15-618 Final Project Report

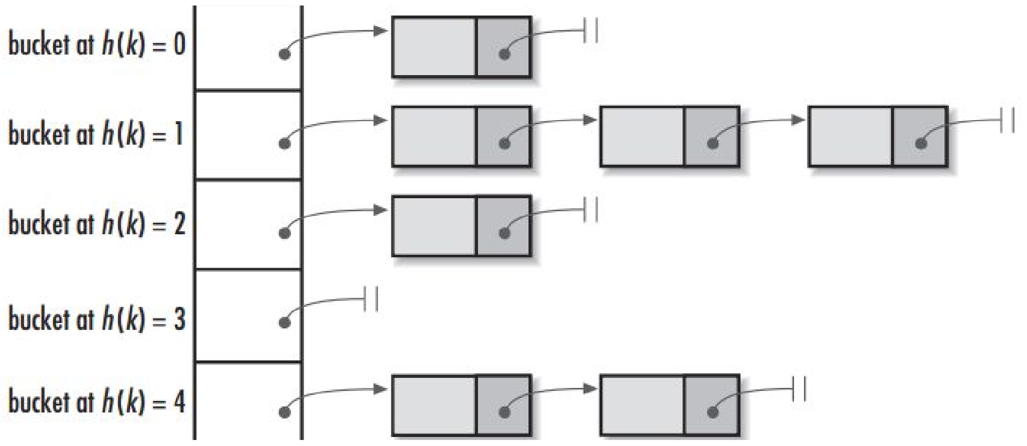
Concurrent HashMaps

Ilan Biala and Varun Sharma

Background

Key-value stores

Data structures are a fundamental component in any algorithm and application, especially in concurrent programming where it is even more important to efficiently organize data for optimal space usage and execution time. Since multi-core systems have become more common, programmers have started sharing data structures between threads, but without proper synchronization their implementation had race conditions and unwanted behavior. With synchronization through the use of locks, programmers observed dramatically slower programs that did not scale as well as they had hoped. As a result, programmers saw a need for data structures that are fundamentally capable of having multiple threads access the data structure simultaneously and perform better than ordinary data structures that use a global lock to synchronize data structure access across threads. Concurrent data structures often use fine-grained locking or even lock-free implementations to reduce contention for a global lock. Key-value stores, typically called hashmaps, are good candidates for fine-grained locking since some underlying implementations inherently allow for more granular locks to be easily used, such as the separate chaining with linked lists implementation. As shown below, the implementation uses an array of pointers which represent each bucket, and are indexed based



on the hash value of the key being inserted. At a minimum, users need to be able to retrieve data from the hashmap, insert (and update) data into the hashmap, check for the existence of data in the hashmap, and remove data from the hashmap. Many implementations also provide ways to iterate over the keys and values in the hashmap, and more feature-rich functionality such as conditional replacement and insertion.

Performance considerations of hashmaps

In general, SIMD operations and data-parallel execution do not map well to data structures, because these optimization techniques require multiple pieces of data, but often data structures only internally operate on a single piece of data to be fully generic. Some data structures such as multimaps could potentially benefit from SIMD operations or data-parallel execution, but their purpose and implementation is out of scope for this project. Locality of reference often yields large speedups; however hashmaps that use separate chaining and linked lists spend most of their time accessing random locations in memory when iterating through a bucket’s linked list. With this implementation, temporal locality of reference is still present, although spatial locality is much more limited. Finally, the amount of parallelism is for the most part limited by the number of buckets in the hashmap, since more buckets allows for more threads to concurrently operate without possible conflicts, and thus less synchronization overhead.

Transactional Memory

In any concurrent data structure, some synchronization overhead will always exist since programmers must implement their program to always be correct at compilation time. As a result, much of the synchronization overhead is still overly conservative and could be avoided. Transactional memory attempts to mitigate this overhead by enabling arbitrary blocks of code to easily be efficient and lock-free. Transactional memory uses the preexisting cache coherence mechanism present in modern multi-core processors to detect data conflicts at runtime with little overhead [1]. From this concept, Intel has implemented two variants of transactional memory starting with its Haswell processors: restricted transactional memory (RTM) and hardware lock elision (HLE). RTM provides the programmer with more flexibility as to how willing the application should be to continue attempting speculative execution, but requires the programmer to also provide fallback paths and error handling. HLE is a much simpler variant that attempts to mitigate most of the synchronization overhead by speculatively attempting to execute a critical section without acquiring (or releasing) the lock, and only aborting the speculative execution if a data conflict is detected. If a data conflict is detected, the speculative execution is aborted, and execution restarts but will acquire the lock. As a result, in scenarios where acquiring the lock is extremely expensive but conflicts are not frequent, HLE sacrifices some of the expected speedup from speculatively executing more than once to make the implementation easier for the programmer. In most cases, HLE will recover most of the lost speedup due to synchronization overhead, so HLE is an ideal first step for programmers to take when attempting to further optimize concurrent data structures [2].

