# Practical Hash Tables for Parallel Programming

John Viega

john@viega.org

# Introduction

While the performance of single processors has begun leveling off across the industry, multi-core systems have exploded. Meaning, if the average developer can better leverage parallelism, she’ll be much better equipped to take advantage of the underlying hardware platform.

Unfortunately, parallel programming is notoriously challenging to get right, particularly when it comes to synchronizing data and communication between threads. So much so, that many high-level programming languages, while supporting multi-threading, implement a “Global Interpreter Lock”, ensuring only one thread runs at a time – it is incredibly difficult to provide a robust high-level language while supporting threads, particularly without dramatically harming performance of single-threaded programs. Since few developers are good at multi-threaded programming, it’s often the right tradeoff.

Thread-safe high-level data structures are one tool that can help reduce the burden on programmers tremendously, allowing threads to communicate through shared data structures. Other languages do provide such primitives, but they are tough to make implement effectively, in a way that scales. The most straightforward approach is to fully synchronize access to individual instances of the data structure, but even that can be hard for the average programmer to get right and suffers from the benefit not being possible to scale up, as the number of cores scales up.

For instance, I recently saw a thread from late 2019 on StackOverflow about this problem[[1]](#footnote-1). In the answers and discussion, it’s clear that people have significant misconceptions here, including:

1. “The only way to implement a hash map with true concurrency is to have an immutable hash map.” – This is false, but it’s clear that some people believe it.
2. That it is impossible to safely shrink true concurrent hash tables, only grow them (we will show that is not the case).
3. That it’s probably not possible to get ordering guarantees (again, it is possible).

The thread in question reflects my review of the state of the art: many practically implemented algorithms that scale to multiple processors well, are not true hash tables, in the sense that they do not have amortized constant time (O(1)) cost for some of their operations (usually all their operations).

I began to take an interest in this problem in October of 2021, when looking for a concurrent hash table for another project. It was clear that most existing algorithms either are not true hash tables, or rely on locking, which is problematic in terms of trying to take advantage of systems with large numbers of cores.

In fact, the only practical concurrent, lock-free hash table algorithm I could find in the real world is a lock-free hash table written by Cliff Click, presented at JavaOne[[2]](#footnote-2) all the way back in 2007 (an open-source implementation exists[[3]](#footnote-3)).

I considered porting the table to C for my project, but ran into several issues that I found unsatisfying:

1. Being written in Java, the algorithm assumed garbage collection. Performantly managing safe deletion in a concurrent data structure needs to be addressed for such a port.
2. The algorithm does not allow for shrinking the table, after many deletions.
3. When getting an iterator on a hash table, it’s possible to get an inconsistent view of the table (for instance, an iterator may include multiple instances of the same key).
4. The algorithm for expanding the hash table is, to my mind, vastly overcomplicated.

This document will show how to solve many problems for making parallelizable hash tables more practical and efficient. We will present several new algorithms, all implemented (in C) in the Hatrack library (github.com/viega/hatrack). Many of the problems we address have not had previous practical solutions.

We start out our journey in concurrent hash tables with a simple table that uses locks, then keep building to solve numerous problems. For instance:

