Algorithmic Improvements for Fast Concurrent Cuckoo Hashing

Xiaozhou Li[1], David G. Andersen[2], Michael Kaminsky[3], Michael J. Freedman[1]

[1]*Princeton University,* [2]*Carnegie Mellon University,* [3]*Intel Labs*

Abstract

Fast concurrent hash tables are an increasingly important building block as we scale systems to greater numbers of cores and threads. This paper presents the design, implementation, and evaluation of a high-throughput and memory-efficient concurrent hash table that supports multiple readers and writers. The design arises from careful attention to systems-level optimizations such as minimizing critical section length and reducing interprocessor coherence traffic through algorithm re-engineering. As part of the architectural basis for this engineering, we include a discussion of our experience and results adopting Intel's recent hardware transactional memory (HTM) support to this critical building block. We find that naively allowing concurrent access using a coarse-grained lock on existing data structures reduces overall performance with more threads. While HTM mitigates this slowdown somewhat, it does not eliminate it. Algorithmic optimizations that benefit both HTM and designs for fine-grained locking are needed to achieve high performance.

Our performance results demonstrate that our new hash table design—based around optimistic cuckoo hashing—outperforms other optimized concurrent hash tables by up to 2.5x for write-heavy workloads, even while using substantially less memory for small key-value items. On a 16-core machine, our hash table executes almost 40 million insert and more than 70 million lookup operations per second.

1. Introduction

High-performance, concurrent hash tables are one of the fundamental building blocks for modern systems, used both in concurrent user-level applications and in system applications such as kernel caches. As we continue our hardware-driven race towards more and more cores, the importance of having high-performance, concurrency-friendly building blocks increases. Obtaining these properties increasingly requires a combination of algorithmic engineering and careful attention to systems issues such as internal parallelism, cache alignment, and cache coherency.

EuroSys'14, April 13–16, 2014, Amsterdam, Netherlands
ACM 978-1-4503-2704-6/14/04.
http://dx.doi.org/10.1145/2592798.2592820

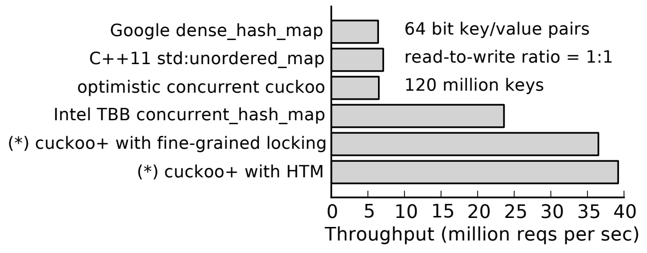


Figure 1: Highest throughput achieved by different hash tables on a 4-core machine. (*) are our new hash tables.

At the outset of this research, we hoped to capitalize on the recently introduced hardware transactional memory (HTM) support in Intel's new Haswell chipset, the TSX instructions [1]. Contrary to our expectations, however, we ended up implementing a design that performs well regardless of its use of HTM, and the bulk of our time was not spent dealing with concurrency mechanisms, but rather in algorithm and data structure engineering to optimize for concurrent access. For fast hash tables, HTM's biggest benefit may be to software engineering, by reducing the intellectual complexity of locking, with a modest performance gain as a secondary benefit.

As a result of these efforts, this paper presents the design and implementation of the first high-performance, multiple-reader/writer hash table that achieves the memory efficiency of multi-way Cuckoo hashing [18]. Most fine-grained concurrent hash tables today store entries in a linked-list with per-bucket locks [2] or Read-Copy-Update (RCU) mechanisms [17, 19]. While often fast, the pointers used in these approaches add high overhead when the key/value items are small. In contrast, our Cuckoo-based design achieves high occupancy with no pointers.

We contribute a design that provides high throughput for multiple writers; prior work we build upon [8] allowed only a single writer, limiting the generality of the data structure. Our design uses algorithmic engineering of Cuckoo hashing, combined with architectural tuning in the form of effective prefetching, use of striped fine-grained spinlocks, and an optimistic design that minimizes the size of the locked critical section during updates.

The result of these engineering efforts is a solid building block for small key-value storage. On a 16-core machine, our table achieves almost 40 million inserts per second, outperforming the concurrent hash table in Intel's Thread Building Blocks by 2.5x, while using less than half of the memory for 64 bit key/value pairs. Figure 1 gives an example of how our scheme (cuckoo+) outperforms other hash tables with mixed random read/write workloads. Section 6 presents a performance evaluation detailing the advantages of this cuckoo-based approach for multicore applications.

2. Background and Related Work

This section provides background information on hash tables and concurrency control mechanisms. We conclude with a brief performance evaluation of the effects of naively applying standard concurrency control techniques to several common hash table implementations. These results remind that high-performance concurrency is not trivial: careful algorithm engineering is important regardless of the underlying concurrency control mechanisms, and the algorithmic effects dominate the choice of concurrency mechanism.

2.1 Hash Tables

As used in this paper, a *hash table* provides `Lookup`, `Insert`, and `Delete` operations for indexing all key-value objects. Hash tables do not support retrieval by any key ordering. Popular designs vary in their support for iterating through the hash table in the presence of concurrent modifications; we omit consideration of this feature.

Interface. On `Lookup`, a value is returned for the given key, or "does not exist" if the key cannot be found. On `Insert`, the hash table returns success, or an error code to indicate whether the hash table is too full or the key already exists. `Delete` simply removes the key's entry from the hash table. We focus on `Lookup` and `Insert`, as `Delete` is very similar to `Lookup`.

High-performance single-thread hash tables. As an example of a modern, extremely fast hash table, we compare in several places against Google's `dense_hash_map`, a hash table available in the Google SparseHash [9] library. Dense hash sacrifices space efficiency for extremely high speed: It uses open addressing with quadratic internal probing. It maintains a maximum 0.5 load factor by default, and stores entries in a single large array.

C++11 introduces an `unordered_map` implemented as a separate chaining hash table. It has very fast lookup performance, but also at the cost of more memory usage.

The performance of these hash tables does not scale with the number of cores in the machine, because only one writer or one reader is allowed at the same time.

Multiple-reader, single-writer hash tables. As a middle ground between no thread safety and full concurrency, single-writer tables can be extended to permit many concurrent readers. Such designs often use optimistic techniques such as versioning or the read-copy-update (RCU) [17] techniques becoming widely used within the Linux kernel.

Our work builds upon one such hash table design. Cuckoo hashing [18] is an open-addressed hashing technique with high memory efficiency and $O(1)$ amortized insertion time and retrieval. As a basis for its hashing, our work uses the multi-reader version of cuckoo hashing from MemC3 [8], which is optimized for high memory efficiency and fast concurrent reads.

Scalable concurrent hash tables. The Intel Threading Building Blocks library (Intel TBB) [2] provides a `concurrent_hash_map` that allows multiple threads to concurrently access and update values. This hash table is also based upon the classic separate chaining design, where keys are hashed to a bucket that contains a linked list of entries. This design is quite popular for concurrent hash tables: Because a key hashes to one unique bucket, holding a per-bucket lock permits guaranteed exclusive modification while still allowing fine-grained access. Further care must be taken if the hash table permits expansion.

2.2 Concurrency Control Mechanisms

As noted earlier, part of our motivation was to explore the application of hardware transactional memory to this core data structure. All concurrent data structures require some mechanism for arbitrating concurrent access, which we briefly list below, focusing on those used in this work.

Locking. Multi-threaded applications take advantage of increasing number of cores to achieve high performance. To ensure thread-safety, multiple threads have to serialize their operations when accessing shared data, often through the use of a critical section protected by a lock.

The simplest form of locking is to wrap a coarse-grained lock around the whole shared data structure. Only one thread can hold the lock at the same time. This tends to be pessimistic, since the thread with the lock prevents any other threads from accessing the shared resource, even if they only want to read the data or make non-conflicting updates.

Another option is to use fine-grained locking by splitting the coarse-grained lock into multiple locks. Each fine-grained lock is responsible for protecting a region of the data, and multiple threads can operate on different regions of the data at the same time. Fine-grained locking can improve the overall performance of a concurrent system. However, it must be carefully designed and implemented to behave correctly without deadlock, livelock, starvation, etc.

