Comparing Intel TBB `concurrent_hash_map` and `libcuckoo`

Manu Goyal, Dave Andersen, Michael Kaminsky

November 5, 2015

Overview

In this benchmark, we compare `libcuckoo`, our high-performance, memory-efficient hash table, with the Intel Thread Building Blocks `concurrent_hash_map` version 4.4. We compare the performance of the two tables across different types of workloads and different numbers of cores, and also compare the memory usage for different table sizes. We ran the benchmarks on a machine with 36 Intel Xeon E5-2666 v3 2.9 GHz (Haswell) CPUs and 60GB memory. Specifically, the machine had 2 sockets, each as its own NUMA node, each with a 9-core chip and 2-thread hyperthreading per core.

The benchmarks we ran measure pure read, pure insert, and mixed read-insert performance, as well as peak memory usage. For the performance benchmarks, we benchmarked maps with 64-bit integer keys, randomly shuffled numbers in the range 0 to the capacity of the table, as well as maps with 100-character string keys, designed to maximize comparison time.

Pure Read

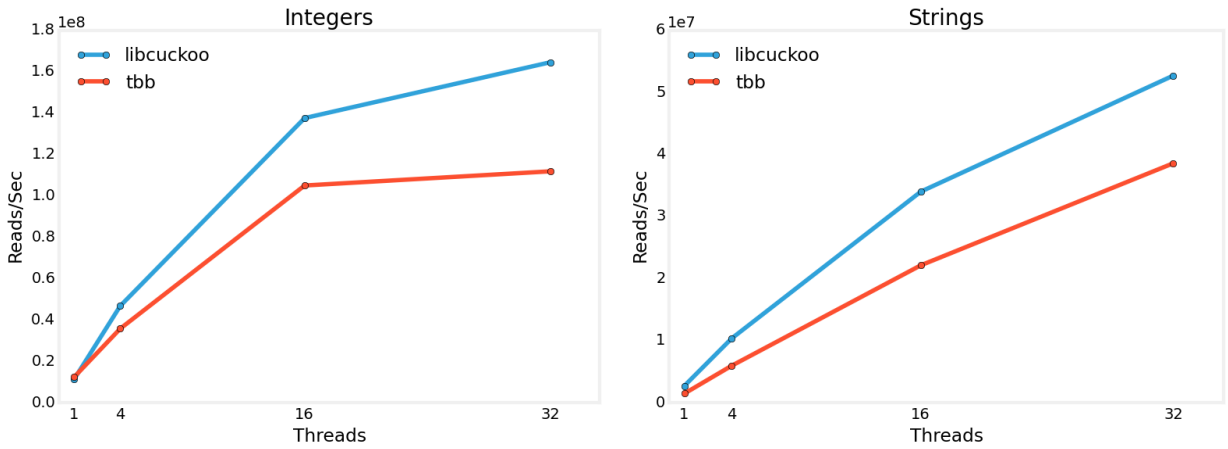
Our read benchmark fills a table up to 90% of its allocated capacity, then concurrently runs reads for data that is in the table and data that isn't. It counts the number of reads executed over 10 seconds. Figure 1 compares the read throughput of the two tables with integer and string keys. For both types of keys, `libcuckoo` outperforms `concurrent_hash_map`, with the difference getting slightly larger as we increase the number of threads. With 32 threads, `libcuckoo` outperforms `concurrent_hash_map` by 47% for integers, and 37% for string keys.

Pure Insert

Our insert benchmark measures the time taken to fill up a table from 0% to 90% of its allocated capacity. Figure 2 compares the insert throughput with integer and string keys. For integers, `libcuckoo` greatly outperforms `concurrent_hash_map`, by over 680%, and for strings, it outperforms `concurrent_hash_map` by 50%. It is interesting to note that `concurrent_hash_map`'s performance actually drops when we go from 4 threads to 16 threads, and then increases only slightly when we go up to 32 threads. With strings, since the cost of copying strings likely dominates the runtime, this effect is less apparent.

