On the Performance of Non-blocking Hash Tables

Jesper Puge Nielsen
Technical University of Denmark
jesper@puge.dk

Sven Karlsson
Technical University of Denmark
svea@dtu.dk

ABSTRACT

Advanced atomic operations, widely available in modern processors, enable developers to implement non-blocking data structures. These scale better with the number of threads than regular lock synchronized data structures.

In this paper, we evaluate our recently proposed lock-free hash table with open addressing and linear probing. In our data structure, we split insertions into two phases: One inserting the value and one maintaining exclusivity among keys. Each phase can be performed atomically and insertions are thereby made lock-free.

Our hash table has a low constant memory usage of one pointer per bucket, exhibits good cache locality, and uses less memory than other non-blocking hash tables at a fill level of 33% and above. Our experiments show that the hash table scales strongly and on average yields 24.3% higher throughput than state of the art lock-free open addressed hash tables, while using a third of the memory.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

1. INTRODUCTION

Hash tables are essential data structures in computer science and used in many different applications across a wide range of application areas. Hash tables were not originally designed for concurrent accesses but safe concurrent operations become a desired feature as more fields turns to concurrent programming. For hash tables to be correct, at most one value can be associated with the same key. In this paper, we present a thorough performance evaluation of our recently proposed concurrent and safe non-blocking hash table [5]. Our hash table inserts values in two phases: A value is first preliminary inserted, without allowing searches to find it. After that, any duplicated values for the same key are removed and the value is fully inserted, making it available for searches. Search and remove operations on our hash table are identical to sequential open addressing hash tables with the exception that we use atomic operations to replace removed values with special markers, called *tombstones*. The space that tombstones take up can be reclaimed and reused when new values are inserted. A prior paper have more details about implementation aspects [5].

Features of our hash table includes good cache locality, a constant low memory usage and strong scalability. This while having a smaller implemented code size than existing non-blocking open addressed hash tables.

2. RELATED WORK

Researchers have tried to make hash tables scale by using a combination of micro-locking and lock-free searches. Cuckoo hashing is an open addressed hash table developed by Pagh and Rodler [6] where each key only maps to two different buckets, which each can contain a number of values. In the case both buckets are full, an empty spot is moved to one of the buckets by swapping a number of values between their two buckets. The series of displacements is called the Cuckoo path. Li et al. [3] have developed Concurrent Cuckoo hashing which allow concurrent operation on the hash table, and use micro-locking during updates. Li et al. suggests using a BFS to find the shortest Cuckoo path and splits the search phase from the actual displacement of values, only locking the hash table during the latter stage. Contrary to Concurrent Cuckoo hashing, our hash table allows for both concurrent searches and updates.

Operations of the same type can progress concurrently in the phase-concurrent hash table presented by Shun and Blelloch [8]. Michael presented a lock-free adaptation of the traditional hash table with chained hashing [4]. Feldman et al. takes a different approach than traditional sequential hash tables in their wait-free hash table based on expanding arrays [2].

3. EVALUATION

We evaluate our proposed hash table in terms of scalability and memory consumption by comparison with the chained hash table proposed by Michael [4], the expanding array based hash table proposed by Feldman et al. [2], the phase-concurrent hash table by Shun and Blelloch [8], Concurrent Cuckoo hashing [3], as well as PH by Purcell and Harris [7]. We have made two simple cache aware modifications to improve the performance of PH. Purcell and Harris keep the buckets and probe bounds separately in their design of the hash table. Instead, we suggest storing the probe bound as part of the bucket to place them on the same cache line. Furthermore, we suggest using linear probing as opposed to quadratic probing, as suggested by Purcell and Harris. Linear probing places more buckets of the probe sequence on the same cache line, causing fewer cache misses, and the linear traversal of the memory is easier to predict for the memory prefetcher. We call this cache aware version, *PH+*.

We have implemented each of the hash tables in C++ based on the pseudo-code provided in their respective papers. We have used compare-and-swap, *CAS*, as fundamen-

tal atomic operation in all the different implementations. We also compare our hash table to a blocking hash table with open addressing and linear probing and the `unordered_map` from the C++ standard library. Both of these hash tables are protected by a readers-writer spinlock that each thread acquires prior to accessing the table and releases afterwards.