Hardware Transactional Memory (HTM). It is often hard to write fast and *correct* multi-threaded code using fine-grained locking. Transactional memory [10] is designed to make the creation of reliable multi-threaded programs easier. Much like database transactions, all shared memory accesses and their effects are applied *atomically*, i.e., they are either committed together or discarded as a group. With transactional memory, threads no longer need to take locks when accessing the shared data structures held in memory, yet the system will still guarantee thread safety.

Previous experience, implementations and evaluations of HTM include Sun's Rock [3, 6] processor, AMD advanced synchronization family [5, 4], IBM Blue Gene/Q [21] and System Z [13].

Recently, Intel released Transactional Synchronization Extensions (TSX) [1], an extension to the Intel 64 architecture that adds transactional memory support in hardware. Part of the recently-released Intel Haswell microarchitecture, TSX allows the processor to determine dynamically whether threads need to serialize through lock-protected critical sections, and to serialize *only when required*. With TSX, the program can declare a region of code as a transaction. A transaction executes and atomically commits all results to memory when the transaction succeeds, or *aborts* and cancels all the results if the transaction fails (e.g., conflicts occur). We focus on the use of Restricted Transactional Memory (RTM) interface of TSX, which gives the programmer the flexibility to start, commit and abort transactional execution. Intel evaluated TSX for high-performance computing workloads [22], already optimized for parallelism, and showed that TSX provides an average speedup of 1.41x.

2.3 Naive use of concurrency control fails

Before making deeper changes, we begin by examining the performance of several hash tables *without* algorithmic optimization, using both naive global locking and using Intel's TSX to optimize this approach. While the poor performance of these approaches is not surprising, their relative simplicity makes them an important starting baseline for understanding further improvements.

Haswell's hardware memory transactions are a best-effort model intended for fast paths. The hardware provides no guarantees as to whether a transactional region will ever successfully commit. Therefore, any transaction implemented with TSX needs a fallback path. The simplest fallback mechanisms is "lock elision": the program executes a lock-protected region speculatively as a transaction, and only falls back to use normal locking if the transaction does not succeed. An implementation of TSX lock elision for glibc [20] has been released. It adds a TSX elided lock as a new type of POSIX mutex. Applications linked against this new glibc library automatically have their pthread locks elided.

Lock elision may seem promising for designing a concurrent, multi-writer hash table: multiple threads may be able to update different sets of non-conflicting entries of the hash table at the same time. Through a set of experiments, we make two observations about TSX lock elision: It outperforms the naive use of a global lock, but it does not ensure that multicore concurrent writes are faster than single-core exclusive access.

We evaluated the `Insert` throughput of the optimistic cuckoo hash table in MemC3, `std::unordered_map` in C++11, and `dense_hash_map` in Google SparseHash [9] library, both with and without TSX lock elision, on a quad-core machine with hyperthreading enabled. All these hash tables allow only one writer at a time, as each `Insert` has to lock the entire table. Global counters were removed in cuckoo hash table and `dense_hash_map` to avoid obvious common data conflicts.

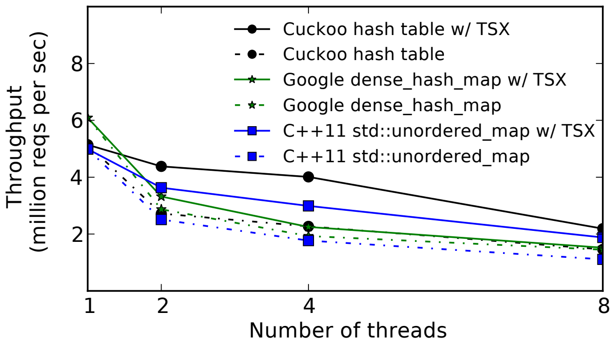


Figure 2: `Insert` throughput vs. number of threads for single writer hash tables with and without TSX lock elision. Each thread is pinned to a different hyper-threaded core. 16 million different keys are inserted in each table.

Figure 2 shows the results of our experiment. With global pthread locks, each hash table's multi-thread aggregate write throughput is much lower than that of a single thread, due to extensive lock contention. By enabling TSX lock elision, the aggregation write throughput is higher than that with pthread global locks, but still much lower than the single thread throughput. This is because most transactions fail and abort, forcing the program to take the fallback lock frequently, resulting in sequential behavior. According to Intel Performance Counter Monitor [12], the transactional abort rates are above 80% for all three hash tables with 8 concurrent writers. We will discuss the reasons for transactional aborts and how to reduce the abort rate in Section 5.

Through this experiment, we find that naively making a data structure concurrent may harm its performance. Simply applying lock elision using hardware transactional memory could mitigate the performance degradation caused by lock contention, but may not be able to scale up throughput as more cores access the same lock protected data structure.

3. Principles to Improve Concurrency

Given that naive application of global locking with or without hardware transactional memory support fails to provide scalable performance, what must be done? In this section we present our design principles to improve the concurrent performance of data structures. Although these principles are general and well known, we state them here to illustrate the framework within which our algorithmic engineering discussed in the next section optimizes for concurrent access in cuckoo hashing. In general, the key to improving concurrency for a data structure is to reduce lock contention. We present three principles to help achieve this reduction:

P1. *Avoid unnecessary or unintentional access to common* *data.* When possible, make globals thread-local; for example, disable instant global statistics counters in favor of lazily aggregated per-thread counters. These simple optimizations are already included in our results for cuckoo hash table and Google `dense_hash_map` in Figure 2. Without them, concurrent performance was much worse.

P2. *Minimize the size and execution time of critical sections.* A promising strategy is to move data accesses out of the critical section whenever possible. As we show in the following section, an optimistic approach can work well here if there are search-like operations that must be performed: Perform the entire search outside of a critical section, and then transactionally execute by only verifying that the found value remains unchanged.

P3. *Optimize the concurrency control mechanism.* Tune the concurrency control implementation to match the expected behavior of the data structure. For example, because the critical sections of our optimized hash tables are all very short, we use lightweight splinlocks and lock striping in the fine-grained locking implementation, and optimize TSX lock elision to reduce transactional abort rate when applying it to the coarse-grained locking implementation.

By following these principles, data structures can reduce the possibility of multiple threads attempting to access data protected by a shared lock or within a same transactional region, thus improve the concurrent performance with either fine-grained or coarse-grained locking. We show how to apply these principles to the design of a concurrent cuckoo hash table in the next two sections, to greatly improve multi-threaded read/write throughput.

4. Concurrent Cuckoo Hashing

We now present the design of a multi-reader/multi-writer cuckoo hash table that is optimized for fast concurrent writes. By applying the principles previously described, our resulting design achieves high and scalable multi-threading performance for both read- and write-heavy workloads.

We begin by presenting the basic operation of cuckoo hashing [18], followed by the multiple-reader/single-writer version that we build upon to create our final solution [8].

4.1 Cuckoo Hashing

Cuckoo hashing [18] is an open-addressed hash table design. All items are stored in a large array, with no pointers or linked lists. To resolve collisions, two techniques are used: First, items can be stored in one of two buckets in the array, and they can be moved to their other location if the first is full. Second, in common use, the hash buckets are multi-way set associative, i.e., each bucket has B "slots" for items. $B = 4$ is a common value in practice.[1] A lookup for `key` proceeds

[1]Without set-associativity, basic cuckoo hashing allows only 50% percent of the table entries to be occupied before unresolvable collisions occur. It is possible to improve the space utilization to over 90% by using 4-way (or higher) set associative hash table [7].

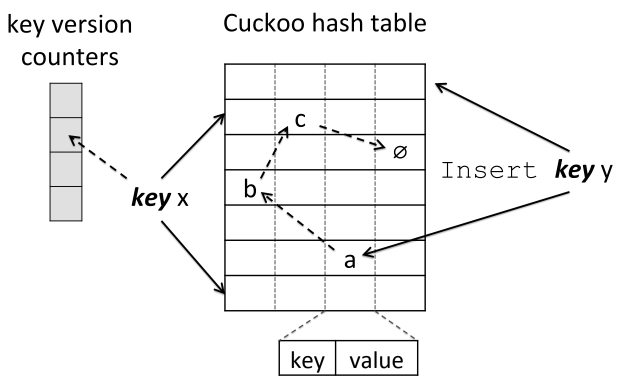


Figure 3: Cuckoo hash table overview: Each key is mapped to 2 buckets by hash functions and associated with 1 version counter. ∅ represents an empty slot. "$a \to b \to c \to \varnothing$" is a cuckoo path to make one bucket available to insert key y.

by computing two hashes of `key` to find buckets b_1 and b_2 that could be used to store the key, and examining all of the slots within each of those buckets to determine if the key is present. A basic "2,4-cuckoo" hash table (two hash functions, four slots per bucket) is shown in Figure 3.