Design

We targeted multi core Intel CPU's for our map implementation because we wanted to use transactional memory and atomic primitives, which are CPU features. Intel's transactional memory extensions are also only available on Haswell and later CPUs. We implemented our maps in C++ because the transactional memory intrinsics are only easily available in assembly, C, and C++, and C++ has existing Concurrent and Atomic map implementations to compare against.

Synchronized Hashmap

We began by implementing a baseline synchronized hashmap, which uses a global mutex to lock all operations. It operates like a standard hashmap with chaining, where there are a certain number of buckets. For each operation, a bucket is selected by taking a hash of the key and modding that by the total number of buckets. Once a bucket has been identified, the value at that location is a pointer to the start of a linked list, and key-value pairs are inserted and searched for within this list. To add a key-value pair, one indexes into the array of buckets, searches the bucket's linked list to see if the key already exists, and either updates it or inserts a new node at the end of the list containing the key-value pair. To get a key-value pair, one indexes into the array of buckets, searches the bucket's linked list, and returns the value if the matching key is found, otherwise an error is thrown. To remove a key-value pair, one indexes into the array of buckets, searches the bucket's linked list, and deletes the corresponding node if its key matches, otherwise an error is thrown. The main problem with this implementation is that everything is synchronized and none of the inherent parallelism provided by having separate chains is taken advantage of. We then added parallelism by implementing a concurrent hashmap with more granular locks.

Concurrent Hashmap

The concurrent hashmap added parallelism by allowing operations to occur within different chains in parallel and also by allowing non-modifying operations to occur within the same chain in parallel. This was done by using a pthreads read-write lock on each bucket. A read lock would be requested for all non-modifying operations and a write lock would be requested for all modifying operations. This improved performance significantly; however, every single operation requires a lock to be acquired, which is expensive because it has to invalidate every core's cache entry for the lock and write the updated lock out to main memory. This becomes more expensive as cores are added due to cache coherence traffic.

Transactional Memory - Hardware Lock Elision

Intel's TSX-NI extensions enable optimistic execution of the critical section of a lock by continually watching cache coherence traffic for the memory that is accessed in the critical section. We utilized these instructions to implement a lock with HLE. To implement the lock, we used the GCC atomic intrinsics to make a simple mutex, which atomically sets a number to 1 when locked and 0 when unlocked. Because of HLE, this lock does not actually lock anything if there are no conflicts detected. This makes read-heavy use cases much faster because nothing is ever locked. When a conflict is detected while executing the critical section, the lock is actually acquired, which prevents other cores from entering their critical section. The threads will then operate sequentially until they all finish, when the lock will reopen. This lock was placed on each bucket, so that it operated in the same manner as the concurrent hash map.

Evaluation Methodology

Benchmark technology, target platform, and implementations

To evaluate our implementation and compare its performance to other implementations, we used Google Benchmark [4], which is a micro-benchmark framework that provides helpful functionality to easily and accurately benchmark small scenarios. Google Benchmark itself determines how many iterations of the given block of code to run to record statistically stable results [5]. We configured Google Benchmark to also measure wall-clock time since we are benchmarking multi-threaded code, we turned off frequency scaling to avoid noisy results and extra overhead, and we fixed the clock frequency on the processor [6]. Our test system was an Intel i7-6700 clocked at 4.0 GHz with 32 GB of DDR4 2400 MHz RAM. We benchmarked five different implementations: our synchronized hashmap, our concurrent hashmap with per-bucket reader-writer locks, our concurrent hashmap with per-bucket locks (writer-only for simplicity), Folly's ConcurrentHashMap [8], and Folly's AtomicHashMap [9]. Folly is an open-source library by Facebook with a heavy focus on performance and scalability. Our implementations as well as Folly's ConcurrentHashMap allow an infinite number of elements to be stored, but Folly's AtomicHashMap requires the programmer to specify the number of elements that will ever be inserted (element deletions do not reduce the count). As a result, Folly's AtomicHashMap is expected to be more performant in many cases, but it sacrifices flexibility as a result. As an aside on correctness - we implemented correctness-focused unit tests for the interface we designed that every implementation inherits from so we could verify the correctness of our implementations. In addition, we implemented wrappers for Folly's ConcurrentHashMap and

AtomicHashMap and tested these as well to make sure our results were consistent with their implementations.