1. Our first locking hash table (swimcap) implements a single-writer hash table, where readers can progress while writes are in progress, without having to use a full read-write lock (which would suspend readers when the table is being updated).
2. Our second locking hash table (swimcap2) shows how to implement a single-writer hash table, where readers are guaranteed wait-free, meaning that read operations will always complete in finite, bounded time, even if writer threads get stuck.
3. Our third locking hash table (newshat) shows how to allow multiple writers, except while changing table sizes.
4. Our first lock-free hash table, hihat1, shows how to build a lock-free hash table (meaning that, individual threads can in theory be starved or move slowly, but overall, the whole system progresses; an impossible guarantee in a system with locks). Unlike the Clack hash table, we provide a wait-free “get” operation, as well as a much simpler (and wait-free) technique for changing table sizes, with the ability to make them either larger or smaller. Our modification operations are also wait-free when there is no resize operation, and lock-free when there is. Additionally, we show how to efficiently manage record deletion.
5. Our second lock-free hash table, hihat64, shows how to build a hash table with all the benefits of #4, with only a single (pointer-sized, generally 64 bits) “compare and swap” (CAS) operation. The previous table uses a double-word (generally 128 bit) CAS.
6. With lohat0, we build a lock-free hash table, where all operations are fully ordered. This means that, unlike other highly concurrent tables, we can present the list in order (as is the default with iterating over Python dictionaries, for example). This is possible, in part, by significantly extending previous research on epoch-based schemes for memory management (that we introduce with hihat1).
7. With lohat1 and lohat2, we explore time/space tradeoffs, where (with lowhat1), we can still get O(1) time insertions, lookups and removals, and sub- O(n log n) sorts, and (with lowhat2), we can get sorts that approach O(n), if we are willing to accept lookup operations potentially growing based on how many times a particular key has been deleted since the last resize.
8. With witchhat and woolhat, we show how to make our hash tables truly wait-free (meaning no individual threads can be starved), with de minimis performance impact (traditionally, lock-free operations are more efficient than corresponding wait free operations).
9. With tophat, we show how to tie everything together, maximizing both single-threaded performance and concurrent performance, in a single construct fit for programming languages that are looking to optimize for performance in both kinds of environments, in a way that is transparent to the end user.
10. From the above work, we build a set abstraction that shows how to implement fully consistent and efficient operations for difference, intersection, union and so on.

Each of the above algorithms are free of intellectual property claims, with code available under an Apache license, including our test bench.

We use our test bench to show that our algorithms perform very well under a wide variety of extreme conditions, including very high write contention across many threads.

# Preliminaries

Hash tables are associative data structures, able to map keys to values. Generally, hash functions allow for amortized (average-case) O(1) lookups, insertions and deletions, with the worst case for a single operation being O(n).

Generally, the “cost” one pays with a hash table is memory. Hash tables use O(n) memory, but they generally intentionally keep a surplus of empty space, and resize when the table gets too crowded. As we will see here, in a parallel environment, the migration can be done without impacting lookup time. In practice, the impact to write operations amortizes out as well, and can be fully bounded, to provide wait freedom.

A core concept behind a hash table is to use a function of the key, called a hash function, to map keys to a fixed size *hash value*, that is then used to index into a fixed-size array, to a location commonly called a *bucket*. In most practical applications, real collisions will occur, at least in the sense of multiple keys mapping to a single bucket (there are approaches to ensuring that keys are statistically incredibly unlikely to have colliding hash values). Therefore, a collision resolution approach is necessary.

One obvious approach to hash bucket collisions is chaining, where items that hash to the bucket are simply kept in a linked list. For various reasons (mainly performance based), chaining is rarely used in practice. It’s far more common to use some form of *open addressing*, which, confusingly, is also known as *closed hashing*. The basic idea is that, when there is a collision, we apply a second function, the *rehash function*, which, given one bucket selects a second bucket. On insertion, if we find a spot occupied after applying the main hash function, we apply the rehash function iteratively, until we find an open spot (care has to be taken not to let the hash table fill up, and to use a rehash function that ensures every possible bucket will eventually be selected). Lookup operations, when finding the “wrong” item in the bucket selected by the main hash function, iteratively apply the rehash function until either the find the correct item, or find an empty bucket (meaning, the item is not in the hash table). Deletion can be performed by moving data around in the table upon deletion, or simply marking records as deleted.