A consequence of this is that `Lookup` operations are both fast and predictable, always checking $2B$ keys.

To `Insert` a new key into the table, if either of the two buckets has an empty slot, it is then inserted in that bucket; if neither bucket has space, a random key from one candidate bucket is displaced by the new item. The displaced item is then relocated to its own alternate location, possibly displacing another item, and so on, until a maximum number of displacements is reached. If no vacant slot is found, the hash table is considered too full to insert and an expansion process is scheduled.

We call the sequence of displaced keys in an `Insert` operation a *cuckoo path*, as illustrated in Figure 3. Write performance of cuckoo hashing degrades as the table occupancy increases, since the cuckoo path length will increase, and more random reads/writes are needed for each `Insert`.

4.2 Prior Work in Concurrent Cuckoo

Basic cuckoo hashing does not support concurrent access. Our work builds upon the two major prior approaches to concurrent cuckoo hashing: Herlihy's lock-striped approach [11], and the optimistic cuckoo with separated path discovery and item movement from MemC3 [8]. Our resulting design realizes the strengths of each: The low space overhead of MemC3's approach and the concurrent writer support of Herlihy's approach.

Our starting point was MemC3's table, which used three building blocks:

- *To eliminate reader/writer false misses, change the order of the basic cuckoo hashing insertions.* Allow concurrent reads and cuckoo movement by moving "holes" backwards along the cuckoo path instead of moving "items" forward along the cuckoo path. This ensures that an item can always be found by a reader thread; if it is undergoing concurrent cuckoo movement, it may be present twice in the table, but will never be missing.

Providing this property requires separating the process of searching for a cuckoo path from using it: find the empty slot, and *then* use the path. As we show, this has a second benefit: This searching process can be moved outside of the critical section.

- *Implement efficient concurrency control by using lock striping.* Lock striping [11] uses a smaller vector of locks (or, in MemC3, version counters) that each maps to a set of items in the hash table. To lock a bucket, a writer thread computes the lock stripe entry corresponding to the bucket and locks that entry. By using reasonable size lock tables, such as 1K-8K entries, the locking can be both very fine-grained and low-overhead.
- *Allow reads to be performed with no cache line writes by using optimistic locking [14].* Instead of locking for reads, the hash table uses a lock-striped version counter associated with the buckets, updates it upon insertion or displacement, and looks for a version change during lookup.

By using these techniques with *only* version counters and a simple global lock for writers, MemC3 provided substantial gains for read-intensive workloads, but still performed poorly for write-heavy workloads. Unfortunately, the basic scheme used in MemC3 was not obviously amenable to fine-grained locking:

1. The cuckoo path can be very long. Grabbing a few hundred locks in the right order to avoid deadlock and livelock is tricky. There is also a nontrivial probability that a path becomes invalid, and the execution of `Insert` needs to restart, further complicating locking, increasing the risk of livelock, and harming performance.
2. The `Insert` procedure for optimistic concurrent cuckoo hashing in MemC3 [8] involves nested locks if fine-grained locking is implemented, which can easily cause deadlocks.

4.3 Algorithmic Optimizations

4.3.1 Lock After Discovering a Cuckoo Path

In MemC3 cuckoo hashing, each `Insert` operation locks the hash table at the very beginning of the process, and releases the lock after the insertion completes. The separated phases of search and execution of the cuckoo path are all protected by the lock within one (big) critical section.

To reduce the size of critical sections, our first optimization was to search for an empty slot before acquiring the lock, then only lock the table when displacing the items along the cuckoo path and inserting the new item. In this way, multiple `Insert` threads can look for their cuckoo paths at the same time without interfering with each other. Inserts are still serialized, but the critical section is smaller.

Algorithm 1 shows the basic `Insert` procedure that allows concurrent reads. ADD(h, b, x) tries to insert key x to bucket b, returns $true$ on success or $false$ if the bucket is full. SEARCH$(h, b_1\ b_2)$ searches for a cuckoo path that makes either bucket b_1 or b_2 available to insert a new item. EXECUTE$(h, path)$ moves items backwards along the cuckoo path, and then inserts key x to the bucket made available. The critical section of this algorithm is the whole process. When the table occupancy is high, this may involve hundreds of bucket reads to search for a cuckoo path, followed by hundreds of item displacements along that path, during which all `Insert` operations of other threads are blocked.

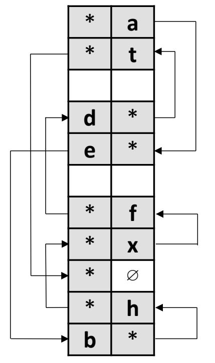
Algorithm 1 MemC3 Cuckoo Insert Procedure.
Region between dashed lines is the largest possible critical section.

```
 1: function INSERT($h$, $x$)                         ▷ Insert key $x$ to table $h$
 2:     $b_1$, $b_2$ ← two buckets mapped by key $x$
 3:     LOCK($h$)
        - - - - - - - - - - - - - - - - - - - - - - - - -
 4:     if ADD($h$, $b_1$, $x$) or ADD($h$, $b_2$, $x$) then
 5:         UNLOCK($h$); return $true$
 6:     if $path$ ←SEARCH($h$, $b_1$, $b_2$) then
 7:         EXECUTE($h$, $path$)
        - - - - - - - - - - - - - - - - - - - - - - - - -
 8:         UNLOCK($h$); return $true$
 9:     UNLOCK($h$); return $false$
```

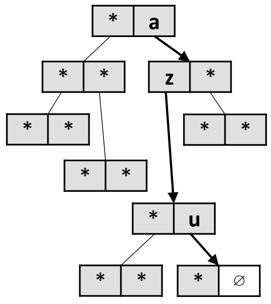
Algorithm 2 Cuckoo Insert – lock after discovering a path.
Region between dashed lines is the largest possible critical section.

```
 1: function INSERT($h$, $x$)                         ▷ Insert key $x$ to table $h$
 2:     $b_1$, $b_2$ ← two buckets mapped by key $x$
 3:     for $i \leftarrow 1,2$ do
 4:         if AVAILABLE($h$, $b_i$) then             ▷ if $b_i$ has an empty slot
 5:             LOCK($h$)
 6:             if ADD($h$, $b_i$, $x$) then
 7:                 UNLOCK($h$); return $true$
 8:             UNLOCK($h$)
 9:     while $path$ ←SEARCH($h$, $b_1$, $b_2$) do
10:         LOCK($h$)
            - - - - - - - - - - - - - - - - - - - - - - - - -
11:         if VALIDATE_EXECUTE($h$, $path$) then
            - - - - - - - - - - - - - - - - - - - - - - - - -
12:             UNLOCK($h$); return $true$
13:         UNLOCK($h$)
14:     return $false$
```

Algorithm 2 shows our new `Insert` procedure. The lock is acquired only when doing the actual writes to the hash table. As the search phase is not protected by the lock, there exists a potential race condition: After one thread reads a bucket to extend its cuckoo path, another thread can write to the same bucket and cause the first thread to read corrupted data. Therefore, `Insert` must re-check if the related entries have been modified before each item displacement in the execution phase, which is handled by VALIDATE_EXECUTE$(h, path)$. If the existing path becomes invalid, it restarts and looks for a new path. Each displacement relocates only one item to



(a) Random displacements



(b) Breadth-first search

Figure 4: Search for an empty slot by `Insert` in a 2-way set-associative hash table. Left(4a) is the traditional approach, right(4b) is our approach. Slots in gray are examined before the empty slot is found. Alphabet letters are keys selected to be moved to their alternate locations along the cuckoo path represented by the arrows ($\rightarrow$).

its alternate bucket, so there is no undo needed if execution aborts. We omit the steps to check if key x already exists in both Algorithm 1 and 2, which should be proceeded within each critical section.