3.1 Experimental setup

We use GCC version 4.9.2 at optimization level 3, -O3, to compile all source code. All experiments are run under Debian 8 with Linux kernel version 3.16.0-4-amd64 on two Intel Xeon X5570 2.93GHz processors in a dual socket set-up with 48GB RAM. The set-up has a total of 16 logical processor cores when Hyper-Threading is enabled. During experiments we pin threads to a specific core. Each of the first four threads are mapped to their own physical cores on the same processor, the next four threads are pinned to the same physical cores as the first, and the remaining eight threads are mapped likewise on the other processor. Our experiments show a significant drop in performance when going from four to five threads and again going from 12 to 13 threads. This is because TurboBoost is disabled when Hyper-Threading is used. The two processors have a non-uniform memory access latency to their shared memory. Debian uses a policy where heap memory is allocated to the processor which first references the memory, and it will take the other processor slightly longer to access the memory. This leads to a small drop in performance when going from eight to nine threads.

3.2 Scalability

We examine the scalability of the hash tables by measuring the time they take to perform a constant number of operations using 1-16 threads.

The hash tables are filled to an initial load before each experiment. All threads are released simultaneously, at which point they will each perform their share of the total operations using a specified distribution of inserts, searches and deletions. We keep the number of insert and removal operations equal to ensure a near-constant load factor on the hash table. The number of buckets are the same in all hash tables, except for Feldmans hash table, the `unordered_map`, and Cuckoo hashing. Our implementation of Feldmans hash table uses arrays of four buckets as nodes in the tree, which we found to give the best performance in our experiments. Likewise, we have implement Cuckoo hashing with three values in each bucket as it gave the best performance.

Prior to each experiment, we preallocate the memory required for all hash tables, except the `unordered_map`, to filter out the cost of memory allocations from the measured time. We likewise generate all random numbers beforehand and store them in memory private to each thread. The execution time of each experiment is determined by the execution time of the slowest thread, from which the operations per microsecond is calculated. Each experiment has been performed ten times, and we present the average along with a 95% confidence interval. The interval bounds can be hard to see on some graphs though, as they in most cases are very tight. The experiments using lock based hash tables have only been performed five times due to their significantly longer execution time.

We have run our experiment with a hash table size of 33,554,432 and an initial load of 33%. The size of the hash

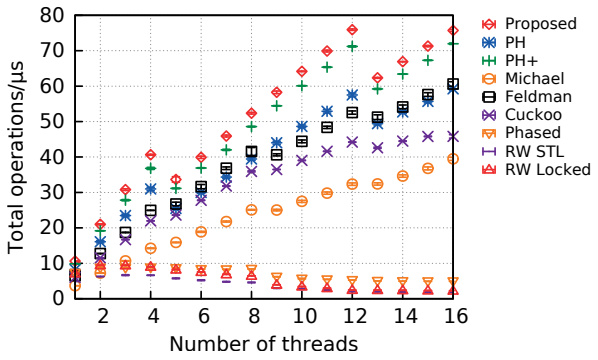

Figure 1: Total number of operations/μs, with an initial load of 33% and a 10:80:10 distribution of operations.

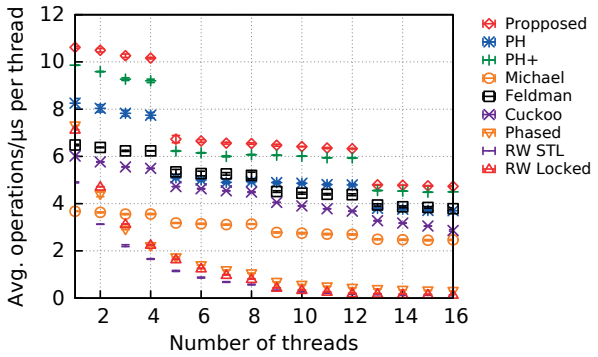

Figure 2: Average number of operations/μs for each thread, with an initial load of 33% and a 10:80:10 distribution of operations.

table was chosen such that the initial values do not fit the L3 cache and memory therefore must be accessed. The experiment performs 134,217,728 operations, consisting of 10% inserts, 80% searches and 10% removals, as read dominated workloads are typical for hash table and therefore commonly used to evaluate them. Figure 1 shows the combined number of operations performed by all threads per microsecond under this configuration. Given perfect scaling, this number would increase linearly with the number of threads, doubling when the thread count doubles. As expected, the lock-based hash tables fail to scale, and their performance even drops below that of a single thread when running on more than a few threads. The phase-concurrent hash table also fail to scale, as threads are blocked while waiting for the desired phase to begin. This indicates that phase-concurrent data structures are ill suited for mixed workloads. The phase-concurrent and lock-based hash tables will be excluded from the remaining experiments.

