# Practical Hash Tables for Parallel Programming

John Viega

john@viega.org

Hatrack provides both a high-level interface to dictionaries and sets, and implementations of numerous hash tables for the sake of comparison and instruction.

This document is meant to be a detailed deep-dive on the techniques in a logical, instructive order, starting with basic locking hash tables, and building up to the “final” hash tables we use in our higher-level dictionary and set implementations.

I will do a separate (much shorter) overview for people that are more interested in understanding the basics without the detail, and just applying the hash tables.

This document is not nearly complete. If you’re actually reading this:

1. Please be patient with me; it’s going to take me a long time to finish this, probably much longer than it took to write the code, especially if it effectively turns into a dissertation.
2. Please communicate with me—I’m incredibly interested in feedback and suggestions.

# 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 many 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 practical (or possible) to get strong ordering guarantees like wait freedom (again, this is not true).

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. And I couldn’t find anything that addressed the practical memory management concerns for C, where memory management is generally left to the programmer.

In fact, the only practical concurrent, lock-free hash tables I could find in the real world seemed to be based on a lock-free hash table written by Cliff Click, presented at JavaOne[[2]](#footnote-2) all the way back in 2007 (multiple open-source implementations exist[[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 (duncecap) 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 (swimcap) 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, hihat, 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, oldhat, 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. While the previous table can be implemented with only a single-word CAS, it loses some collision resistance if you do.
6. With lohat, 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 hihat).
7. With lohat-a and lohat-b, we explore time/space tradeoffs, where (with lowhat-a), we can still get O(1) time insertions, lookups and removals, and sub- O(n log n) sorts, and (with lowhat-b), 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 crown, we explore what happens to hihat if we try to cache linear probe information in an attempt to speed up bucket lookups.
10. With tiara, we implement hihat without sorting information, and with only 64-bit hashes, but allowing our core access algorithms to work with only a single 128-bit CAS operation!
11. 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.
12. From the above work, we build a high-level dictionary interface, that can help the programmer deal with practical issues, including memory management in the face of extreme parallelism.
13. We additionally 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.

## Basic Hashing Approach

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.

Certainly, if there are biases in the hash function, that can lead to more probing than would be expected. We discuss hash function selection below.

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. That is, if table\_size is a power of two, a bucket location can be found from a hash value by calculating: hash\_value & (table\_size – 1). Again, this is much more efficient.

Since we will perform this operation often, will not frequently need the full table size, and can easily calculate the full table size, our implementations will all store table\_size – 1, calling it last\_slot.

Let’s look at a simple example of hash table operations; it will motivate another basic consideration se need to worry about—how to cope with deleting from the table. Yes, this should be remedial for anyone with an undergraduate degree in CS, but we won’t dwell long on it.

Below is a hash table with four buckets, that already contains two items. We’ll consider the table to be of fixed-size (generally, hash tables expand as necessary, and we will discuss that below).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bucket Index | 0 | 1 | 2 | 3 |
| Hash Value | 0x…04 | 0 (empty) | 0 (empty) | 0x…03 |
| Contents | “item m” | <empty> | <empty> | “item n” |

Let’s look at what happens when we add a new item, “item x”, under the assumption that its hash function (for the sake of example, hex ending in 0x07) points us to the bucket with index 3. We see there’s a hash value associated with it, so we move on to the next bucket, where we see there’s an item associated with it. Since we’re using linear probing, we do so by taking the hash value, adding one to it, and then modding by the table size. That tells us to look at bucket 0. But that’s full too, so we add one more, getting bucket #1, where we can place our item:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bucket Index | 0 | 1 | 2 | 3 |
| Hash Value | 0x…04 | 0x…07 | 0 (empty) | 0x…03 |
| Contents | “item m” | “item x” | <empty> | “item n” |

When we want to retrieve the contents associated with the hash value 0x…07, we go through a similar process:

1. We first compute the hash value, modulo the table size, which points us to bucket #3.
2. We compare the hash values, and see that they’re different, so we “probe”, by adding one, modulo the table size.
3. We now go look in bucket 0. The hash values again do not match, so we probe again.
4. Now we look in bucket 1, see that the hash value is the right hash value, and we then return “item x”.

