CMSC858P Final Project

Patrick Gough   Eric Wang
Computer Science
University of Maryland
College Park, Maryland
{pgough, ewang119}@umd.edu

1 Introduction

We present a performance analysis of several LRU (Least Recently Used) cache implementations under high-concurrency workloads. Specifically, we evaluate our lock-free implementation of LRU using the cmuparlay/Flock library [11] [10], our locking C++ implementation using an Intel TBB [5] Hashmap and scoped locks, Guava Cache [4], and Caffeine Cache [1]. We benchmarked each implementation by measuring performance in terms of get() and put() throughput (million operations per second) across varying thread counts, cache sizes, and Zipfian skew parameters. We aim to understand the advantages and trade-offs lock-free caches, as well as how they compare to current state-of-the-art locking caches.

Our code is available at https://github.com/ewang119/Lock-Free-LRU-Cache-Benchmarks, including the lock-free and blocking implementations of the LRU cache and code for generating the benchmark graphs.

2 Background

An LRU cache is a fixed-size cache that evicts the least recently used element when an insert is performed at capacity. The typical implementation, which we used, is a combination of a Hashmap and a Doubly Linked List. The hashmap maps keys to nodes, and nodes are stored in the list from most recently used to least recently used order. The LRU cache supports O(1) get(key) and put(key) operations.

For this project, two of the caches we benchmarked were Guava and Caffeine Cache, both Java-based. Guava cache is a Java based cache developed by Google. It uses an Approximate LRU eviction policy to support concurrent cache inserts, as given many threads, the status of the most recently used element becomes obscured. Thus an approximate LRU policy becomes acceptable and even necessary.

We also benchmarked Caffeine cache, a high-performance concurrent cache developed by Ben Manes. It uses a Windowed-TinyLFU (Least Frequently Used) policy, which LRU-based Window Cache (very small portion of the cache) and a LFU-based Main Cache (the rest of it). New elements are inserted into the Window Cache, and once they are evicted, they get moved to the Main Cache.

3 Flock-Based Lock Free LRU Cache

We attempted two approaches to implement the Lock Free LRU Cache. For our first approach, we observed the code from the Flock doubly linked list. We started with a serial implementation from Geeks for Geeks [7], and we used the lock-free doubly linked list code alongside the std::unordered_map to implement the cache. The std::unordered_map mapped keys to nodes in the linked list. The linked list key field was used as the value, and the value field was unused. While it makes sense to only have the value field, we wanted to modify the linked list code as little as possible. A few changes had to be made to the linked list to make the code work - The insert function was modified to insert a node directly adjacent to the head as normally used in LRU cache implementations; and the remove function was modified to remove specific nodes as opposed to keys, as the std::unordered_map mapped keys to the linked list nodes directly, as opposed to a second key in the linked list.

Unfortunately, the std::unordered_map structure is not thread safe, and as a result of Flock's thread helping functionality, there were multiple concurrent reads/writes that resulted in segmentation faults. We tried several fixes such as surrounding the hashmap with a single try_lock and pre-allocating the entire capacity to eliminate the costly allocation operations. These fixes were ineffective and we decided to abandon this implementation.

Our second approach was to implement an LRU cache using the Flock lock-free hashmap instead of std::unordered_map. This hashmap similarly mapped keys to linked list nodes, and the algorithm similarly used the nodes' key fields to store the value. We similarly modified the insert and remove functions to return and accept nodes directly as opposed to keys. We also modified the hashmap to try to calculate the size (the Flock code does not track the size) so we could enforce the LRU capacity.

To approximate the size of the LRU cache at any time, we add two atomic values, *total_ins* and *total_res*. When inserting nodes into the cache, we use the compare and modify primitive to increment *total_ins* by one, and likewise with the *total_res*. The size is determined by taking the difference of the two values, and the last node in the linked list is ejected when the size exceeds the capacity.