Much work has been done on approaches to hashing and open addressing. Our work is mostly agnostic to it all. Here are the core considerations:

1. In most cases, our implementations assume a 128-bit hash function, with a uniform distribution of hash values. Generally, universal hash functions or cryptographic hash functions are strong choices. Our implementations actually use XXH-128, which is non-cryptographic, but very fast on a wide variety of architectures, and does very well on statistical randomness tests. Also, to remove hashing choice from performance benchmarks, we have all implementations use the same algorithm, and pre-compute all hash values, so that the hash operation during testing is simply a lookup into a fixed-sized array.
2. All of our implementations use linear probing for the rehash function, meaning that, starting from the bucket indicated by the initial hash, the reprobe operation is simply: reprobe(original\_hash\_value, table\_size): (original\_hash\_value + 1) % table\_size. This is a common choice, because it’s been widely demonstrated to be the best general purpose approach in the face of modern cache architectures.
3. As with many implementations, our table sizes (measured by number of buckets) are limited to powers of two, so that the (generally slow) modulus operation in the linear probing can be implemented by a fast logical AND operation.
4. All of our algorithms handle deletion by marking the bucket deleted. Most of our algorithms allow reuse of the bucket, if a key is reinserted. But none of them ever allow replacing one bucket with another. We will discuss this decision more when appropriate, but the decision allows for simple, efficient algorithms in the case of multiple writers. Without this decision, there are obvious race conditions, that would require much more complicated (and I expect generally less efficient) algorithms.

The core advantage to using the 128-bit hash function, is that, assuming a reasonable hash function choice, it’s so statistically unlikely to have hash collisions, that the hash value itself can, in practice, stand in for object identity, saving the cost associated with checking the values of keys (which can be significant, for instance, in the case where strings are used for keys, and two different strings have a reasonable chance of having the same hash value).

Hatrack currently is built for (and tested on) both Macs and Linux machines, with 64-bit word sizes. Whenever it makes sense, our integer data types are declared with an explicit size, with a strong preference to 64-bits. All of our hash table implementations have a consistent API:

**void algorithm\_init(algorithm\_t \*self)**

The input is a pointer to memory of the appropriate size to hold the hash table; this function must be called and completed by a single thread, before other operations may begin. Note that, given a single agreed upon address to hold the pointer, it’s straightforward to allow threads to race to initialize the structure (we will see the approach when changing table sizes). However, this complexity is generally needless in practice, so we avoid it.

**void algorithm\_delete(algorithm\_t \*self)**

Similarly, this operation assumes that you call it from a single thread, and that you’ve guaranteed that all operations on the table have completed, and that no more are going to start. Typically, in C, this would be implemented by reference-counting the table, which is a well understood problem, that we currently leave to the user.

**void \*algorithm\_get(algorithm\_t \*self, hatrack\_hash\_t \*hash\_value, bool \*found)**

The data type hatrack\_hash\_t is an array of two 64-bit integers, representing the application of the selected hash function to the key. Note that the key is NOT passed as an input to this API, since the hash value is sufficient to test for identity tests in all cases. The return value is an arbitrary 64-bit value. Generally, we expect this to be a pointer to memory that holds the key/value pair. Or, an implementation could easily double the space allocated to the item in each bucket, to store actual values for both keys and items. We took this approach so that one algorithm can be used both for dictionaries and sets (where we only care about keys, not associated values). The final field is a pointer to memory, into which the algorithm will store a boolean value, indicating whether the item was in the table or not, so far as the operation was concerned. This is used to distinguish between an item of zero value being returned from a NULL pointer being returned. For instance, Hatrack’s testing harness passes in 32-bit integer keys/values in the item field, instead of using pointers, so requires this distinction. But, when storing a pointer to a key/value pair, the field may not be necessary. If set to NULL, the value will be ignored.

**void \*algorithm\_put(algorithm\_t \*self, hatrack\_hash\_t \*hash\_value, void \*item, bool ifEmpty, bool \*found)**

This function really combines two separate notions of put into a single function, but, in Hatrack, is always implemented with two functions under the hood. One function will only insert if the item wouldn’t be overwriting a previous insertion of the associated item (putIfEmpty), and other will overwrite, if present. This call combines the functionality of both.