Crucially, when we’re looking up items in a hash table, we have to know when to stop probing, because the item is not going to be in a hash table. That’s done by stopping whenever see an empty bucket. Let’s consider what happens when, in the above table, we go to look up an item having the hash key ending in 0x01:

1. We’ll first look in bucket 1, which belongs to a different item.
2. We then probe to bucket 2. But, since this bucket is empty, we conclude that there is no entry in the table for the item in question.

## Handling Deletion

Having empty buckets indicate that a key is not present makes deletion a bit more problematic. Let’s consider what happens when we delete “item m”. Let’s say we take the naïve approach, and just empty out the bucket. This becomes problematic when we try to look up the item with the hash key ending in 0x07:

1. We look at bucket 3, which is full.
2. We then look at bucket 0, which is empty, causing us to decide that the item we’re looking for isn’t in the table – even though it is!

The most common approach to this issue is to “move” buckets on deletion, if appropriate. A basic algorithm here is:

1. When we delete a bucket, start scanning to the right (modulo the table size).
2. If the current bucket we’re scanning is empty, we are done.
3. If the item in the current bucket’s original hash value, modulo the table size, is less than or equal to the index of the empty bucket to it’s left, move it, then continue scanning the next bucket (goto step 2).
4. Otherwise, when the current bucket wouldn’t make sense to move, we are done.

This strategy is fine for single-threaded applications, but is a bit more challenging if we want to support full parallelism, where we might have both multiple simultaneous readers and multiple simultaneous writers.

If the deletion thread might run in parallel to reads, what happens if we still try to do our copying strategy? Let’s first assume we can atomically update the entire contents of a bucket—the hash value PLUS the contents. Because, if we can’t, then the bucket can be in an inconsistent state, yielding corrupt reads.

If we can do atomic updates, we still have a problem. Let’s say that the delete operation “moves” by first copying from a source bucket to the destination to its left, then erasing the source bucket. Consider the following order of events on the above table:

1. Thread A starts a delete operation on the hash key 0x04. It rightly decides it needs to move the bucket at index 1 (with the hash key 0x07) to the left. It intends to do that by first writing over bucket 0 with the contents of bucket 1, then emptying bucket 1.
2. In parallel, Thread B starts a lookup operation for the hash key 0x07. It starts at bucket 3, and sees it has to probe.
3. Thread B loads bucket 1, BEFORE Thread A overwrites it, determining that it needs to probe to bucket 2.
4. Thread B then gets suspended by the scheduler.
5. In the meantime, Thread A overwrites bucket 1, and erases bucket 2, completing the deletion operation.
6. Thread B wakes back up, and looks in bucket 2, seeing that it’s empty.

In this example, Thread B concludes that the item with hash key 0x07 is NOT in the table, even though it clearly is in the table.

To address this problem, we introduce an invariant that we borrow from the Cliff Click hash table:

Once a hash key is associated with a bucket, no other hash key can ever be associated with that bucket.

So, in our tables, when the item with hash key 0x04 is deleted, the table ends up looking like this, conceptually:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bucket Index | 0 | 1 | 2 | 3 |
| Hash Value | 0x…04 | 0x…07 | 0 (empty) | 0x…03 |
| Contents | <deleted> | “item x” | <empty> | “item n” |

With our two contending threads, Thread B will keep probing until it either finds the key it’s looking for, or finds an empty bucket. So it will get the right result.

Note that, if we re-insert an item associated with the hash-value 0x04, we will end up putting back in the same bucket.

This approach GREATLY simplifies our job for parallel hash tables. It does have the disadvantage that deletions cause our tables to fill up faster, leading us to the question of what we do when the table gets full.

## Resizing or migrating table stores

With other hash tables, when they get full, people will refer to a “resize” operation, where we grow the table, typically by doubling the size. Instead of waiting until the hash table is completely full, there’s generally a metric for determining when it’s “full enough”. We’ve seen metrics such as measuring the length of individual probes, resizing when it gets “too long” by some definition. But, we’ve more commonly seen a fixed percentage—specifically, when the table is 75% full, we resize. This is conceptually easier, and has lower overhead to manage.

For our tables, we use a similar metric. If 75% or more of the buckets are occupied (generally measured by whether they have an associated hash value), we will want a new set of buckets. We will copy entries into this new set of buckets, *but not entries corresponding to a deletion*.

