Analyzing the Scalability of Lock-Free Hash Tables in Rust

A Preprint

John M. Truskowski
Department of Computer Sciences
University of Wisconsin-Madison
`jtruskow@gmail.com`

Sapan Gupta
Department of Computer Sciences
University of Wisconsin-Madison
`sgupta223@wisc.edu`

December 18, 2019

Abstract

We explore how the performance of lock-free hash tables varies with the tuning of various hyper parameters, namely, key/payload size and dynamic growth of the table. A major component of the project was a lock-free hash table implementation in Rust.

1 Introduction

Hash tables are a quintessential data structure for many applications due to their O(1) insertion and lookup times (in the best case). However, due to the increasing scale of data, the need for concurrent reads and writes to these hash tables grows ever more important. A critical data structure for this is the latch-free hash table, as traditional methods for controlling concurrent access (locks) introduce significant overhead and degrade the performance.

Latch-free hash tables guarantee that when one or more active threads are performing an operation on data structure, at least one thread will be able to complete within a finite number of steps, agnostic of other threads. These hash tables must perform well in the presence of contention, deletion and growing input sizes. We explore how a Rust implementation of latch-free hash tables performs when varying input size and key size for a variety of common database workloads.

2 Related Work

Prior to the publication of the Maier paper [1] which motivated this project, there was extensive prior work on the subject of concurrent lock-free sets and hash tables. In the absence of any feasible array-based lock-free hash table algorithms, Michael et. al proposed a list-based algorithm. Though like most previous work, it focused on correctness rather than scalability.

Similarly to Maier, et. al, Kim and Kim analyzed the performance of a number of hashing methods (linear, chained, and hopscotch hashing) implemented using lock-free data structures and Compare-And-Swap (CAS) operations [3]. Their novel comparison of these techniques showed that linear hashing is best for insert operations, while chained hashing is best for lookup (though using more space). However, they use uniformly-distributed keys which means there is little contention – thus their results may not scale well to real database systems.

Perhaps the most helpful piece of literature that we have encountered is Jeff Preshing's Junction library [4] and accompanying blog posts. In his blog, Preshing motivates and provides a minimal implementation of a latch-free hashtable in C++. His post provides an easily-digestible explanation for how CAS instructions may be used to improve performance compared to the traditional locking mechanisms. Though this simple implementation comes with a number of assumptions (32-bit keys and values, non-zero keys and values, fixed maximum entries, and no delete operation) these may be relaxed in a more complete implementation. Preshing's Junction library provides a full implementation of a few concurrent hash tables – namely using Linear, Leapfrog, and Grampa probing.

The folklore solution for latch-free hash table uses the concept of a tombstone value to mark the table entries that get deleted. Though this solution maintains elements' referential integrity by avoiding rearrangement of elements in the table to preserve data structure invariants, it leads to the eventual growth of search costs, as the elements are never freed. There are many papers that specifically try to tackle the problem of deletion in an open addressing hashing data structure.

In 2018, Sanders, et. al proposed a solution for preserving referential integrity of the elements involving deletion in a linear probing, while keeping a bound on search cost overheads.

Apart from this, there has not been much subsequent research on this topic since the Maier paper was published in 2016. WarpDrive (2018) introduces a novel probing scheme for massively parallel hash maps, but it is directed at multi-GPU architectures and thus we feel it is not particularly relevant to the multi-core CPU environment.

3 Approach

Our approach to the problem consisted of two distinct phases:

1. Implementation of the folklore solution presented by Maier et. al.
2. Relaxation of various constraints (static size of hash table and fixed key and value sizes)

3.1 Folklore Implementation

The basic folklore implementation has a number of constraints which come with the benefit of simplicity. It uses a fixed key/payload size and is statically-sized, meaning it cannot grow if the load factor gets too high.

3.2 Constraint Relaxation

The more generalized version of folklore uses tombstones for deletion [1]. These values may be removed from the hash table on a resize. Here, we introduced another check that allows new element hashes to this location while checking for duplication of keys during insertion [2]. This reduces the number of unusable entries while limiting the cost for searches in the table.

We introduce an unsafe and locked version of resize for the table to relax the static size constraint of folklore implementation. To allow for a lock-free resize hidden from the underlying application, we need more time to get familiarized with the concepts of wait-free MPMC queues and Epoch-based memory reclamation [5].