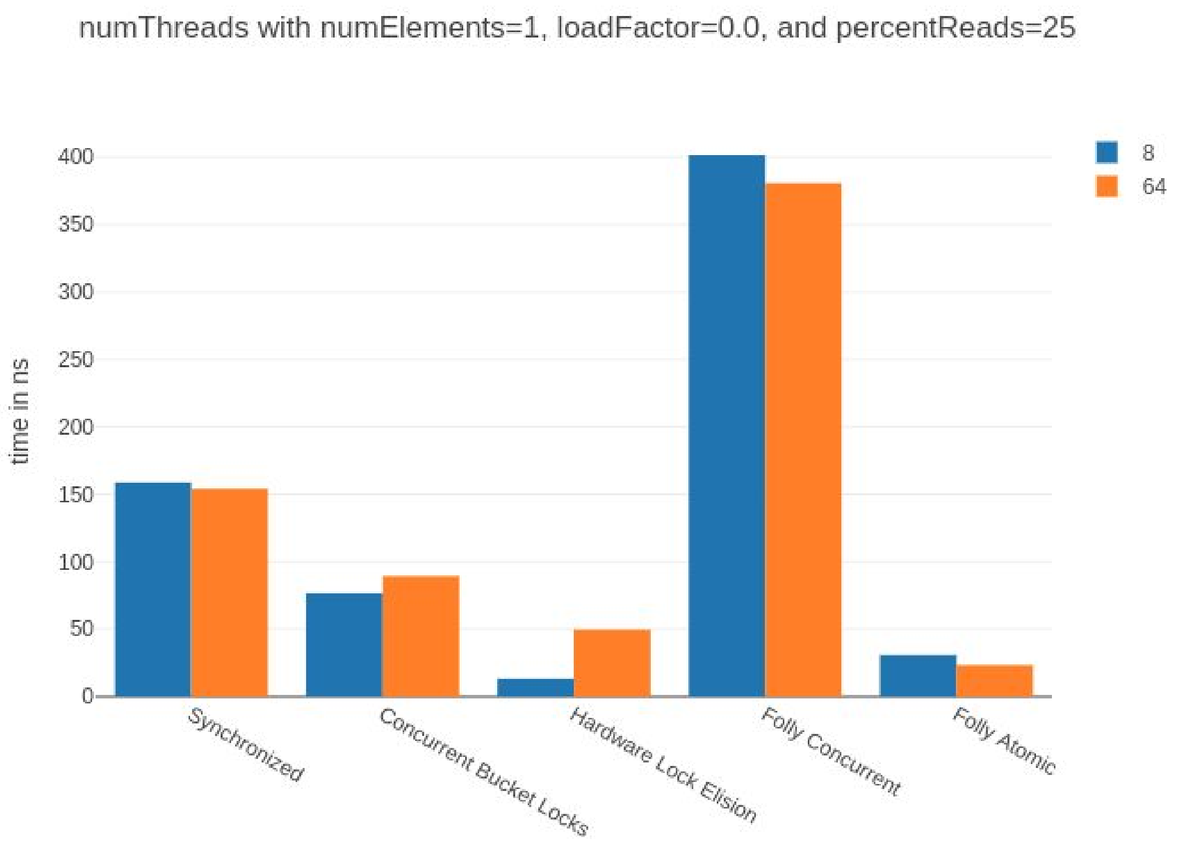
Workloads and Experimental setup

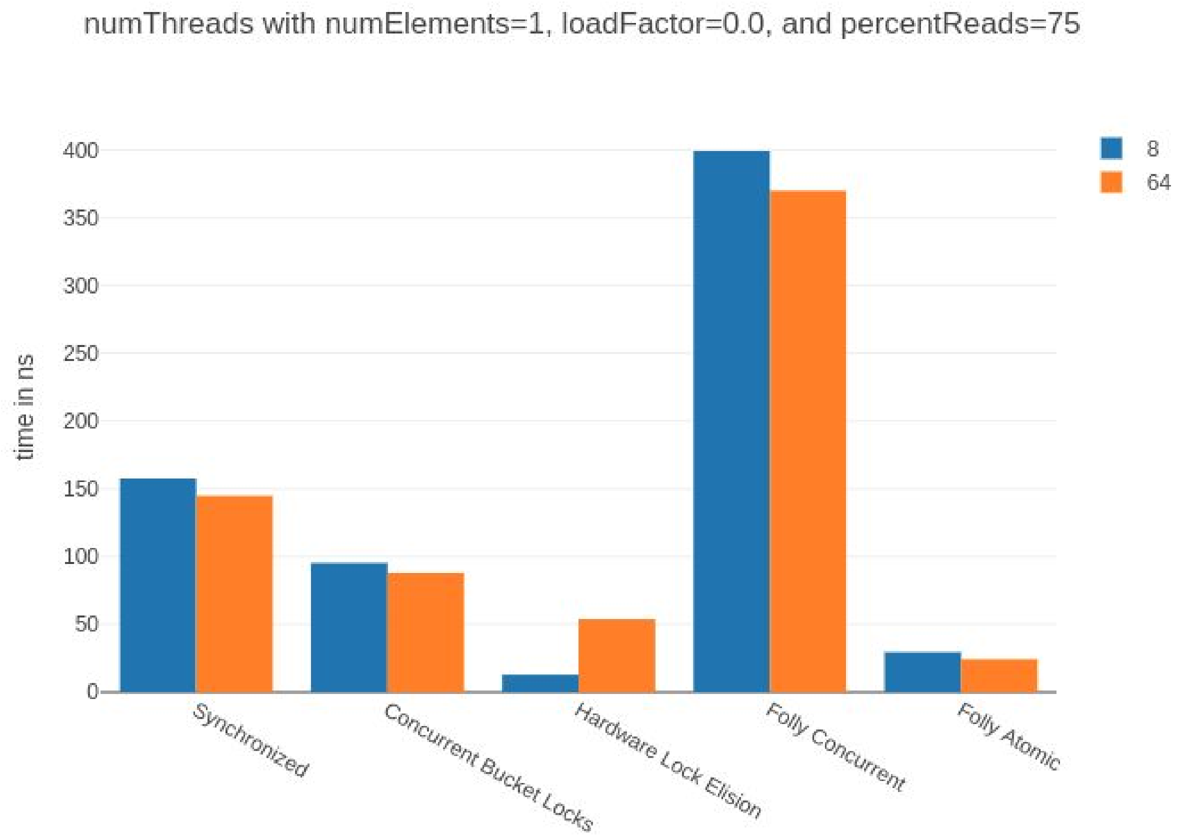
In both of our benchmarks, we pre-populate the hashmap with the desired number of elements. We only start timing when all threads have a randomized access pattern and are ready to start operating on the hashmap. Since Folly's AtomicHashMap can only store a fixed number of elements and writing a benchmark that would not violate this constraint was quite difficult, our benchmarks only retrieve and modify values, they do not remove or insert new key-value pairs.

We created two separate benchmarks to test different scenarios. The first workload is a high-contention scenario where all threads are operating on 1 element in the hashmap. For this workload, we test with both 8 and 64 threads and 0, 25, 50, 75, and 100% reads. The second workload is a generalized benchmark that focuses more on typical usage. For this workload, we vary the following in all combinations: the number of elements between 16, 256, 4096, and 65536; the load factor between 0.2, 0.4, 0.6, 0.8, and 1.0; the percentage of reads (versus modifications) between 0, 25, 50, 75, and 100%; and the thread count between 1,2, 4, 8, 16, 32, and 64. We've chosen these parameters since they target various amounts of contention, are reasonable estimates of typical usage (both in the number of elements being operated on and the amount of reads and modifications being performed), and show the quality of multi-core scaling for each implementation.