If our table had a lot of deletions, we might have few records to copy over. It might make sense to keep the table the same size, or even SHRINK the table. We therefore will usually refer more generically to “table migration”, which may or may not include resizing the table. Note that we call sets of buckets (and any necessary metadata) *stores*. So, we will talk about table migrations from an old store to a new store.

We have not yet done any investigation into tweaking our metrics around migrations. We use the following metrics in most cases:

1. If, when we insert a new item, we find that 75% of the buckets are occupied, we trigger a migration.
2. If, when factoring out deletions, the CURRENT store is at least 50% full, we DOUBLE the number of buckets in the new store.
3. If, when factoring out deletions, the CURRENT store is up to 25% full, we HALVE the number of buckets in the new store.
4. Otherwise, we keep the number of buckets the same, only clearing out deletions.

When we migrate stores, we conceptually have new buckets. So, while re-insertions will end up in a new bucket, they do not violate our invariant above—each bucket can only *ever* be occupied by at most one hash value.[[4]](#footnote-4)

Note that the above thresholds seem to perform well enough in practice, that we have not yet investigated changing them. They are convenient thresholds in that, when we do need to compute them, they are extraordinarily cheap to compute:

1. 50% of the table size is a simple shift right by 1.
2. 25% is a simple shift right by 2.
3. 75% is the table size minus 25% of the table size.

## Hash Function Selection

Every “real world” hash table we’ve seen has used a hash function that produces 64-bit outputs.

A 64-bit output is generally considered to be enough that the “zero” value is so absolutely unlikely, that it can stand in to mean “not occupied”. However, that includes a pretty big assumption that, for an arbitrary input, all possible hash values are equally likely, without any bias.

There are a few easy ways to address this problem:

1. Use a well-regarded cryptographic hash function, such as SHA256. While there’s not a forthcoming proof that there’s no bias in any such function, it passes common statistical tests, and would be well regarded for this use case.
2. Use a universal hash function, where such proofs are available. Universal hash functions do require selecting a “key”, which can be done when initializing a table.
3. Use a non-cryptographic hash function that nevertheless passes common statistical tests.

The common objection to using cryptographic hash functions is performance. However, that’s generally a bogus objection:

1. Most modern architectures now have instructions specifically for accelerating the most widely used cryptographic hash functions (including SHA256).
2. Even without such acceleration, such algorithms generally would perform more than acceptably for the vast, vast majority of applications.
3. Hash values can often be *cached*, so that they’re only computed a single time per data item.

Universal hash functions are a solid alternative that tend to be less complex and costly, plus have good provable collision resistance properties. Plus, they also can often be accelerated via hardware. For instance, the AES-GCM encryption mode leverages a universal hash function called GHASH, that is often either fully implemented in hardware, or has instructions accelerating it (PCLMULQDQ on the x86).

Still, there’s sight additional burden in managing keys, making simple, fast APIs to such algorithms uncommon.

In hatrack, we currently give a nod to the generally accepted practice of using fast non-cryptographic functions, that nonetheless do well on statistical tests. Specifically, we use the XXH-128 algorithm (though we expect to eventually add compile-time options here).

One key decision we made in hatrack is to use 128-bit hash values. In all cases. Yes, 64 bit hash values would be sufficient for letting a “0” hash value stand in for an empty bucket (we’d expect a hash value to be zero about .0000000000000000000001% of the time, requiring on average 2^63 items to generate such a value).

Unfortunately, in practice, 64 bits is not sufficient to avoid two DIFFERENT items yielding the same hash value. Since we don’t care about any “specific” hash value, this is an instance the well known “birthday” problem. It means that, instead of the expected number of operations before an issue arises isn’t on the order of 2^64, it’s on the order of 2^32. That’s a small enough number, that we can expect to see real collisions in real-world applications.

Some hash tables using 64-bit hash values (or smaller), ignore this problem, but that is foolhardy. More common is to not just check the hash value, but to check the contents of the actual key for equality. That’s a reasonable approach, though it requires extra logic, and extra operations.

Instead, we can use a 128-bit hash value as an identity, which DOES give us O(2^64) operations before any collision, which is, again, far beyond anything worth worrying about—as long as the hash function doesn’t have a significant bias, of course.