To summarize, each `Insert` optimistically searches for a cuckoo path, displacing items along the path with lock protection. Execution terminates at the end of the path or if the path becomes invalid (and then `Insert` restarts). How often does a path become invalid after being discovered? We can estimate the probability that a cuckoo path of one writer overlaps with paths of other writers: Let N denote the number of entries in the hash table, L ($\ll N$) denote the maximum length of a cuckoo path, and T denote the number of threads. For a given writer, the maximum probability that its cuckoo path overlaps with at least one of other paths is as below (derivation can be found in Appendix B).

$$P_{invalid\_max} \approx 1 - \big((N-L)/N\big)^{L(T-1)} \qquad (1)$$

For example, the maximum length of a cuckoo path in MemC3 is $L = 250$. Suppose $N = 10$ million, $T = 8$, then $P_{invalid} < 4.28\%$. This upper bound assumes all paths are at maximum length, which occurs only rarely; the expected probability is much lower. It is, however, non-negligible. We apply further algorithmic optimizations next to reduce the odds of such a failure by several orders of magnitude.

4.3.2 Breadth-first Search for an Empty Slot

Basic cuckoo hashing searches for an empty slot using a greedy algorithm: if the current bucket is full, a random key is "kicked out" to its alternate location, and possibly kicks out another random key there, until a vacant position is found. Each bucket touched by the process is a part of the cuckoo path. As table occupancy grows, the average length of cuckoo paths increases, because it needs to examine more buckets to find an empty slot. It may require hundreds of displacements for one `Insert`, which greatly slows down the performance.

A cuckoo hash table can be viewed as an undirected graph called a *cuckoo graph*, which has a vertex for each bucket, and an edge for each key in the table, connecting the two alternative buckets of the key. The "random displacements" scheme used by basic cuckoo hashing to look for an empty slot is thus a random *depth-first search* (DFS) of the graph. To reduce the number of item displacements and the size of critical sections, we use *breadth-first search* (BFS) instead. Each slot in a bucket is considered as a possible path, and extends its own path to alternate buckets in the same way.

Figure 4 shows an example of the two searching schemes in a 2-way set-associative hash table. Both schemes examine 18 slots (9 buckets) to find an empty slot in the search with no item actually moved. Figure 4a is the traditional searching scheme where each time only one random key is displaced. The cuckoo path discovered is $a \rightarrow e \rightarrow b \rightarrow h \rightarrow x \rightarrow f \rightarrow d \rightarrow t \rightarrow \varnothing$. Figure 4b uses BFS to look for an empty slot. While the number of examined slots are same, the BFS cuckoo path is $a \rightarrow z \rightarrow u \rightarrow \varnothing$, which is much shorter.

The prior work on MemC3 used an optimization of tracking two cuckoo paths in parallel, completing when either found an empty slot, but still used a DFS strategy. This strategy, in general, reduced the expected length of a cuckoo path by a factor of two. In contrast, the BFS strategy we present here reduces the expected length to *a logarithmic* factor: For a B-way set-associative cuckoo hash table, where the maximum number of slots to be checked to look for an available bucket before declaring the table is too full is M, then the maximum lengths of cuckoo paths from BFS is as below (derivation can be found in Appendix C).

$$L_{\text{BFS}} = \left\lceil \log_B \big(M/2 - M/(2B) + 1\big) \right\rceil. \qquad (2)$$

As used in MemC3, $B = 4$, $M = 2000$. With two-way DFS, the maximum number of displacements for a single `Insert` is 250, whereas with $L_{\text{BFS}} = 5$.

This optimization is key to reducing the size of the critical section: While the total number of slots examined is still M, this is work that can be performed without a lock held. With BFS, however, at most five buckets must be examined and modified with the lock actually held, reducing both the duration of the critical section and the number of cache lines dirtied while doing so.

Shorter cuckoo paths also reduce the chance of a path becoming invalid (and of transactional aborts). Based on Eq. 1, with $L_{\text{BFS}} = 5$, and the same settings of the example at the end of §4.3.1 , the new worst-case $P_{invalid} < 1.75 \times 10^{-5}$ — an extremely rare event.

Prefetching. BFS provides a second benefit: because the schedule of buckets to visit is predictable, we can prefetch buckets into cache before they are accessed to reduce the cache-miss penalty. In the *cuckoo graph*, each alternative bucket of the keys in the current bucket are considered *neighbors* of that bucket. BFS scans all neighbors of a bucket to extend the cuckoo path. Before scanning one neighbor, the processor can load the *next_neighbor* in cache, which will be accessed soon if no empty slot is found in the current neighbor. This cannot be done with the traditional DFS approach, because the next bucket location is unknown until one key in the current bucket is "kicked out".

4.3.3 Increase Set-associativity

As discussed in §4.1, higher set-associativity improves space utilization. Then cuckoo hash table in MemC3 is 4-way set-associative, which achieves 95% maximum load factor, and high performance for read-intensive workloads.

The impact of set-associativity on the read and write performance of cuckoo hashing is two-fold:

- Higher set-associativity leads to *lower* read throughput, since each `Lookup` must scan up to $2B$ slots from two buckets in an B-way set-associative hash table. If a bucket fits in a cache line, then the read throughput would not be affected too much.
- Higher set-associativity may *improve* write throughput, because each `Insert` can read fewer random buckets (with fewer cache misses) to find an empty slot, and needs fewer item displacements to insert a new item. However, the set-associativity cannot be too high, since a `Lookup` is required to check if the new key already exits in the hash table before each `Insert`, which becomes slower as set-associativity increases.

To achieve a good balance between read- and write-heavy workloads, we use a 8-way set-associative hash table. §6 evaluates the performance with different set-associativities and different workloads. Our choice of 8-way associativity may require reading more than one cache line per bucket, but this extra cost is offset by the fact that the two lines can be fetched together, costing only memory bandwidth, not latency, and that sequential memory reads are substantially faster because they typically hit in the DRAM row buffer.

4.4 Fine-grained Locking

Fine-grained locking is often used to improve concurrency. However, it is non-trivial to implement fine-grained per-bucket locking for traditional cuckoo hashing. There are high deadlock and livelock risks.

In basic cuckoo hashing, it is not known before displacing the keys how many and which buckets will be modified, because each displaced key depends on the one previously kicked out. Therefore, standard techniques to make `Insert` atomic and avoid deadlock, such as acquiring all necessary locks in advance, are not obviously applicable. As noted earlier, simply using the optimization of finding the path in advance was not enough to solve this problem because of lingering locking complexity issues.

By reducing the length of the cuckoo path and reordering the locking procedure, our optimizations make fine-grained locking practical. To do so, we go back to the basic design of lock-striped cuckoo hashing and maintain an actual lock in the stripe *in addition* to the version counter (our lock uses the high-order bit of the counter). Here we favor spinlocks using compare-and-swap over more general purpose mutexes. A spinlock wastes CPU cycles spinning on the lock while other writers are active, but has low overhead, particularly for uncontended access. Because the operations that our hash tables support are all very short and have low contention, very simple spinlocks are often the best choice.

To `Insert` each new key-value pair, there is at most one new item inserted and four item displacements. Each insert or displacement involves exactly two buckets. The `Insert` operation only locks the pair of buckets associated with ongoing insertion or displacement, and releases the lock immediately after it completes, before locking the next pair. Locks of the pair of buckets are ordered by the bucket id to avoid deadlock. If two buckets share the same lock, then only one lock is acquired and released during the process. In summary, a writer must only lock at most five (usually fewer than three) pairs of buckets sequentially for an `Insert` operation.

Although there is a small chance that any cuckoo insert will abort because of other concurrent inserts, it is likely to succeed on a re-try. It is worth noting that this design only avoids livelock probabilistically. A writer thread that encounters excessive insert aborts *could* pessimistically acquire a full-table lock by acquiring each of the 2048 locks in the lock-striped table, but we have never observed a condition where this would be warranted.

The combination of these techniques results in a cuckoo hash table that (i) retains high memory efficiency (the efficiency of the basic table plus the small additional lock-striping table), (ii) permits highly concurrent read-write access, and (iii) has a minimally-sized critical section that reads and dirties few cache lines while holding the lock or executing under hardware transactional memory.

5. Optimizing for Intel TSX

As shown in §2.3, naive use of TSX lock elision to hash tables with a global lock does not provide high multi-threaded throughput. The key to improving concurrent performance is to reduce the "transactional abort rate." In the Haswell implementation of TSX, the underlying hardware transactional memory system uses tags in the L1 cache to track the read- and write-sets of transactions at a granularity of a cache line. Transactions abort for three common reasons:

1. *Data conflict on a transactionally accessed address.* A transaction encounters a conflict if a cache line in its read-set is written by another thread, or if a cache line in its write-set is read or written by another thread.
2. *Limited resources for transactional stores.* A transaction will abort if there is not enough space to buffer its reads and writes in cache. Current implementations can track only 16KB of data.
3. *TSX-unfriendly instructions.* Several instructions (e.g.,

XABORT, PAUSE) and system calls (e.g., mmap) cause transactions to abort.