Results

High-contention results

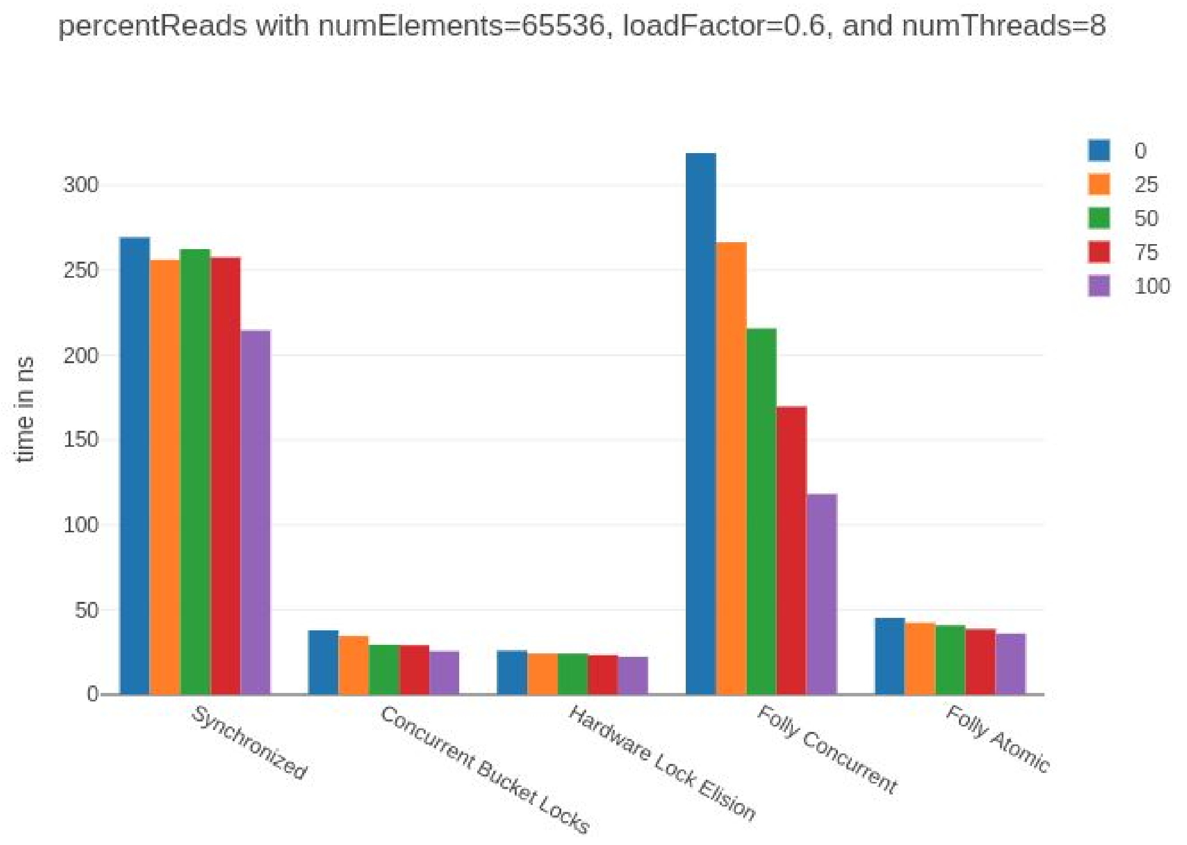




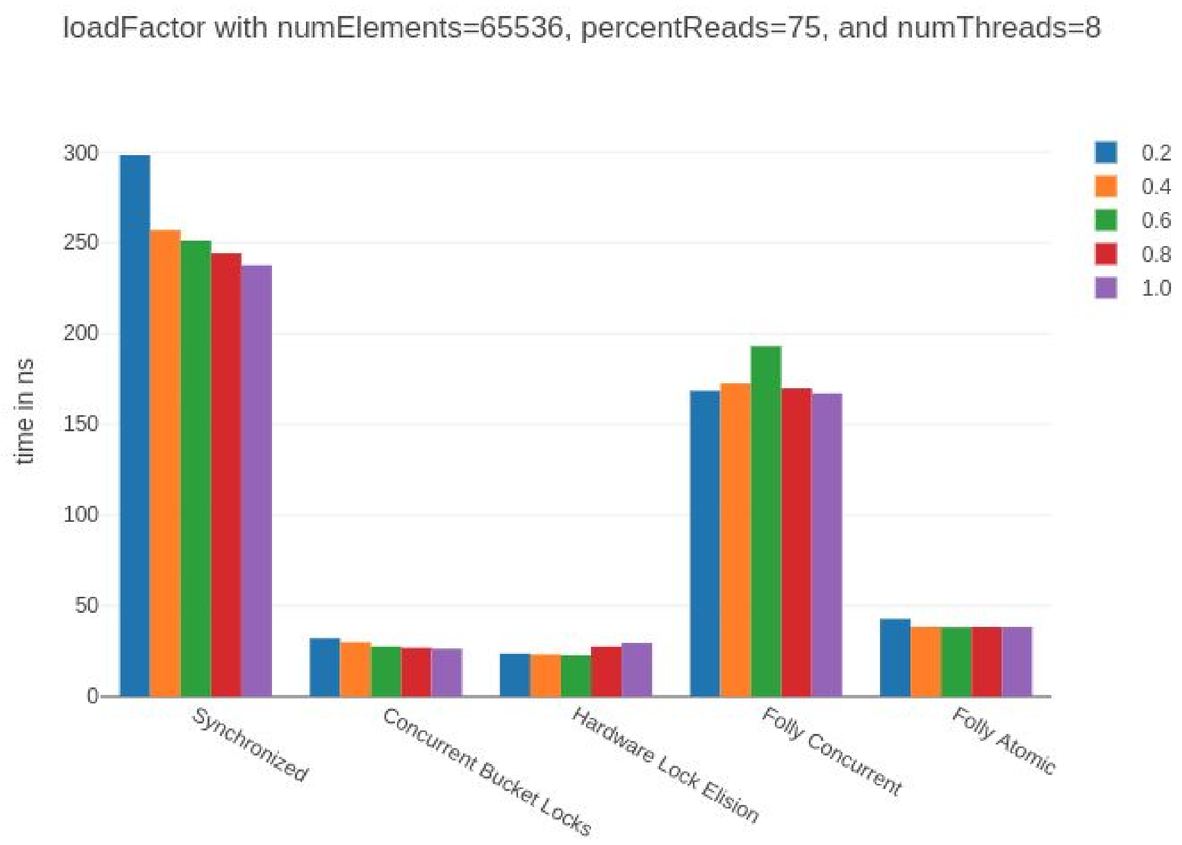
The above graphs are the result of our high-contention workload at 25% and 75% read workloads, where all threads operate on one element. In this workload, HLE performs extremely well and yields nearly a 10x speedup compared to our standard concurrent hashmap implementation and about 4x compared to Folly's AtomicHashMap implementation when 8 threads are used, because there is a 1:1 mapping between threads running in software and execution contexts supported by the CPU. HLE's performance degrades by a factor of about 10 when 64 threads are used, since context switches, interrupts, and some instructions and system calls cause HLE abortions [3, 10]. Folly's ConcurrentHashMap implementation is also slower in this case (and in most other cases), because Folly's implementation has much more functionality, which likely leads to larger overhead to maintain correctness for other operations.