One might expect a 128-bit hash value would be less efficient to calculate than a 64-bit hash value. That’s becoming less and less true on modern architectures. For instance, in Hatrack, we selected XXH-128 as our default algorithm, which gives a 128-bit output. It is almost as fast as XXH3, which gives a 64-bit output (for instance, with throughput of 29.6GB/s vs. 31.5GB/s, in the x86 benchmark on their web site). That’s ever so slightly faster than sequentially reading memory out of ram, and nearly 30 times faster than a typical cryptographic hash (Blake2 gets 1.1GB/s in the same benchmark). It’s much faster than most other non-cryptographic hash functions as well, while doing just as well on statistical tests as cryptographic functions (again, like Blake2).

Requiring 128 bit hash values (with good properties, of course) means that we can use the hash value as a stand-in for identity testing, and we can do that without sacrificing hash function performance.

However, especially when we move to lock-free and wait-free algorithms, we will need to be able to update hash values atomically. Thankfully, most modern architectures have instructions allowing us to operate on 128-bit values (which is largely why XXH-128 can be so fast).

This includes the “compare-and-swap” operation at the heart of most lock-free algorithms. For instance, the x86 has had the CMPXCHNG16B instruction to swap 128-bit values for quite a while, with the only requirement that values being swapped are aligned to a 16-byte (128-bit) memory address.

For our lock-free and wait-free algorithms, we ONLY use a 128-bit compare-and-swap operation on the hash value. Such operations can always be replaced 64-bit compare-and-swap operation, if you are willing to accept 64-bit hashes in the identity test (or take a more complicated approach to identity testing).

Not all of our compare-and-swap operations are of hash values. But, while our algorithms often benefit from a 128-bit CAS, they can always be adapted to avoid it. In fact, we can even support 128-bit hash values with only a 64-bit CAS operation—we do this with the oldhat hash table.

## Other Considerations

When Python changed their underlying dictionary implementation (quite a while ago now), anecdotally people raved about the fact that the dictionaries preserve insertion order. That is, when you ask Python to iterate over the contents (e.g., when printing), it will, by default, give them to you in the order they were inserted.

If a value is overwritten without being removed, Python orders based on the first write time. However, if an item is removed then reinserted, Python will order based on when it was reinserted.

Our tables all keep information on insertion order, though not always with the same degree of precision as with Python (which leverages the global interpreter lock, making dictionary access effectively single threaded).

As a final consideration, NONE of our algorithms currently allow the user to pre-select a store size at allocation time. Cliff Click rightly pointed out in the source code for his algorithm that the user’s best guess is often disastrously wrong (for instance, using far more memory than necessary), and that the cost of sizing up quickly is close to irrelevant.

We may consider adding this option, primarily because comparative benchmarks often make use of it, but we do think it’s better for the user to not have an option to consider that detail.

# Low-Level Operations and API overview

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.

While we have a high-level interface for Dictionaries and Sets that we discuss later, we have implemented a large number of hash tables that all have a consistent API:

**algorithm\_t \*algorithm\_new()**

This always allocates a new hash table of the specific type with the default allocator. Then, it calls the appropriate initialization function (see next), and returns the value.

**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\_cleanup(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.

This operation cleans up any internal state the table has, but does not free the memory at the input pointer address (use the delete operation, next, for that). This allows dictionaries to be stack allocated, or allocated using a second allocator, beyond the system allocator.

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

This operation always calls the cleans up operation, above, and then calls the system free() function on the input pointer.

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

The data type hatrack\_hash\_t is conceptually a 128 bit (signed) integer, 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 \*found)**

This function inserts an entry into the hash table, replacing the existing entry, if it already exists. If an entry previously existed, it is returned, otherwise NULL is returned. Since NULL can be a valid entry, you may also pass a memory address to the final parameter to get information on whether a value was returned.

Note that the convention of returning data we are ejecting from the table is meant to aid with memory management; at the time of return, we guarantee that no other threads are able to access the contents any more – WHILE PERFORMING TABLE OPERATIONS. It does not ensure that other threads have no copy of the contents, nor does it ensure that other threads won’t free it out from under us.

We cannot stress enough that the guarantee only applies to other table operations. When dynamic memory is used for items put into a table, there does need to be another layer or memory protection. We address that problem in our high-level API.