We suspect that part of the reason for this poor scaling is due to the fact that `concurrent_hash_map` doesn't deal well with multiple NUMA clusters, because data tends to be modified by CPUs on different NUMA clusters. `libcuckoo` specifically avoids this issue by separating all shared data by core, so minimal data ends up being transferred across NUMA nodes. Figure 3 confirms this hypothesis. `concurrent_hash_map` increases its throughput by over 240% on 16 threads, compared to its performance in Figure 2, while `libcuckoo` is not much affected by restricting to one NUMA node, increasing throughput by a more modest 26%. In both cases, `libcuckoo` significantly outperforms `concurrent_hash_map` on integer inserts.

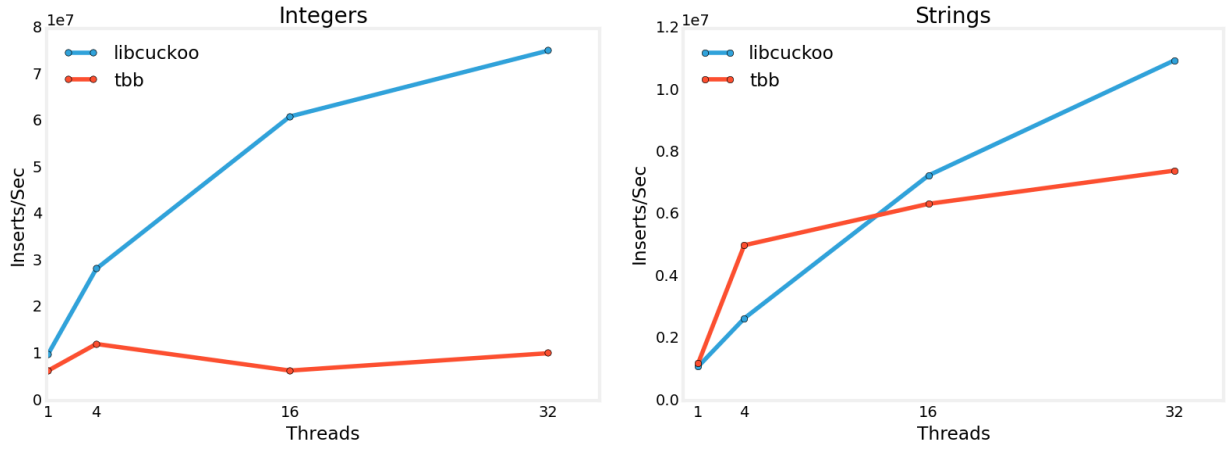
Another possibility for the performance difference is the data cache or TLB miss rate. We roughly examined these numbers for both tables, and found that the cache miss rate was about the same for both



(a) (b)

Figure 1: Pure read throughput for integer and string keys

tables, with `libcuckoo` tending to be slightly lower. The TLB miss rate, however, was far higher for `concurrent_hash_map`, likely due to the larger memory footprint. We mitigated this problem by utilizing linux hugepages in `concurrent_hash_map`, and, while this did decrease the TLB miss rate of `concurrent_hash_map` significantly, it had no effect on performance. So ultimately, cache and TLB miss rates do not seem to explain the performance difference for inserts.



(a) (b)

Figure 2: Pure insert throughput for integer and string keys

Mixed Workload

Our mixed benchmark runs a mixed workload of inserts and reads at a configurable ratio, and measures the time and number of operations taken to fill up the table from 0% to 90% of its allocated capacity. Figure 4 compares the performance of the two tables at different ratios of inserts (all with 32 threads), with