General-usage workload

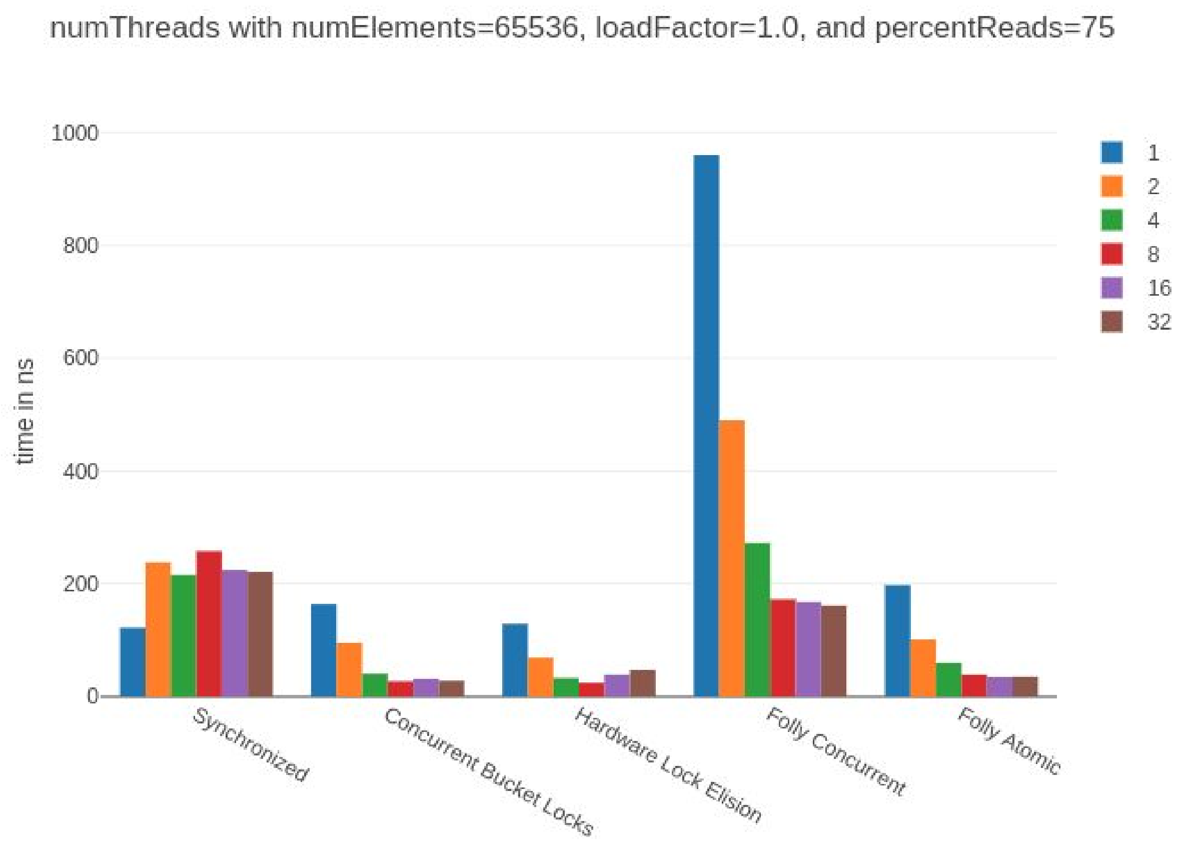
After running the general use benchmark with all of the different combinations of inputs specified in our experimental setup, we analyzed the results and found these to be the most interesting and provide the clearest picture as to the performance of the different implementations we analyze.