To reduce the risk of a segmentation fault, one slight tweak was made. Similar to the Flock hashmap implementation, we added a "version" counter to the head of the linked list. When a node is inserted, the version is incremented by one and then checked against the true version. This checker ensures in-order inserts when called multiple times.

4 Blocking Intel-Based LRU Cache

We designed a locking LRU cache using the Intel TBB Hashmap, standard C++ doubly linked list, and the C++ std::scoped_lock. The Intel hashmap is highly optimized and takes advantage of fine-grained locking. Our doubly linked list also uses fine-grained locking, as we lock only the nodes that we are modifying as opposed to the whole list. We use a dummy head and tail node to maintain structure.

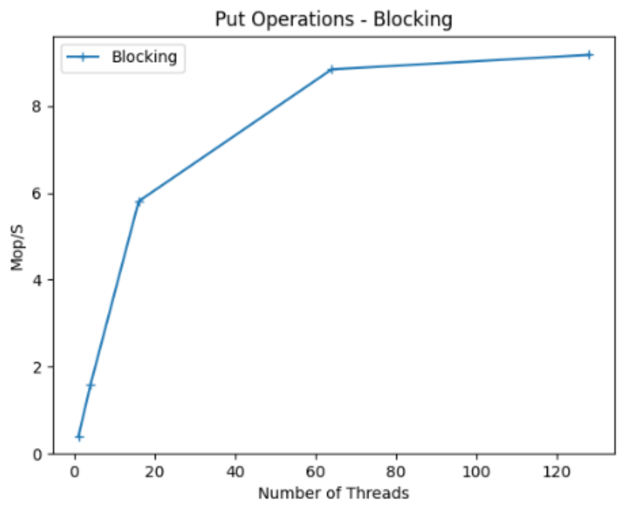
In order to avoid the issue of deadlocks, we decided to enforce a lock order when acquiring locks on our doubly linked list. The accessor in the Intel TBB Hashmap allows us to bypass any deadlock issues.

5 Benchmarking Setup

We benchmarked our implementations alongside Guava Cache and Caffeine Cache on a machine with 32 Intel(R) Xeon(R) Gold 6142 CPU cores (each with two threads for a total of 64 threads) and 376 gigabytes of RAM. We were primarily interested in varying the number of threads, so for each of the four implementations, we ran 1 million put operations and 1 million get operations on a cache size of 100,000 with a Zipfian parameter of 0.75 on one, four, 16, 64, and 128 threads (128 being our overclocking test case).

In order to benchmark Guava and Caffeine, we used the Java Microbenchmark Harness (JMH) [6]. To benchmark our locking TBB implementation, we used Google Benchmark [3]. Finally, when benchmarking our lock-free flock cache, we used the parlay parallel for loop and manually computed the runtime in nanoseconds to calculate the MOPS/S of each operation [8].

To obtain the keys to insert into our cache, we sampled from a Zipfian distribution in either YCSB [9] for Java or dirtyzipf [2] for C++ and inserted into an array of keys. The total number of keys we used was equal to the cache size tested that iteration. We called put() traversing through the array in order, but we called get() accessing every third element in the array and looping back around using modulus. Since all of our cache sizes are powers of 10, they are coprime with 3, meaning that traversing through the array in this manner still allows us to hit every element exactly once with no repeats, but not sequentially.