We relaxed the constraint of fixed sized keys by introducing 32-bit and 64-bit keys in hash table. Absence of function overloading in Rust meant that we needed to implement specific versions of each function interacting with a specific key size. This included using size specific hash functions.

[1] A tombstone is a reserved value (the maximum integer in our case) that represents a deleted value. Once a tombstone is used to mark an entry in hash table, that entry cannot be used any longer.

Since payloads do not require hashing and different sizes do not require special handling, we were able to introduce more variety of sizes like 8-bit, 16-bit, 32-bit and 64-bit payloads to the hash table.

4 Testing Methodology

In our testing, we decided to use a similar methodology as Maier in order to facilitate comparison between the implementations. Since the performance of many parallel algorithms depend on the scalability of parallel inserts and finds [1], we focused our testing around these benchmarks.

For the statically-sized table, each test consisted of 25 million insertions of random keys and values into a table initialized with size 2^{26}.

We used SpookyHash [8] as our hash function from Rust's fasthash library. It is similar to Google's CityHash, and has 32bit, 64bit, and 128bit versions (though we didn't test 128-bit key sizes as rust doesn't have atomic 128-bit primitives). We found that this hash function provided a much better distribution than the MurmurHash3 function recommended by Jeff Preshing in his blog post [6].

4.1 Hardware

We ran our experiments on a Google Cloud instance with 48 vCPUs (Intel Skylake) and 200GB of memory.

5 Results

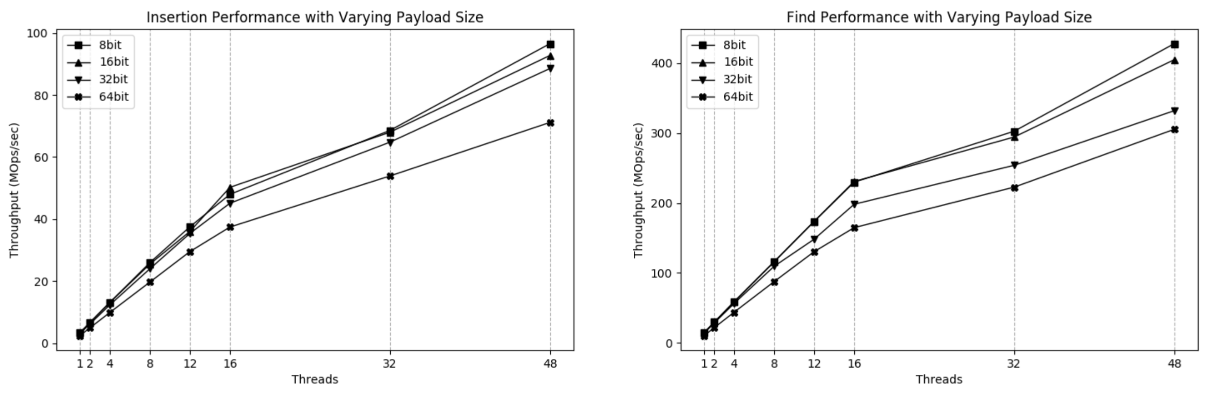

Figure 1: Insertion and Find throughput for varying payload size (Averaged over 5 runs)

5.1 Payload Size

The throughput curve for insertions and finds for varying payload sizes generally looks as we expect. In both cases, increasing the size of the payload reduces throughput for both insert and find benchmarks. Find performance is significantly higher than insert performance (roughly 4x throughput), likely due to the increased overhead of CAS operations required for insertions.

Strangely, the change in throughput actually improved for all payload sizes when going from 32 to 48 threads. We would expect improvements in throughput to diminish as more threads are added due to increased overhead for creating the threads as well as more threads hammering on the CAS operation. We are unsure of why this is happening.