All the lock-free data structures scale close to linear, except when going from 4-5, 8-9 and 12-13 threads due the effects of simultaneous multithreading and non-uniform memory accesses. Cuckoo hashing scales well at thread counts below eight threads, but its performance starts to degrade slowly at higher thread counts. Figure 2 shows the average number of operations performed by each thread per microsecond. This number would be constant in a strong

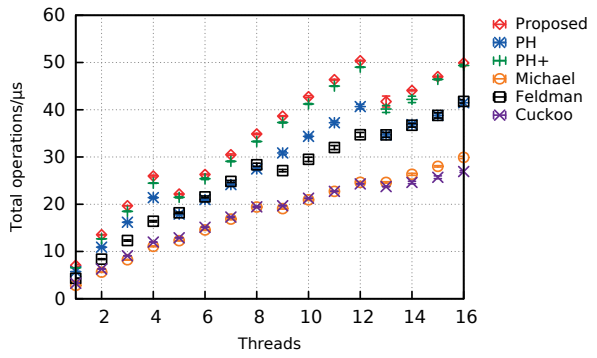

Figure 3: Total number of operations/μs, with an initial load of 70% and a 10:80:10 distribution of operations.

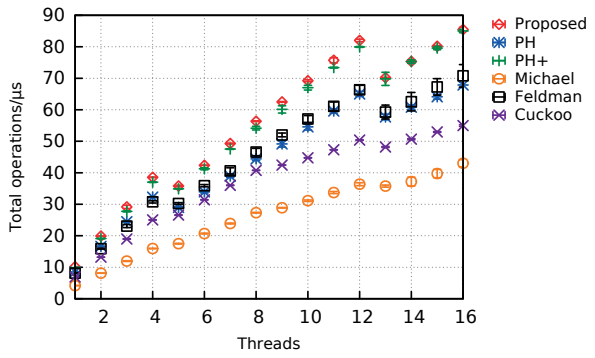

Figure 4: Total number of operations/μs, with an initial load of 33% and a 33:34:33 distribution of operations.

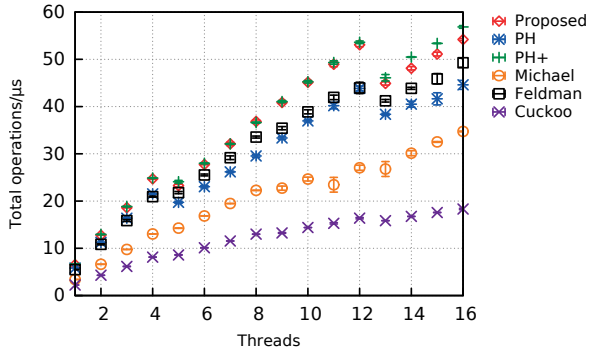

Figure 5: Total number of operations/μs, with an initial load of 70% and a 33:34:33 distribution of operation.

scaling system. When more than four threads are used, all non-blocking hash tables reach a near constant number of operations per thread and microsecond. Our hash table performs best in this experiment, performing 5%-10% more operations per microsecond than PH+.

We expect our hash table and PH+ to have better cache locality than the others due to their cache awareness and linear access pattern, which makes prefetching easier for the processors. PH also have a linear access pattern but generates more cache misses, as both the buckets and probe bounds will cause misses. The nodes in Michael's linked list and Feldman's tree can be allocated in an arbitrary order on the heap, which makes it hard for the prefetcher to preload cache lines.

We have used the hardware performance counters of the processors to verify our expectations. The measurements were done using the performance application programming interface [1] version 5.4.1. During our experiments we measured the number of L1 and L2 cache misses, the number of cycles stalled waiting for memory, the total instructions completed and the number of compare-and-swap instructions. All low-level metrics have been collected via Intel's hardware performance counters. For example, the number of stall cycles has been collected via the performance event aggregating stall cycles across all stall scenarios. The values were roughly equal regardless of the number of threads when summing the counters across processor cores. We have therefore chosen to present the values from running the experiments with 16 threads. Results are listed in Table 1. Our hash table generates fewer cache misses while running the experiment than the other hash tables. The fewer cache misses leads to fewer cycles where the processor is stalled waiting for memory, which combined with the fewer instructions required to complete the experiment results in better performance. PH+ is closest to our hash table, generating 8.2% more L1 cache misses, which causes its slightly longer execution time.

Figure 3 shows the results from running the experiment with an initial load of 70% and a distribution of operations of 10% inserts, 80% searches and 10% removals. The observed performance is generally worse than the experiments with a lower initial load. This is caused by the greater contention for buckets, leading to longer probe sequences. The hash tables still scale at the higher load.