For high performance, the program must minimize transactional aborts. From the first two causes, we draw several conclusions about general issues with transactional aborts:

- Transactions that touch more memory are more likely to conflict with others, as well as to exceed the L1-cache-limited capacity for transactional reads and writes.
- Transactions that take longer to execute are more likely to conflict with others.
- Sharing of commonly-accessed data, such as global statistics counters, can greatly increase conflicts.
- Because the hardware tracks reads and writes at the granularity of a cache line, false sharing can create transactional conflicts even if no data appears to be shared.

The observant reader will no doubt note that many of these same issues arise in cache-centric performance optimizations. Our solutions are similar but not identical. To address these issues and improve the multi-threaded concurrent performance of cuckoo hashing with coarse-grained locking and TSX lock elision enabled, we just need to follow principle P1 and P2 presented in §3, which are detailed in §4. Our algorithmic optimizations can significantly reduce the size of the transactional region in a cuckoo `Insert` process from hundreds of bucket reads and writes to only a few bucket writes, which greatly reduces the transactional abort rate caused by data conflicts or limited transactional stores.

The third cause of transactional abort indicates that a program should *minimize the occurrence of TSX-unfriendly instructions within transactional regions.* A common example is if dynamic memory allocation must invoke a system call such as `brk`, `futex`, or `mmap`. While our implementation of Cuckoo hashing does not do this, we observed this problem when testing TSX using chained hashing and Masstree [16]. It is therefore useful to pre-allocate structures that may be needed inside the transactional region. If they are not used, one can simply store them in a per-thread cache and use for a subsequent transaction (or preallocate and free if using a malloc that already does this, such as `tcmalloc`). This is an application of principle P3.

Further, we *use a tuned version of TSX lock elision that matches the expected behavior of the data structure.* The generic glibc version of TSX lock elision for pthread mutexes can be improved substantially if the application's transactional behavior is known in advance, as is the case for our optimized cuckoo hash table, in which every transaction is small. This is another application of principle P3. We detail our implementation of TSX lock elision in Appendix A.

6. Evaluation

In this section, we investigate how the proposed techniques and optimizations contribute to the improvements of read and write performance in cuckoo hashing.

Platform. Most experiments (except Figure 7) run on a 4-core Haswell-microarchitecture Intel i7-4770 at 3.4GHz. This is the highest core count currently available with TSX support. The L1 D-cache is 32KB; the L2 cache is 256KB, the L3 cache is 8MB. The machine is equipped with 16GB of DDR3 SDRAM.

Method and Workloads. 8 byte keys and 8 byte values are used for most experiments. The default cuckoo hash table is 8-way set-associative with $2^{27} = 134,217,728$ slots, which uses about 2 GB memory. Each bucket has all the keys come first and then the values, and fits exactly two cache lines: one for 8 keys and another for 8 values. We evaluate different set-associativities in §6.3 and key-value sizes in §6.4.

We focus on the performance benefit from our optimizations and TSX support for workloads with concurrent writes by measuring the aggregate throughput of multiple threads accessing the same hash table. We focus on three workloads: a) 100% `Insert`, b) 50% `Insert` and 50% `Lookup`, and c) 10% `Insert` and 90% `Lookup`.

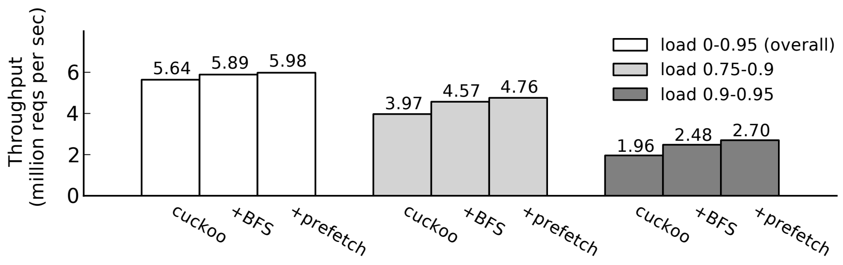
Each experiment first creates an empty cuckoo hash table and then fills it to 95% capacity, with random mixed concurrent reads and writes as per the specified insert/lookup ratio. Because Cuckoo hashing slows down as the table fills (more items must be moved), we measure both overall throughput and throughput for certain load factor intervals (e.g., empty to 50% full). Each data point in the graphs of this section is the average of 10 runs. We observed that the performance is always stable, so we do not include error bars in the graphs.

6.1 Factor Analysis of Insert Performance

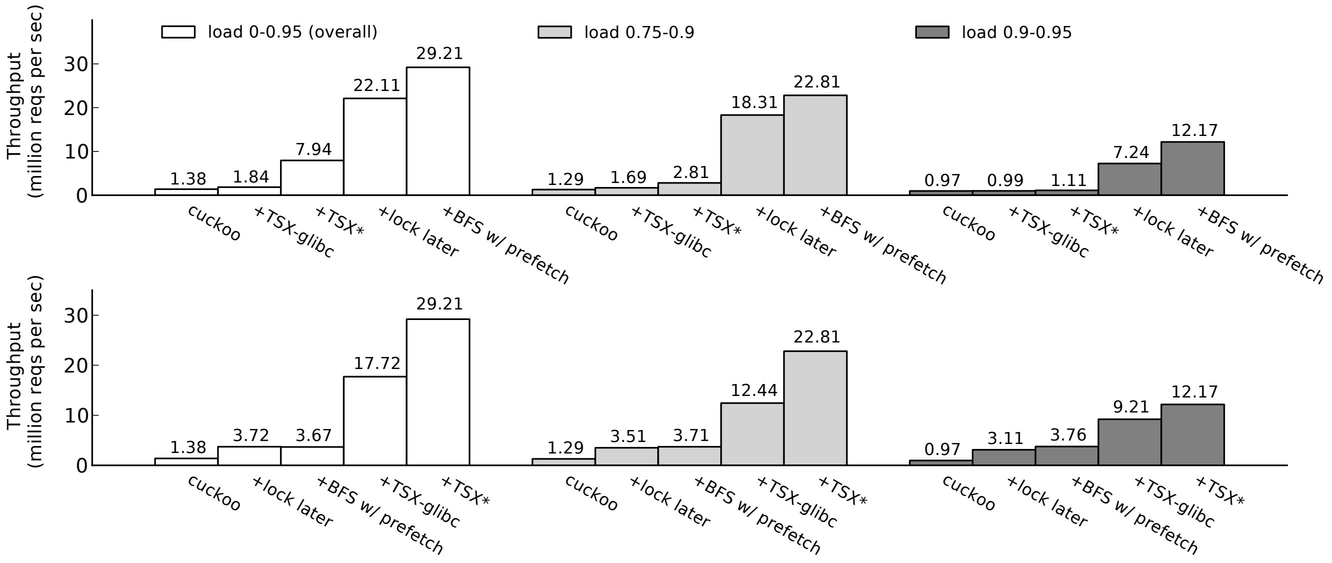
This experiment investigates how much our optimizations and the use of Intel TSX improve the `Insert` performance of cuckoo hashing. We break down the performance gap between basic optimistic cuckoo hashing and our optimized concurrent cuckoo hashing. We measure different hash table designs with the `Insert`-only workload starting from the basic cuckoo and adding optimizations cumulatively as follows:

- **cuckoo**: The optimistic concurrent multi-reader/single-writer cuckoo hashing used in MemC3 [8]. Each `Insert` locks the whole hash table.
- **+lock later**: Lock after discovering a cuckoo path.
- **+BFS**: Look for an empty slot by breadth-first search.
- **+prefetch**: Prefetch the next bucket into cache.
- **+TSX-glibc**: Use the released glibc TSX lock elision [20] to support concurrent writers.
- **+TSX***: Use our TSX lock elision implementation that is optimized for short transactions (Appendix A) instead of TSX-glibc.