However, we do not address that problem in our low-level API; it gives us enough to tell us that it’s ejected an item, and that’s it. And, our test suite built on top of these low-level APIs uses only integers for keys and values, which are copied in, requiring no memory allocation.

**bool algorithm\_add(algorithm\_t \*self, hatrack\_hash\_t \*hash\_value, void \*item)**

This function inserts an entry into the hash table, but only if it does NOT exist. It returns true if the operation is successful, and false if the item was NOT inserted.

This function is here to replace single-threaded logic of the form:

if not table.contains(key): table.put(key, value)

That logic does not work in a multi-threaded environment, as it has an obvious race condition—even if the return from table.contains() is false, the return value could change by the time our put operation runs.

**void \*algorithm\_replace(algorithm\_t \*self, hatrack\_hash\_t \*hash\_value,**

**void \*item, bool \*found)**

This function inserts an entry into the hash table, but ONLY IF it replaces an existing entry. The return value works the same way it does with the put operation.

This function is here to replace single-threaded logic of the form:

If table.contains(key): table.put(key, value)

Which suffers from the same race condition as we noted in the add operation above.

**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, for some odd reason, 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.

# Progress Guarantees

Key to this work is an exploration of *progress guarantees* for hash table operations. Progress guarantees relate to the conditions under which threads can make forward progress, in the face of parallelism. For instance, the weakest form of progress guarantee for an operation is *deadlock freedom*, which guarantees that, for a lock, some thread attempting to acquire the lock will eventually succeed. That implies whole-system progress, but does not imply that every thread will make progress.

Deadlock freedom of course requires that threads holding the lock will eventually release it, which requires a cooperative scheduler, and requires the programmer to use the lock correctly. Note that this guarantee does not apply to EVERY thread. For instance, if, once a mutex is unlocked, our underlying mutex implementation always selects a new thread to acquire the lock from those waiting at random, it’s possible (though obviously extremely unlikely), for individual threads to be starved—waiting on the lock forever, because other threads keep acquiring it.

Starvation freedom is a stronger progress condition, with a guarantee that every attempt to acquire a lock will eventually succeed. Of course, the algorithm needs to be correct, the underlying lock implementation must cooperate (for instance, by giving out locks in the order in which they are requested), and the scheduler must guarantee that it will eventually schedule all threads.

An underlying issue with locks, though, is that the scheduler can suspend threads that are holding locks, leading to an unnecessary delay in progress. So called *non-blocking progress guarantees* are generally considered more desirable, in that a long delay from any one thread does not prevent other threads from making parallel progress.

The weakest non-blocking guarantee is obstruction freedom, which guarantees that any single thread will progress -- if all other threads stop.

Generally more desirable is lock freedom, which guarantees that some thread will eventually complete an operation, whether or not other threads stop.

Lock freedom does not ensure all threads make forward progress, however. Consider, for instance, an operation that updates a value on a fixed-size array, using a compare-and-swap (CAS) operation, as with the following pseudo-code:

function array\_set(arr, ix, desired\_value):

expected\_value = arr[ix].value

while CAS(arr[ix], expected\_value, desired\_value): pass

The CAS operation typically updates the memory location associated with the first operand, if and only if the value at the time of the operation is equal to the value associated with the second argument. If it is, the new value is taken from the third argument, and the CAS operation returns True. If the value is not as expected, then not only does the CAS operation return False, but the value associated with the second parameter gets updated with the value of arr[ix] at the time of the operation. The CAS operation is atomic.

The above array\_set() operation contains no locks, and is lock free. However, there is no strict guarantee for any particular thread’s progress. To illustrate, let’s imagine we have 1000 threads all trying to update a particular array element as fast as possible, for as long as possible. One thread might find that, with every single CAS operation it makes, the operation fails, because the expected value changes due to one of the other 999 threads.

Still, this algorithm is lock-free. Even though we can use the CAS operation to implement a spin-lock, the above algorithm does use CAS in a way that would constitute a lock, in that no thread ends up with exclusive access to any resource in a way where that thread can be suspended (the only exclusive access is that guaranteed by the underlying atomicity of the read and CAS operations).

In a version of this algorithm using locks, if a thread that were holding the lock was suspended, other threads trying to update would have to wait for that thread to wake up and release the lock. In contrast, in this algorithm, the only way one threads’ progress is impeded, is if some other thread made forward progress instead.