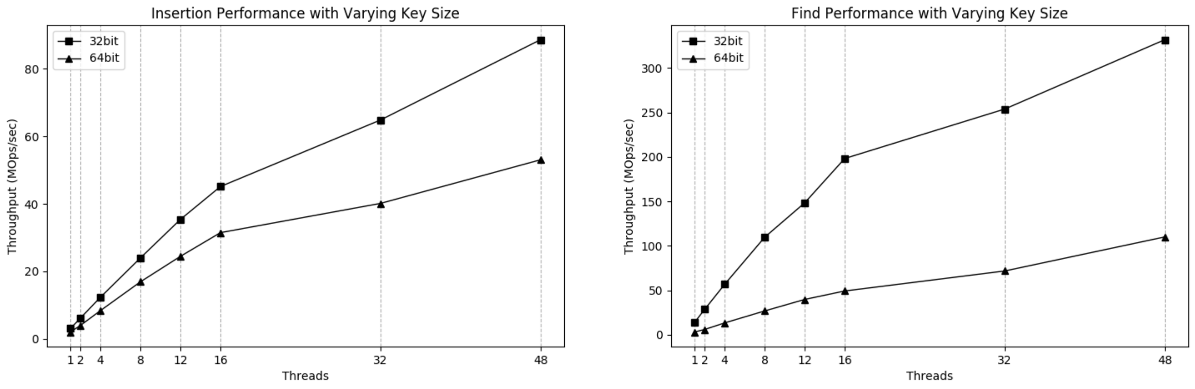

Figure 2: Insertion and Find throughput for varying key size (Averaged over 5 runs)

5.2 Key Size

We also explored how using key sizes of 32 and 64 bits affects performance of the hash table. The payload size was kept fixed at 32 bits. Clearly, key size has a much more profound effect on performance on than payload size. Increasing key size to 64 bits reduced throughput by 40% for insertions and 66% for finds.

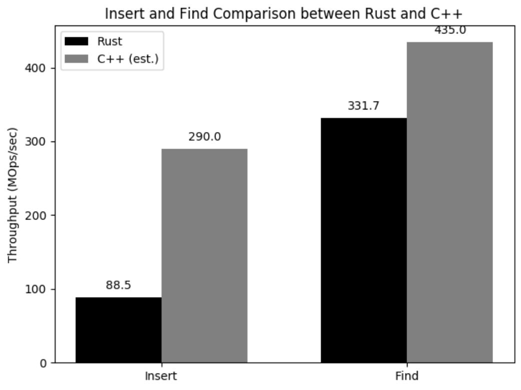

Figure 3: Performance comparison of Rust and C++ (32bit key/payload sizes, 48 threads)

5.3 Rust / C++ Comparison

For both insertions and finds, Maier et. al's C++ implementation outperformed our Rust code. There is some debate in the Rust community as to whether C++ or Rust is faster [7], however there are a variety of external factors that may have produced this result as well. While we tried to follow Maier's methodology as close as possible, we had to make some concessions due to our accelerated timeline. These are listed below:

1. *Pre-Computation of Keys*: In their testing, they randomly generated all keys to be inserted prior to the experiment. In our testing, this led to decreased throughput (likely due to many threads hammering on the shared data structure that stored the pre-computed keys) so we opted to have each thread generate a key/value pair upon insertion.
2. *Hardware*: We ran our experiments on different hardware.
3. *Hash Function*: Maier used two CRC32C x86 with different seeds to generate the upper and lower 32 bits of each hash value. We used a single 32-bit hash function that performed 'good-enough' for our benchmarks.

4. *Input Buffering*: In Maier's experiments, each thread reserved chunks of 4096 operations to perform at once. In our implementation, each thread performs a single operation at a time. This may make our measurements more prone to variance [1] (Though averaging over 5 runs should help alleviate this)

5. *Number of Operations*: We performed 25 million inserts/finds per experiment [2] compared to 100 million for the C++ implementation.

An interesting result that is immediately obvious from Figure 3 is that our ratio of insert-to-find performance (0.267) is significantly lower than the C++ implementation (0.667). We attribute this to the pre-computation of key/value pairs that Maier did in their experimentation. In our case, each thread having to randomly generate a key and value for each insert likely resulted in a notable performance hit.

6 Discussion

6.1 Challenges

We faced a number of challenges over the course of this project. Probably the largest was learning Rust. Of course, having to learn a language with new syntax is a significant undertaking. However, we found a number of Rust concepts (such as traits, mutability and borrow checker) unintuitive and lacking an analogy to other languages. Additionally, we were getting extremely low throughput compared to the C++ implementations we were using for comparison. Ultimately (thanks to Suryadev Rajesh and Muthunagappan Muthuraman), we discovered that using the `-release` flag when building the Rust project is necessary to achieve good performance.