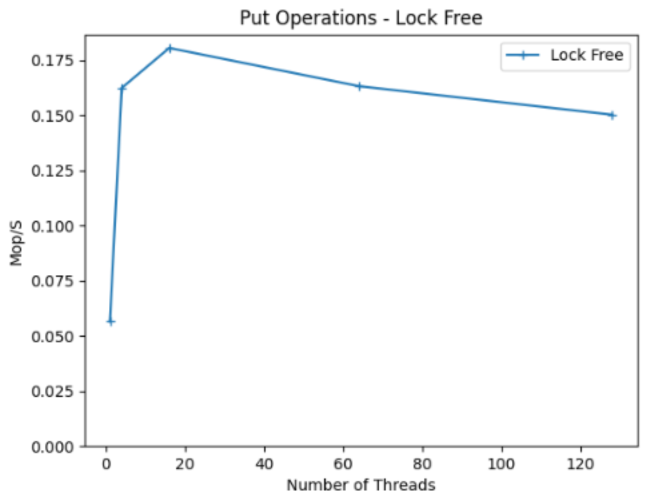


Figure 1: Blocking and Lock-Free LRU Cache MOPS/S over thread count for put

When benchmarking the put operation, we observe the blocking implementation is several orders of magnitude faster than the lock-free implementation as shown in Figure 1. The blocking version also has consistent speedup as more threads are introduced, even with overclocking as shown in the 128-thread case. On the contrary, the performance of the lock-free implementation declines after peaking at 16 threads. As shown in Figure 2, the case is similar for get operations.

We believe that heavy contention is one explanation for the lock-free implementation's relatively slow performance. Both the lock-free hashmap and the double linked list insert function use a single lock, no matter which key is being queried

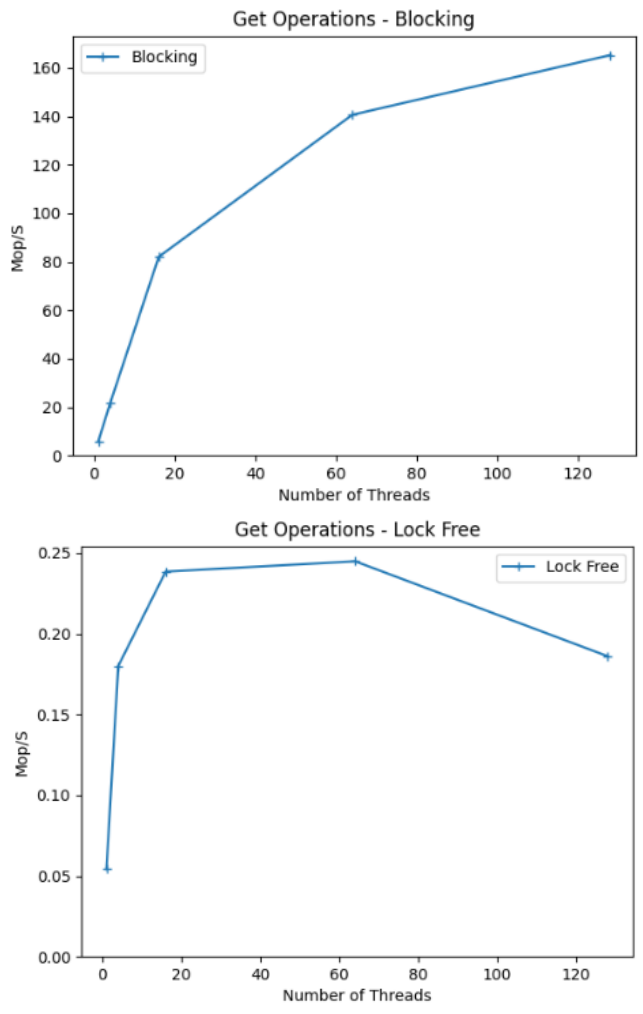


Figure 2: Blocking and Lock-Free LRU Cache MOPS/S over thread count for get

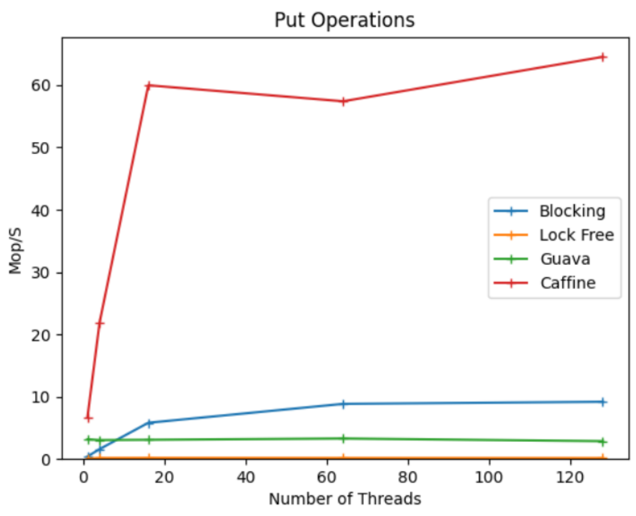


Figure 3: Blocking, Lock-Free, Guava, and Caffeine LRU Cache MOPS/S over thread count for put