Single-thread `Insert` performance is shown in Figure 5a. All locks are disabled, so "lock later" and "TSX" do not apply here. At high load factors, BFS improves single-thread



(a) Single thread `Insert` performance (all locks disabled)



(b) Aggregate `Insert` performance of 8 threads, with locking

Figure 5: Contribution of optimizations to the hash table `Insert` performance. Optimizations are cumulative.

write performance by $\sim 26\%$, and data prefetching further increases the throughput by $\sim 9\%$.

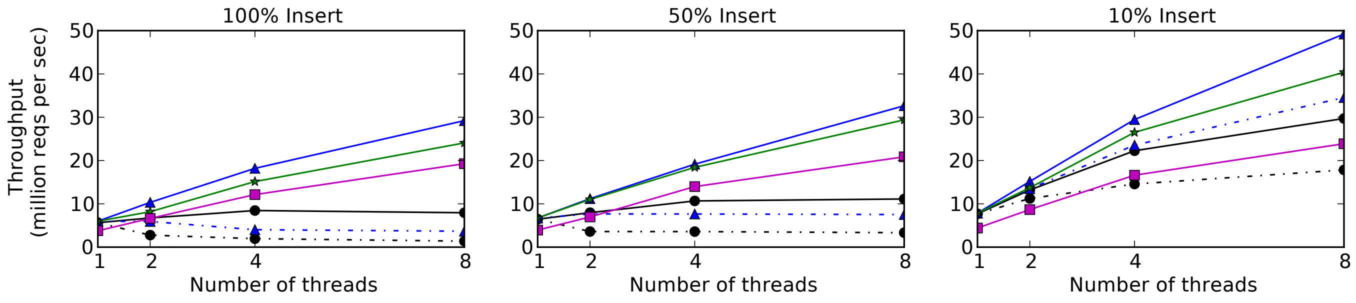
At low table occupancy, these optimizations are less important. In most cases, there are plenty of empty slots, and so no keys need to be moved. Further, when the cuckoo paths are all short, there is no savings in item motion to outweigh the slightly increased search cost of BFS over DFS. At high occupancy, BFS substantially reduces the number of item displacements, and prefetching is more useful because more buckets need to be evaluated as insertion candidates.

Multi-thread insert performance is shown in Figure 5b, measured by aggregating the throughput from 8 threads accessing the same hash table. A global lock is used for each `Insert` in the optimistic cuckoo hashing. Due to lock contention, the multi-threaded aggregate throughput of the optimistic cuckoo hashing is much lower than the single-thread throughput. The performance difference between the original optimistic cuckoo hashing scheme and optimized cuckoo hashing with TSX lock elision is roughly $20\times$.

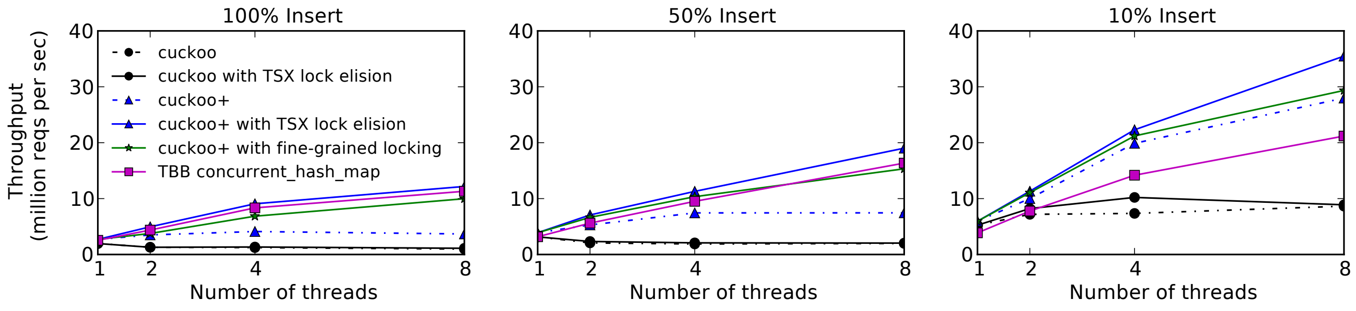
To understand the source of these benefits, the upper plot of Figure 5b shows the optimization sequence with lock elision enabled first and algorithmic optimizations applied later. With no algorithmic optimizations, using the customized TSX* elision improves overall throughput by $\sim 4.3\times$ over basic TSX lock elision. Comparing the top and bottom figures, when TSX* is applied *after* our algorithmic changes, it still improves throughput by almost 2x. This demonstrates the importance of using TSX in a way that is well-matched to the properties of the transactions it is handling. The improvements from fine-grained locking (not shown) are similar to those from applying TSX*, but slightly slower.

Simply reducing the size of the critical section *without* TSX or fine-grained locking results in only modest improvements (bottom graph, far left): from 1.38 to 3.7 million operations per second. However, once the system is capable of supporting fine-grained concurrent access, the improvement from algorithmic improvements is large (top graph, far left): from 7.94 to 29.2 million operations per second.

High performance is a consequence of both sufficiently fine-grained concurrency *and* data structures optimized to make that concurrency efficient. Neither of these optimizations alone was able to achieve more than 8 million operations per second, but they combine to achieve almost 30 million. Of particular note was that the algorithmic improvements needed



(a) Average throughput to fill the table from 0% to 95% occupancy.



(b) Average throughput at high table occupancy ($0.9\% - 0.95\%$).

Figure 6: Throughput vs. number of threads. “cuckoo” is the optimistic cuckoo hashing used in MemC3, “cuckoo+” is cuckoo with optimizations in §4.3. TSX lock elision is the optimized version in Appendix A. The cuckoo hash table is 2 GB with $\sim$ 134.2 million slots. Table occupancy is for cuckoo hashing only. TBB concurrent_hash_map is inserted with the same number and size of key-value pairs, with $2\times$ to $3\times$ more memory used than cuckoo hash table.

here were concurrency-specific: Without concurrency, for example, the BFS changes were performance-neutral, but with fine-grained locking, BFS increased performance by over 30%.

This latter conclusion is particularly true under high contention: The rightmost graphs in the figure show the performance improvements for the highly-loaded portion of the hash table fill, growing from 90% to 95% (a load factor that might occur with a heavy insert/delete workload). In this case, the performance gains of the algorithmic engineering are even more important: The high contention means that TSX alone encounters frequent aborts, only improving performance by about 10%. The algorithmic optimizations then provide a roughly 11x improvement.

6.2 Multi-core Scaling Comparison

This section evaluates hash table performance under an increasing number of cores, comparing both our original and optimized table, and also the Intel TBB [2] `concurrent_hash_map` for comparison. We initialize the TBB table with the same number of buckets and key-value type, then operate with the same workloads.

Cuckoo+ scales well as the number of cores increases, on both our 4-core Haswell machine (Figure 6), as well as when using fine-grained locking on a 16-core Xeon machine without TSX support (Figure 7). On the Haswell machine, the performance increase from 4 to 8 cores is slightly lower than up to 4 cores because there are only 4 physical cores.

In comparison, the basic *optimistic cuckoo hash table* scales poorly for a write-heavy workload, even using TSX lock elision. As shown in Figure 6, its total `Insert` throughput actually drops as more cores are used, except for the read-heavy workloads (rightmost graphs) for which its optimistic design works well. Notably, however, even under 10% inserts, cuckoo+ still substantially outperforms optimistic cuckoo.

The fine-grained locking version of Cuckoo+ also scales well for all workloads. Its absolute performance is up to 20% less than the TSX-optimized version, however, suggesting that there is a non-negligible benefit from hardware support.

To put these numbers in perspective, we also compare against the Intel Thread Building Blocks hash table. This comparison is slightly unfair: TBB supports concurrent iteration and other features that our hash table does not, but at a high level, it demonstrates both that our table's performance is good (it outperforms TBB substantially), particularly for read-intensive workloads, and that Cuckoo+ retains the memory efficiency advantages of the core Cuckoo design: It uses $2-3\times$ less memory for these small key-value objects, occupying only about 2GB of DRAM versus TBB's 6GB.