A second challenge was related to the initial hash function we chose. In Jeff Preshing's blog, he recommended Google's MurmurHash3 function, as he found that "it's fast and scrambles integer inputs quite nicely." [6] However, this produced strange behavior for us. We found that past 16 threads performance began to degrade significantly. Upon switching to a new hash function, we saw performance improve to the levels we were expecting.

In Rust, resizing hash table even in a single threaded environment has been a challenge. Since resize involves replacing the internal fixed-sized key-value vector by a bigger vector. In addition, all the key-value pairs in the older vector need to be rehashed to the newer vector. For this level of mutability from inside a thread-safe insertion method, we used atomic reference count and read-write lock on the internal vector. This is not a lock-free implementation/thread-safe implementation of resize. There has been speculation about Epoch-based memory reclamation as a means to provide a thread-safe dynamically resizeable hashtable, though we couldn't use this mechanism due to the time-constraint and complexity of the concept.

6.2 Further Work

Given more time, there are a number of areas we would like to explore further. One is testing additional key sizes. We found that 32 and 64 bit hash functions are by far the most common. For anything smaller than 32 bits, we would likely have to design our own hash function. We opted not to explore a custom hash function due to time constraints, and it would also make fair comparison with SpookyHash more challenging.

Dynamic migration of hashtable in a multithreaded lock-free environment is another area that requires more exploration. At this time, multi-producer multi-consumer wait-free queues seem like holy grail for us to use for performant resize functionality.

[2]While we would have liked to perform the full 100 million operations, we were renting a Google Cloud instance, thus we wanted to reduce the CPU time we used. In the experimentation we did, a major difference between the two sizes was not detected, thus we chose to use 25 million operations for our final testing.

Performance of our hash table with a more mixed (real-world) workload involving insertion, search, deletion and inherent table migration is another interesting analysis that would provide more insight into the relative performance comparisons between Rust, native-Rust and C++ implementations.

7 Conclusion

We found that the performance of the hash table is far more sensitive to key size compared to payload size. Throughput scales well as value sizes increase, so a lock-free hash table may be a good choice for applications with large payloads. If an application requires key sizes larger than 64 bits, a concurrent hashtable in Rust may not be a good choice, as Rust only offers atomic primitives up to 64 bits. The hashtable will have much higher throughput with smaller key sizes.

Additionally, our experiments show that for our benchmarks, C++ is faster than Rust. However, additional work is required to bring our experiments even further in line with the methodology used by the Maier paper to confirm these findings.

Overall, we conclude that under the correct constraints, concurrent hash tables in Rust are a good way to enable massively parallel operations on a large-scale hashtable. We feel that the advantages provided by the Rust language (especially its memory and thread-safety guarantees [9]) make up for the slightly reduced throughput we saw in comparison to the C++ implementation. In addition, once we are more familiar with this programming language and its internal workings, there is a lot of room for better optimized code in Rust to compete in performance with the C++ version.

References

[1] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent Hash Tables: Fast and General?(!) *arXiv preprint arXiv:1601.04017*, 2016

[2] Peter Sanders. Hashing with Linear Probing and Referential Integrity *arXiv preprint arXiv:1808.04602*, 2018

[3] Euihyeok Kim and Min-Soo Kim. Performance analysis of cache-conscious hashing techniques for multi-core CPUs. *International Journal of Control & Automation (IJCA)*, 2013

[4] Jeff Preshing. Junction GitHub Repository (2016-2018). https://github.com/preshing/junction

[5] Open source documentation host for crates of the Rust Programming Language. Crate crossbeam. https://docs.rs/crossbeam

[6] Jeff Preshing. The World's Simplest Lock-Free Hash Table (2013) https://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/

[7] Rust vs. C++: Fine-grained Performance. Hacker News Forum, Feb. 2016. https://news.ycombinator.com/item?id=11047144

[8] Bob Jenkins. A 128-bit non-cryptographic hash function. http://www.burtleburtle.net/bob/hash/spooky.html

[9] Rust Programming Language Docs. https://www.rust-lang.org/