If ifEmpty is set to true, then the algorithm will return either true or false, depending on whether the insert was successful, or not. This version is meant to replace single-threaded code of the form (in Python) that has an obvious race condition in multi-threaded scenarios:

if x not in dictionary:

dictionary.insert(x, y)

Note that, when ifEmpty is true, the found parameter will be ignored, and the return value will be true when the insert was successful, and false when there was already something in the table.

When ifEmpty is false, the found parameter, if provided, will have the memory that it points to set to either true or false, depending on whether the item was already in the table. The return value in this version will be a NULL pointer when the item was NOT in the table, and will be the previous value of the item if there WAS a previous item in the table.

Note that, we return the old item on overwrite to make sure that the caller has the opportunity to do whatever memory management is necessary, with regard to the items. Different applications will have different requirements; this API allows us to be agnostic to those requirements, as we never do more than store a 64-bit pointer (that can alternately be used as a 64-bit value, which still would not require memory management). The Hatrack hash tables do give serious consideration to memory management, but only for the records used internally in the data structure.

**void \*algorithm\_remove(algorithm\_t \*self, hatrack\_hash\_t\*hash\_value, bool \*found)**

This returns the previous value of the item, if present. If found is non-null, this function will store, in the memory address provided, a true if there was a previous item in the array, in which case the return value will contain that item (again, for the sake of memory management). Otherwise, when there was no item found to remove, the address pointed to by found will get the value false, and the function will return NULL.

**uint64\_t algorithm\_len(algorithm\_t \*self)**

In a multi-threaded environment, this function is always going to be subject to a race condition; by the time the thread gets the result, a parallel operation could have added or removed items from the structure. We provide this function more because it’s expected, but it should only ever be considered an approximation of the current state, and not a reliable value. If you want a more reliable value, use algorithm\_view().

**hatrack\_view\_t \*algorithm\_view(algorithm\_t \*self, uint64\_t \*num\_items)**

This returns a view of all the items in the dictionary, as an array of hatrack\_view\_t objects. The parameter num\_items must be provided, and the memory address to which it points will receive the number of items in the returned array.

In our “ordered” hash tables (which we will from now on refer to as “fully ordered” hash tables), the results will be sorted, in the order by which the keys were committed to the table (as identified uniquely by the hash value). If a key was deleted and then reinserted, the ordering will be based on the most recent insertion. If an insertion overwrites a previous insertion of the same key, the ordering time will not change (mimicking the behavior of Python dictionaries). In other hash tables, we will still sort the results based roughly on insertion time, but generally by the time in which the record was created, which could be different from the true insertion time.

Note that, in versions that are not considered fully ordered, there is the possibility of the view being inconsistent. For instance, the same key may appear multiple times in the array.

## Refhat: a reference, single thread only hash table

# Swimcap: Minimal locking readers

# Swimcap2: Wait-free readers

# Newshat: Multiple readers, multiple writers

# Hihat1: Lock-free on resize, wait-free otherwise

# Hihat64: Extending Hihat1 to support single-word CAS

# Order-preserving hash tables

Ballcap: Order-preserving, with locks

Lohat0: Order-preserving, lock-free

# Exploring time/memory tradeoffs with Lohat1 and Lohat2

# Witchhat: A fully wait free hash table.

# Woolhat: A wait free and fully order-preserving hash table

# Tophat: A simple solution for general-purpose programming languages

# Efficiently implementing sets

# Performance Analysis

# Related Work

# Conclusions

1. https://softwareengineering.stackexchange.com/questions/401503/implementing-a-hash-table-with-true-concurrency [↑](#footnote-ref-1)
2. https://docs.huihoo.com/javaone/2007/java-se/TS-2862.pdf [↑](#footnote-ref-2)
3. https://github.com/boundary/high-scale-lib/blob/master/src/main/java/org/cliffc/high\_scale\_lib/NonBlockingHashMap.java [↑](#footnote-ref-3)