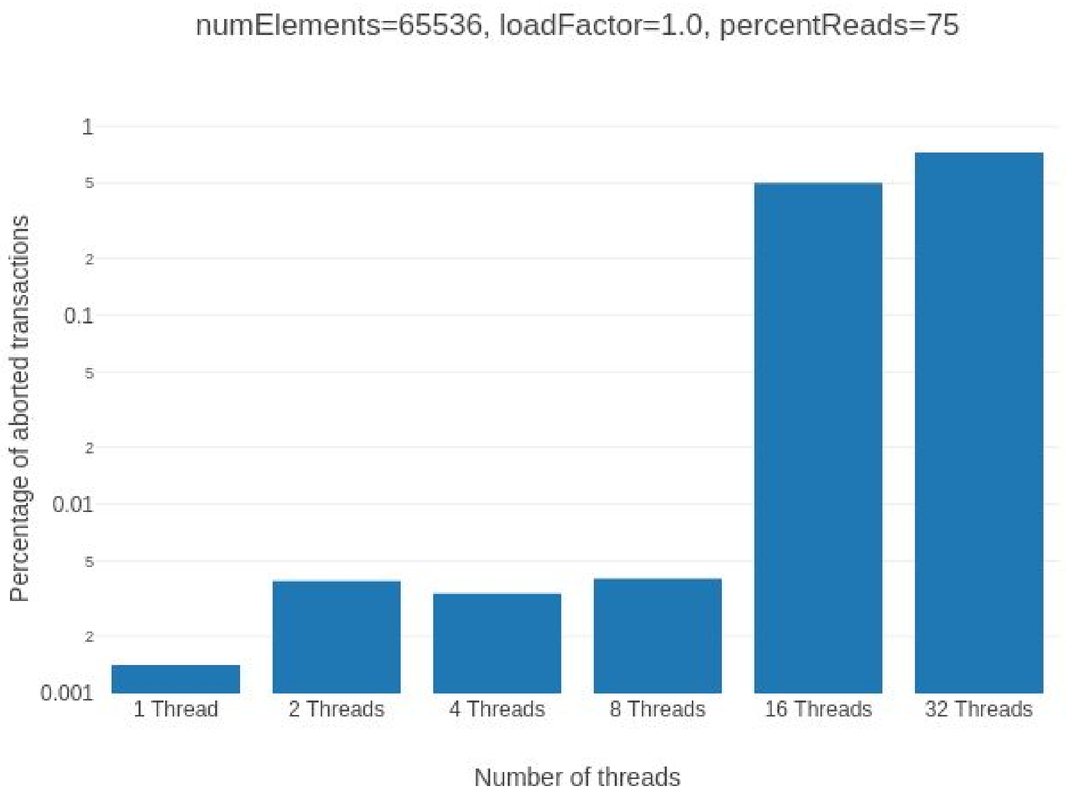
Above is the result of our benchmark with the percentage of reads varied for each implementation. In general, the trend seen for each implementation is unsurprising since reads are faster than updates in hashmaps, especially for Folly's ConcurrentHashMap, which uses read-write locks to allow for more concurrent accesses among threads in many cases. Our concurrent hashmap with HLE performs the best since we have a 1:1 thread mapping between software and hardware execution contexts, so again we avoid unnecessary transactional memory abortions. Additionally, our concurrent hashmap still outperforms Folly's AtomicHashMap because internally the AtomicHashMap uses sequential memory, and so random accesses result in false sharing of data since multiple key-value pairs fit in a 64-byte cache line. Lastly, our performance benchmark results line up with Folly's own results [11], which is a good sign and shows that our results are consistent with theirs. We do not have a concrete answer as to why Folly's ConcurrentHashMap performs poorly, but our theory is that there is more overhead in each operation to maintain correctness with the much larger feature set that the implementation provides. We see the performance improves significantly though, so Folly's ConcurrentHashMap does allow for concurrent operations, it just starts off with very poor performance.



This test shows that load factor does not have a large impact on performance for every implementation, but the synchronized and concurrent implementations get slightly faster with a higher load factor. This effect is probably seen because we vary the load factor by keeping the number of elements constant (to keep total work performed constant) and decreasing the number of buckets internally, so the buckets are packed more tightly in memory. This means that more of the data stored by the hashmap is closer together, and so spatial locality is slightly improved. Hardware Lock Elision gets slightly slower with a higher load factor. This could be caused by more aborts due to more elements being closer in memory and modifications to a cache line causing another transaction to abort since Intel's transactional memory operates at cache line granularity.



This test shows how the time to execute a single read/modify operation varies with the number of threads running. The computer this was tested on has 4 cores and 8 threads, so the synchronized implementation plateaued around ~4-8 threads because everything is done sequentially and context switches happen on the order of milliseconds, while this takes ~200 ns. This means that the only other threads contending for the mutex are the ones that are currently scheduled. The synchronized implementation is faster for a single thread because the mutex is held in the cache with no contention, while the concurrent and HLE implementations have to keep track of 65,536 different locks. All of the other implementations get faster with more threads likely because of cache locality with all of the threads accessing the same data, so it is more likely to be cached locally. The HLE implementation gets slower after the number of threads exceeds the number of hardware threads at 8. This is because context switches can cause HLE aborts.