The results in Figure 7 show that these results also extend to larger machines, using a dual-socket Xeon server with 16 total cores, each a bit slower than those in the Haswell machine. Neither server has perfect speedup after 8 cores—memory

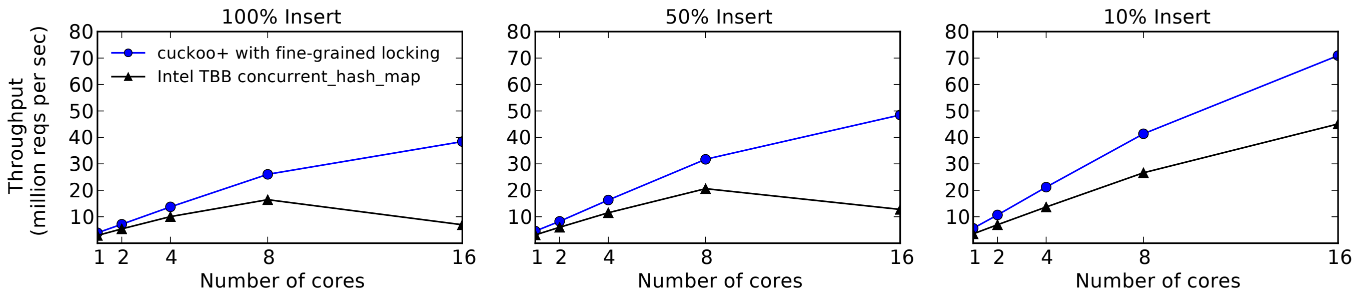


Figure 7: Overall throughput vs number of cores. On a 16-core machine without TSX support.

operations begin to traverse the QPI interconnect between the sockets—but Cuckoo+ continues to scale for write-heavy workloads where TBB scales only for read-heavy workloads.

6.3 Set-associativity and Load Factor

In this section, we evaluate the impact of set-associativity and load factor on cuckoo hashing performance, using the optimized cuckoo hashing with TSX lock elision. The experiments use the same workloads and hash table with same number of slots as before.

Figure 8 shows the aggregate `Lookup`-only throughput of 8 threads for 4- 8- and 16-way set associative hash tables, all at 95% table occupancy. As expected, lower associativity improves throughput, because each reader needs to check fewer slots in order to find the key. Each `Lookup` in a 4-way set-associative hash table needs at most two cache line reads to find the key and get the value. Each `Lookup` in a 8-way set-associative hash table needs at most two cache line reads to find the key and one more cache line read to get the value. Each `Lookup` in a 16-way set-associative hash table needs at most four cache line reads to find the key and one more cache line read to get the value.

Figure 9 shows the 8-thread aggregate throughput of table with different set-associativities, for different workloads at different table occupancy. Write performance degrades as the table occupancy increases, since an `Insert` operation has to read more buckets to find an empty slot, and needs more item displacements to insert the new key.

The load factor is important in this discussion because of the different use modes for hash tables: Some applications may simply fill the table in one go and then use it (perhaps modifying inserted values but not deleting keys), thus caring more about total insert rate. Others may issue inserts and deletes to a table at high occupancy, thus caring more about 90%-95% insert throughput.

Our results show that 8-way set-associativity has the best overall performance. It always outperforms 4-way set-associativity for 100% and 50% `Insert` workloads, and for 10% `Insert` workloads when the load factor is above 0.85. 16-way set-associativity always performs worst at low or moderate table occupancy. It starts to outperform 4-way set-associativity when the load factor is above 0.75, and achieves the highest throughput for write-heavy workloads when the

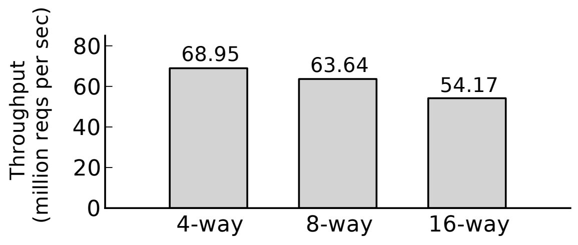


Figure 8: 8-thread aggregate `Lookup` throughput of hash tables with different set-associativities at 95% occupancy. Use optimized cuckoo hashing with TSX lock elision.

load factor is above 0.92. We therefore use 8-way associativity as our default because of its generality.

6.4 Different Key-Value Sizes

All previous experiments used workloads with 8 byte keys and 8 byte values. In this section, we evaluate the cuckoo hash table performance with different value sizes. Figure 10 shows the results of our two experiments.

In Figure 10a, we configure the hash table with 2^{25} entries, show throughput as the value size increases from 8 bytes to 256 bytes. As expected, the throughput decreases as the value size increases because of the increased memory bandwidth needed. On our 4-core machine, hyperthreading becomes much less effective with large values, because the machine runs out of memory bandwidth, and so performance scales only to the point of running one thread on each of the 4 physical cores. For example, with 256 byte values, single-thread throughput is 3.05 millions reqs per second, 4-thread throughput is 3.6× higher than 1-thread throughput, but 8-thread throughput is only 27% higher than 4-thread throughput.

Figure 10b reveals an interesting consequence of our current design when used with TSX: Large values increase the amount of memory touched during the transaction and therefore increase the odds of a transactional abort. For this experiment, we fix the hash table at 4GB and increase the key-value pair size to 1024 bytes. TSX lock elision outperforms fine-grained locking with small key-value sizes, but is worse at 1024 bytes. Improving our table design to reduce this effect seems a worthwhile area of future improvement.

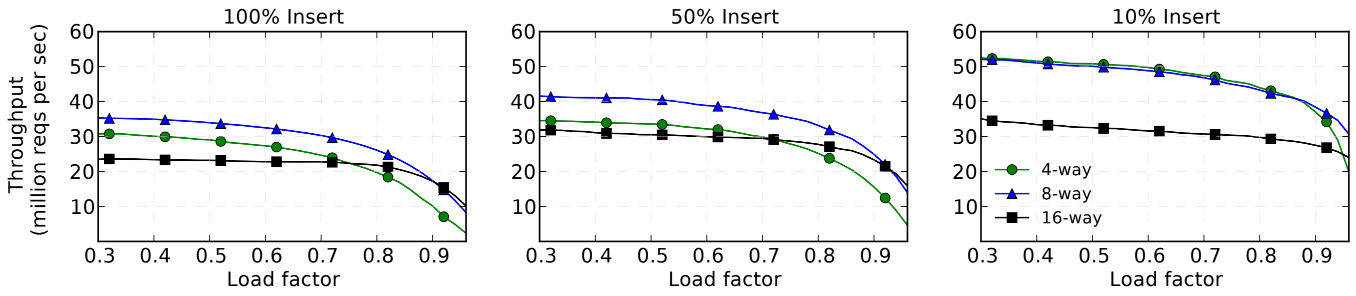
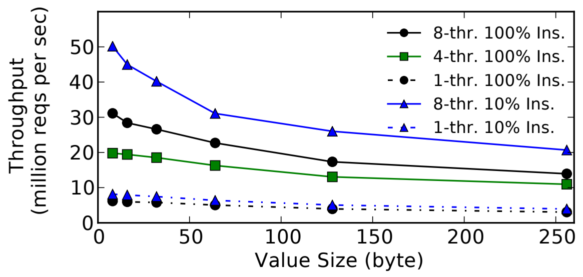
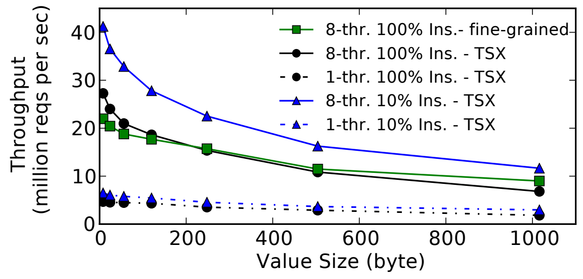


Figure 9: 8-thread aggregate throughput of hash tables with different set-associativities at different table occupancy. Use optimized cuckoo hashing with TSX lock elision.



(a) Hash table with fixed number ($\sim$ 33.4 million) of entries, using optimized cuckoo hashing with TSX lock elision.



(b) Hash table with fixed size (4 GB), using optimized cuckoo hashing with fine-grained locking or TSX lock elision.

Figure 10: Throughput with 8 byte keys and different sizes of values. *thr* stands for thread, *ins* for insert.

7. Discussion and Implementation Availability

Our results about TSX can be interpreted in two ways. On one hand, in almost all of our experiments, hardware transactional memory provided a modest but significant speedup over either global locking or our best-engineered fine-grained locking, and it was easy to use. This confirms other recent results showing, e.g., a "free" 1.4x speedup from using TSX in HPC workloads [22]. On the other hand, the benefits of data structure engineering for efficient concurrent access contributed substantially more to improving performance, but also required deep algorithmic changes to the point of being a research contribution on their own.

