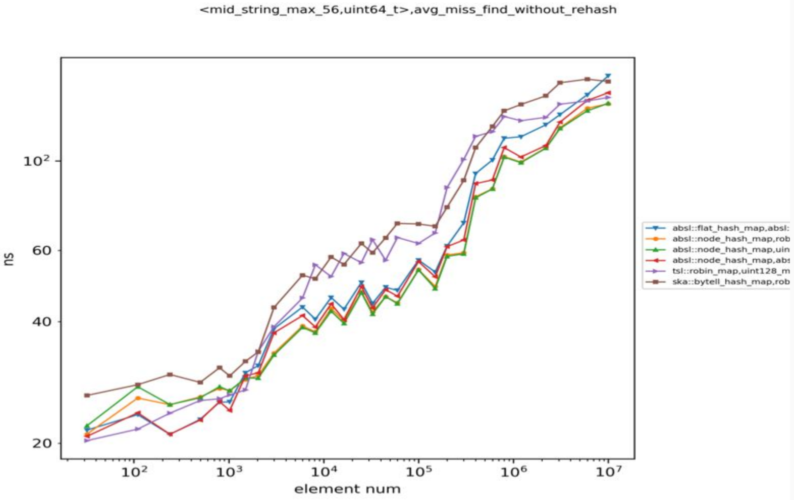
Lower is better

<mid_string_max_56,uint64_t>,avg_miss_find_without_rehash.png

Without Quartz



With Quartz



Key Observations

1. Benchmarks without quartz were faster for almost all of the results
2. Less powerful "m843" node was used for running the benchmarks without quartz
3. "m054" node running the simulator was comparatively slower for the test cases
4. Same pattern was observed for all the test cases
5. Slowdown could be due to the simulator, and may not be relevant if actual persistent memory is used
6. RAM is big enough, and not enough simulations have been run to exhaust all the memory
7. Slight differences may be due to different cache sizes between the two nodes
8. To see a noticeable change or speed up for quartz, bigger test benchmarks are needed to fetch table from the disk
9. Quartz was significantly faster than non-quartz version for Robin Hood hash in Average erase time comparison
10. Persistent memory provides faster access times than traditional memory for workload requiring frequent read and write operations with large amounts of data.

Future work

1. Choose an open-source database system like Postgres
2. Replace its hash function with different hashing algorithms like XXHash3 and CityHash
3. Benchmark the performance with existing benchmarks such as TPC-C
4. Test the new hashing scheme on the Quartz persistent memory simulator
5. Compare the performance difference between new and old hashing schemes on the simulator
6. Run persistent memory benchmarks on the Quartz simulator
7. Calculate the error margin and adjust the readings accordingly.

Conclusion

- Hashing function for non-cryptography use cases should be fast and avoid collision
- Different hash functions can be used depending on the workload
- Similar graph pattern seen when using traditional memory and Quartz
- In average erase insert time, where one erase happens after one insert and keeping the map size constant, Quartz scales faster significantly
- Hypothesis: increasing hash table size should result in reasonable performance difference
- Whether persistent memory is faster than traditional memory in a specific scenario depends on workload and use case
- It is essential to evaluate benefits and limitations of both types of memory to determine best fit for intended use case
- More thorough testing is needed.

Questions ?

Thank you !

nbhatia3@ucsc.edu