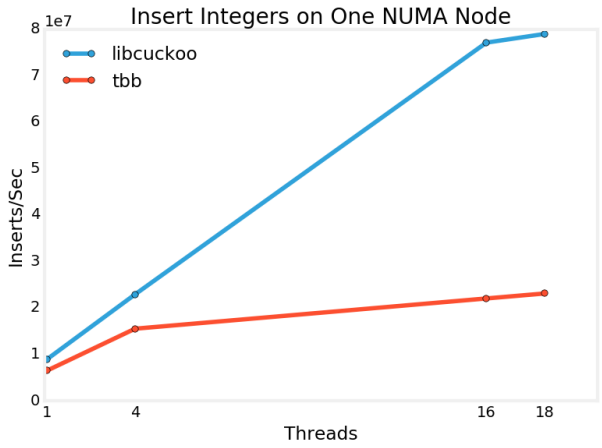
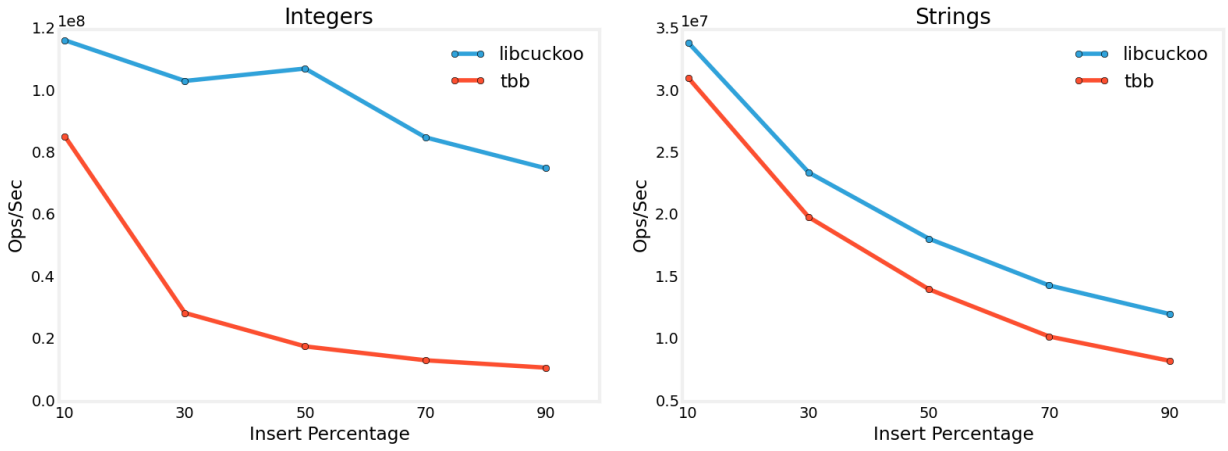


Figure 3: Pure insert for integer and string keys on one NUMA node

`libcuckoo` doing better with both integer and string keys. We see again that the difference between the two tables is much greater at higher insert percentages compared to lower percentages (588% compared to 36%, respectively), because the difference between `libcuckoo` and `concurrent_hash_map` is more pronounced for inserts than it is for reads.



(a) (b)

Figure 4: Mixed read-insert throughput for integer and string keys

Memory Usage

Finally, we compare the approximate memory usage of the two tables. While not a completely accurate measure of the amount of memory used by each table, we measured the maximum resident set size as determined by Ubuntu's `time` command for the insert benchmark, which is the peak memory usage for the process. For integers, `libcuckoo` scales far better than `concurrent_hash_map`, using 72% less memory than `concurrent_hash_map` with the largest table.

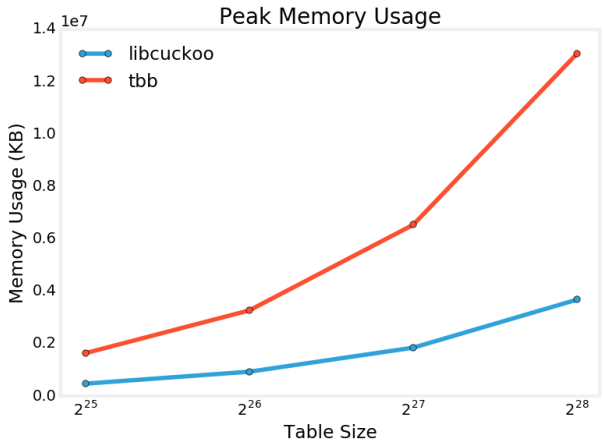


Figure 5: Peak memory usage for insert benchmark with integer keys. Table size is the number of elements each table has capacity for

Conclusion

`libcuckoo` has a number of features that cause it to perform better and use less memory than `concurrent_hash_map`. `libcuckoo` stores data in a cache-optimized form and avoids false sharing between CPU's, which allow it to scale inserts and reads very well to a large number of CPU's with low memory overhead. Furthermore, the cuckoo hashing algorithm lets it achieve very high table load factors before needing to expand, which significantly reduces memory usage.