In the above plot, we show the number of transactional memory aborts dramatically increases as the number of software threads becomes greater than the number of hardware execution contexts. This is because of context switching, system calls, interrupts, and a small set of instructions that cause transactional memory abortions, and all of these causes are much more prevalent as the number of threads increases past the number of logical cores.

Analysis and Conclusion

After reviewing our results, we believe we were quite successful at achieving our goal of implementing a highly concurrent, generic hashmap. We see significant performance improvements between the traditional synchronized hashmap implementation and our concurrent hashmap implementations, and our exploration of Intel’s transactional memory shows that HLE is a good first optimization for hashmaps. We analyzed our implementations with four different parameters independently varied, and we noticed that our implementation’s performance did not vary significantly as the number of elements increased. As we showed with our high-contention workload, different workloads do exhibit different execution behavior. One important conclusion we drew was that HLE favors writers over readers, and this could hurt performance for read-heavy applications compared to our standard concurrent hashmap. For example, if there is a reader and a writer and both are speculatively executing and the writer finishes first, the writer will cause the reader to abort and restart its transaction. Since our general usage workload accurately represents many independent threads operating on the hashmap, we did not see a lack of parallelism or dependency relations limit speedup. Instead, we found that the number of hardware execution contexts limited the speedup of our concurrent hashmap with HLE, since having a greater than 1:1 mapping of software threads to hardware execution contexts would cause unnecessary abortions, which would result in wasted cycles and more interconnect traffic. Additionally, since this is a concurrent data structure, performance is always going to be limited by communication and synchronization overhead. We did not have feasible access to a machine with more cores, since none of the clusters supported Intel’s transactional memory extensions, so our conclusions are only partially valid up to 8 logical cores. However, we speculate that transactional memory's performance scales with the performance of the interconnect on the CPU, so most likely on enterprise CPUs with up to 64 logical cores we would see performance benefits. After this point, we would likely see performance level off as the interconnect's latency is no longer reasonably constant and begins to increase in modern, high-performance CPUs. Since very little computation is actually done in the internal implementation of a hashmap, it's not as beneficial to break down the percentage of time to spend in each region. We do know however that the random memory accesses when iterating over the linked list and loading invalid values from L3 cache or from main memory are where the majority of time is spent outside of any lock acquisitions and releases that take place. Since we are already quite competitive with - and even outperforming - industry-standard implementations, it would require a significant amount of profiling to identify room for improvement. We could improve the performance of our concurrent hashmap with HLE in some cases by implementing HLE on top of a read-write lock compared to a standard lock, which would limit the performance impact when a writer and many readers are speculatively operating and the writer causes an abortion, which causes every reader to sequentially grab the mutex and block other readers. Finally, since this is a generic data structure, writing an implementation that targets a CPU is still the correct choice, since hashmaps are mostly used in general-purpose programming.

References

Hashmap Image: http://www.itcuties.com/java/hashmap-hashtable/

[1]: https://dl.acm.org/citation.cfm?id=165164

[2]: https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell

[3]: https://software.intel.com/en-us/blogs/2013/05/03/intelr-transactional-synchronization-extensions-intelr-tsx-profiling-with-linux-0

[4]: https://github.com/google/benchmark

[5]: https://github.com/google/benchmark#runtime-and-reporting-considerations

[6]: https://github.com/google/benchmark#disable-cpu-frequency-scaling

[7]: http://www.cs.cmu.edu/~418/lectures/17_lockfree.pdf

[8]: https://github.com/facebook/folly/blob/master/folly/concurrency/ConcurrentHashMap.h

[9]: https://github.com/facebook/folly/blob/master/folly/docs/AtomicHashMap.md

[10]: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

[11]: https://github.com/facebook/folly/blob/master/folly/AtomicHashMap.h#L65

Work Division

Ilan was the main contributor of the following:

- Developing the initial ConcurrentMap interface and the concurrent hashmap implementation with locks for each chain
- Creating the Makefile and getting compilation and linking working
- Writing unit tests for the ConcurrentMap interface
- Getting Folly set up (compilable and linkable in our toolchain)
- Writing the benchmarks
- Brainstorming and developing the workloads for the benchmarks

Varun was the main contributor of the following:

- Developing the initial ConcurrentMap interface and the concurrent hashmap implementation with locks for each chain
- Getting Folly set up (compilable and linkable in our toolchain)
- Developing the synchronized and concurrent with HLE hashmaps as well as implementing wrappers for Folly's ConcurrentHashMap and AtomicHashMap
- Creating the data processing pipeline that ingests results from Google Benchmark and produces graphs
- Brainstorming and developing the workloads for the benchmarks

The total credit should be distributed 50%-50%.