The focus of this paper was on the algorithmic and systems changes needed to achieve the highest possible hash table performance. As is typical in a research paper, this results in a fast, but somewhat "bare-bones" building block with several limitations, such as supporting only short fixed-length key-value pairs. To facilitate the wider applicability of our results, one of our colleagues has, subsequent to the work described herein, incorporated this design into an open-source C++ library, libcuckoo [15]. The libcuckoo library offers an easy-to-use interface that supports variable length key value pairs of arbitrary types, including those with pointers or strings, provides iterators, and dynamically resizes itself as it fills. The price of this generality is that it uses locks for reads as well as writes, so that pointer-valued items can be safely dereferenced, at the cost of a 5-20% slowdown. Specialized applications will, of course, still get the most performance using the hybrid locking/optimistic approach described herein, and part of our future work will be to provide one implementation that provides the best of both of these worlds.

8. Conclusion

This paper describes a new high performance, memory-efficient concurrent hash table based on cuckoo hashing. We demonstrate that careful algorithm and data structure engineering is a necessary first step to achieving increased performance. Our re-design minimizes the size of the hash table's critical sections to allow for significantly increased parallelism. These improvements, in turn, allow for two very different concurrency control mechanisms, fine-grained locking and hardware transactional memory. On a 16-core machine, with write heavy workloads, our system outperforms existing concurrent hash tables by up to 2.5x while using less than half of the memory for small key-value objects.

Acknowledgments

The authors are grateful to Manu Goyal for his critical contributions to the release of the cuckoo hashing C++ library, as well as to Bin Fan, Dong Zhou, Hyeontaek Lim, our shepherd Frank Dabek and the anonymous reviewers for their insights and helpful comments. This work was supported by National Science Foundation Awards CSR-0953197 (CAREER) and CCF-0964474, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] *Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*. Number 253665-047US. Intel Corporation, June 2013.

[2] Intel Threading Building Block. https://www.threadingbuildingblocks.org/.

[3] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, Mar. 2009.

[4] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proc. 5th EuroSys*, pages 27–40, 2010.

[5] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proc. 43rd MICRO*, pages 39–50, 2010.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proc. 14th ASPLOS*, pages 157–168, 2009.

[7] U. Erlingsson, M. Manasse, and F. McSherry. A Cool and Practical Alternative to Traditional Hash Tables. In *Proc. 7th Workshop on Distributed Data and Structures (WDAS'06)*, Santa Clara, CA, Jan. 2006.

[8] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing. In *Proc. 10th USENIX NSDI*, Lombard, IL, Apr. 2013.

[9] Google SparseHash. https://code.google.com/p/sparsehash/.

[10] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proc. 20th ISCA*, pages 289–300, 1993.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[12] Intel Performance Counter Monitor. www.intel.com/software/pcm.

[13] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proc. 45th MICRO*, pages 25–36, 2012.

[14] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[15] libcuckoo. https://github.com/efficient/libcuckoo.

[16] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th EuroSys*, pages 183–196, 2012.

[17] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-Copy Update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[18] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[19] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proc. USENIX ATC*, pages 11–11, 2011.

[20] TSX lock elision for glibc. https://github.com/andikleen/glibc.

[21] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proc. 21st PACT*, pages 127–136, 2012.

[22] R. M. Yoo, C. J. Hughes, K. Laiz, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *Proc. SC*, 2013.

A. Optimized TSX lock elision

Intel TSX provides two interfaces for transactional memory. The first is Hardware Lock Elision (HLE), a legacy compatible instruction set extension that allows easy conversion of lock-based programs to transactional programs. The second mode is Restricted Transactional Memory (RTM), a new instruction set interface with more complete transactional memory implementation. It provides three explicit instructions—XBEGIN, XEND, and XABORT—for programmers to start, commit, and abort a transactional execution, respectively. RTM is not backwards compatible, but it allows much finer control of the transactions than HLE. We focus on the use of RTM since it is more powerful and flexible than HLE and can serve as a upper bound of the performance improvements one may realize through TSX.

The released TSX RTM lock elision implementation for glibc [20] can be improved by specializing it for our hash tables. As a generic implementation, it is designed to work well for any mix of transactions, including the case of a mix of short transactions that must potentially coexist with long-running ones. In contrast, in the hash table workloads, all transactions are short. We further observed that the generic version misuses the EAX abort status code for RTM and takes the fallback lock too frequently. This causes performance to suffer because whenever a fallback lock is taken by one core,

```
void elided_lock_wrapper(lock) {
  xbegin_retry = 0; abort_retry = 0;
  while (xbegin_retry < _MAX_XBEGIN_RETRY) {
    // Start transaction
    if (status=_xbegin() == _XBEGIN_STARTED) {
      // Check lock and put into read-set
      if (lock is free)
        return; //Execute in transaction
      // Abort transaction as lock is busy
      _xabort (_ABORT_LOCK_BUSY);
    }
    // Transaction may not succeed on a retry
    if (!(status & _ABORT_RETRY)) {
      // There is no chance for a retry
      if (abort_retry >= _MAX_ABORT_RETRY)
        break;
      abort_retry ++ ;
    }
    xbegin_retry ++;
  }
  take fallback lock;
}

void elided_unlock_wrapper(lock) {
  if (lock is free)
    _xend();  // Commit transaction
  else
    unlock lock;
}
```

Figure 11: Optimized TSX lock elision

all the other cores have to abort their concurrent transactions.

We implemented our own TSX elision wrapper around existing lock functions. It is optimized for short transactions and elides the lock more aggressively. Figure 11 shows the implementation of our RTM elision wrapper, a modified version of the released glibc one [20]. It is a small library separated from glibc pthread, and thus does not require building a new glibc library. Its fallback lock can be of any type, including the custom spinlocks we use for cuckoo hashing.

Implementation details. `_xbegin()`, `_xabort()`, and `_end()` calls are wrappers around the special instructions that begin, abort, and commit the transaction. `_xbegin()` returns `_XBEGIN_STARTED` if the transaction begins successfully. `_ABORT_RETRY` is an EAX abort status code which indicates the transaction may succeed on a retry. We found that even if `_ABORT_RETRY` is not set in the EAX register, the transaction may succeed still on a retry. Whenever `_ABORT_RETRY` is not set, however, the glibc TSX lock elision aborts the transaction and takes the fallback lock immediately, forcing all other concurrent transactions to abort. Instead, we always retry several times before taking the fallback lock (using more retries if `_ABORT_RETRY` is set).

B. Cuckoo path overlap probability

Upper bound for the probability of a cuckoo path being invalid. Let N denote the number of entries in the hash table, L ($\ll N$) denote the maximum length of a cuckoo path, and T denote the number of concurrent writers. A cuckoo path has the highest possibility of overlapping with others when all the T paths are at their maximum length L. For a cuckoo path with length L, the probability that it does not overlap with another cuckoo path with length L is

$$P = \binom{N-L}{L} \Big/ \binom{N}{L} = \prod_{i=0}^{L-1} \frac{N-L-i}{N-i}. \tag{3}$$

The probability that the cuckoo path overlaps with at least one of other $(T-1)$ paths is

$$P_{invalid\_max} = 1 - P^{T-1} = 1 - \prod_{i=0}^{L-1} \left(\frac{N-L-i}{N-i}\right)^{(T-1)}. \tag{4}$$

Because $i \ll N$, we can assume $\frac{N-L-i}{N-i} \approx \frac{N-L}{N}$, so that

$$P_{invalid\_max} \approx 1 - \big((N-L)/N\big)^{L(T-1)}. \tag{5}$$

C. BFS cuckoo path length

Maximum length of cuckoo paths by breadth-first search. Let B denote the set-associatitivity of the hash table, M denote the maximum number of slots to be examined when looking for an empty slot before declaring the table is full, L_{BFS} denote the maximum length of the cuckoo path. The search process expands to two BFS tree rooted by the two alternative buckets of the key to be inserted. Each tree has at most $M/2$ slots. Therefore,

$$B + B^2 + B^3 + \cdots + B^{L_{BFS}} \geq M/2, \tag{6}$$

which gives us

$$L_{\mathrm{BFS}} = \left\lceil \log_B \left(M/2 - M/(2B) + 1\right) \right\rceil. \tag{7}$$