or inserted. The Intel hashmap is believed to be more optimal, resulting in reduced bottleneck. Our changes to the Flock-provided structure code were relatively minor so we also believe that Flock is not yet at the point necessary to implement more complicated data structures by merging simpler primitives. Future work should focus on furthering development of Flock and making it easier to integrate into complicated applications.

As observed in Figures 3 and 4, the lock-free implementation performs poorly compared to Caffeine Cache and Guava Cache. With respect to the put operation, Caffeine Cache has the highest initial speedup, topping 60 MOPS/S with 16 threads, and growing in the overclock case. Interestingly, the Guava Cache was more optimal than our blocking implementation with a low thread count, but our blocking implementation outperformed it above four threads. This result is likely a product of Intel's cache being very optimized, whereas Guava is likely not optimized for large thread counts. With respect to the get operation, our blocking version consistently outperformed Guava likely due to the aforementioned optimization of Intel's cache.

References

[1] Caffeine cache. https://github.com/ben-manes/caffeine.

[2] dirtyzipf. (https://github.com/ekg/dirtyzipf.

[3] Google benchmark. https://github.com/google/benchmark.

[4] Guava cache. https://github.com/google/guava/.

[5] Intel tbb. https://github.com/uxlfoundation/oneTBB.

[6] Java microbenchmark harness. https://github.com/openjdk/jmh.

[7] Lru cache implementation using doubly linked list. https://www.geeksforgeeks.org/lru-cache-implementation-using-double-linked-lists.

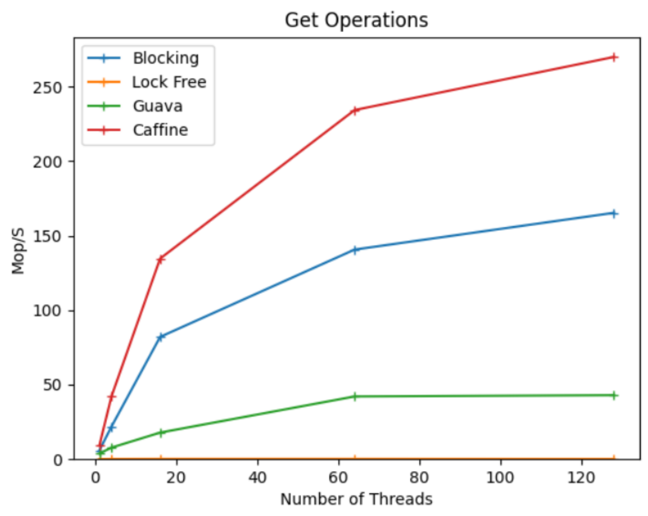


Figure 4: Blocking, Lock-Free, Guava, and Caffeine LRU Cache MOPS/S over thread count for get

[8] Parlay library. https://github.com/cmuparlay/parlaylib.

[9] Ycsb repository. https://github.com/brianfrankcooper/YCSB/tree/master.

[10] BEN-DAVID, N., BLELLOCH, G. E., AND WEI, Y. Lock-free locks revisited. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, NY, USA, 2022), PPoPP '22, Association for Computing Machinery, p. 278–293.

[11] BLELLOCH, G. E., ANDERSON, D., AND DHULIPALA, L. Parlaylib - a toolkit for parallel algorithms on shared-memory multicore machines. In Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (New York, NY, USA, 2020), SPAA '20, Association for Computing Machinery, p. 507–509.