We have also run the test suite with an equal amount of insert, searches and removals, and an initial load of 33% and 70%, with results shown in figures 4 and 5 respectively. The results show the same trends as the previous tests, but with a smaller performance gap between hash tables. PH+ outperform our hash table with up to 4.9% in the experiment with initial load of 70%. During insertions, our hash table traverses the entire probe sequence to look for duplicated keys, whereas PH+ use the probe bound to limit number of buckets traversed. High fill levels increase the length of probe sequences, and therefore PH+ slightly outperform our hash table in the experiment with high fill level and insertion heavy workload.

Across all our experiments, the version of PH with our cache awareness improvements outperforms the original version by 20.4% on average. This shows the importance of cache awareness in data structure design. Our hash table outperform the original PH by 24.3% on average, and the improved version by 3.2% on average.

3.3 Memory consumption

We examine the memory consumption of the different hash tables by comparing the amount of memory used by each hash table at different fill levels. We have implemented each hash table with the least amount of memory possible. Figure 6 shows the effective memory usage of the implemen-

Hash table	L1 cache misses	L2 cache misses	Stall cycles	Instructions completed	CAS instructions
Proposed	438464672	386321179	65706564380	17322980299	42321868
PH	730697603	629610368	89544027950	18781202362	56730007
PH+	474615592	411477878	71133223848	18830274972	56629454
Michael	1010355283	726012745	128151689064	47688505090	28412561
Feldman	532016521	463568433	84012488025	20793597746	29530321
Cuckoo	1041213528	950397712	103102073174	32063822563	75048307

Table 1: Average value of hardware performance counters for the experiment with 16 threads, an initial load of 33% and a 10:80:10 distribution of operations.

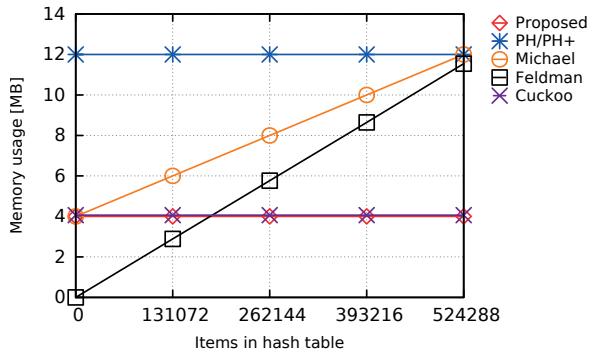

Figure 6: Memory used by the hash table implementations with different number of inserted values.

tations when each hash table has 524,288 buckets. Our hash table has a constant low memory usage, as each bucket is implemented by a single pointer. Michael's hash table also uses one pointer for each bucket but require memory for the linked lists. Each node in the linked lists consists of two pointers, one pointing to the value and one pointing to the next node. PH also uses a constant amount of memory but requires significantly more than ours, as both a value pointer, a versioned state and the maximal probe length is stored for each bucket. Cuckoo hashing uses the same amount of memory for buckets as our hash table but also requires 64 kilobytes of memory for the locks that protects various parts of the hash table.

Feldman's hash table initially uses very little memory, but the amount grows linearly as values are inserted and arrays are expanded. We have found that Feldman's and our hash table use an equal amount of memory to store the same number of values, when our hash table is filled to approximately 33% of its capacity. At higher capacities, the memory consumption of Feldman's hash table outgrows the constant consumption of our hash table.

4. CONCLUSIONS

We have evaluated our recently proposed non-blocking hash table with open addressing and linear probing. We compared our results to state-of-the art algorithms. Our experiments show that our hash table scales close to linear with the number of threads and on average outperforms Purcell and Harris' hash table by 24.4%. We have developed a cache optimized version of their hash table with 20.4% higher throughput than the original. We still outperform this version of the hash table by 3.2% while using a third of the memory.

5. REFERENCES

[1] PAPI. http://icl.cs.utk.edu/papi/. Retrieved: 2015-09-12.

[2] S. Feldman, P. LaBorde, and D. Dechev. Concurrent multi-level arrays: Wait-free extensible hash maps. In *Proceedings - International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 155–163, 2013.

[3] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, 2014.

[4] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

[5] J. P. N. Nielsen and S. Karlsson. A scalable lock-free hash table with open addressing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 33:1–33:2, 2016.

[6] R. Pagh and F. F. Rodler. Cuckoo hashing. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2161:121–133, 2001.

[7] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. *Lecture Notes in Computer Science*, 3724:108–121, 2001.

[8] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 96–107, 2014.