In short, the major difference here is that, while individual threads may starve, it would only be because other threads are making forward progress.

Certainly, in lock-free algorithms, it tends to be the case that starvation is not practical. Still, when there are situations with lots of contention, individual threads could see a delay. In practice, such delays may not matter much. For instance, early in the process of building Hatrack, I measured the number of times a put operation recursively called itself, due it participating in a migration, where a migration was again necessary by the time the put operation resumed. In that testing, the table was usually (but not always) growing during migrations, and so it was no surprise to see the number of retries in a test follow a logarithmic curve. In my testing, I never saw it require more than five retries.

However, on the same machine (an 8 core Macbook M1), I wrote a test program that fires up eight threads, all of which attempt to CAS a unique value to a global variable as many times as possible. There, the number of retries, by sight, roughly seemed to follow a log-normal distribution. In a test run, about 90% of attempts took fewer than 10 fails, and 99% took fewer than 71 fails, yet there were extreme outliers, with three attempts taking over 5,000 fails, but less than 6,000 (See Appendix A for the test program, and an example run). That’s clearly a long, logarithmic tail. I even did one where a single attempt failed 16,150 times (that run saw 4 attempts with more than 5,000 failures).

Still, even a few-thousand retries for an operation that takes just a few cycles probably won’t matter for most applications. An example test run on my laptop took 3.25 seconds to complete 100M operations, plus the I/O to report results. That’s certainly over 30M successful operations per second.

Nonetheless, we can imagine applications with real-time needs, or where semantically the current contents of a data structure may not make sense if a single operation is delayed so much relative to most successful operations.

For such applications, we might want something stronger than lock freedom: wait freedom.

With wait freedom, all threads are guaranteed to make progress in finite time, meaning we must be able to find a finite bound to the number of operations a thread will take to complete. There are generalized approaches to converting lock-free algorithms into wait-free algorithms, most notably “helping”—after a fixed number of retries, a thread signals for help, and all other threads attempting the same operation first ensure that threads needing help complete their operation.

Usually, such helping mechanisms are heavyweight, and meant to be avoided except in the long, long tail.

Hatrack began as a project to build lock-free hash tables, with the goal of making them wait-free, if the performance was roughly equal to lock-free in practice.

But, considering the fact that OS schedulers are generally fair, it stood to reason that the impact of any algorithmic changes necessary to achieve lock freedom might be greater in practice than the impact of the scheduler suspending threads that hold locks.

To that end, we also built efficient hash tables using locks, both for comparative testing, and for educational purposes. In general, we’ve found that our lock-free hash tables do tend to edge out locking hash tables, and that the additional overhead our wait-free hash tables is inconsequential.

Additionally, all of our parallel hash tables perform admirably even single-threaded, obviating the need to use separate hash table implementations in single-threaded applications, as is common in language implementations.

Note that demonstrating lock freedom and wait freedom deserves some attention. Absence of locks is not enough to prove lock freedom.

For instance, consider the contention code in Appendix A. It may seem lock-free to some, because it does not use any mutexes, or any other blocking primitive. However, it does use a “spin-lock” to keep threads from starting their work until all threads have done their initialization work. It’s easy to see that one threads’ progress is totally dependent on other threads’, and thus is trivially not lock free.

To demonstrate lock freedom, we also need to show the absence of operations or other logic that would keep threads executing the algorithm from achieving forward progress. Threads may still need to wait on each other’s forward progress, which could involve an unbounded amount of waiting, of course. Generally, then, when arguing for lock-freedom, algorithms should not only be lacking in blocking operations, but also any loops or recursive function should not have cross-thread dependencies, beyond compare-and-swap operations, where failure will only occur at the expense of forward-progress for other threads. While such conditions are not all strictly necessary for lock-freedom, they should be sufficient, and it should be easy for us to convince ourselves that our algorithms meet them.

For wait freedom, we not only need to demonstrate lock freedom, but also need to look at any use of loops or recursion in the algorithm, and demonstrate that those will always complete in finite time.

# Refhat: a reference, single thread only hash table

Our algorithm refhat is a single-threaded hash table that is meant to be simple, fast and correct in single-threaded environment (only), to use as a performance and implementation baseline. We do constrain it to have behavior consistent with our other tables:

1. We use the same approach to bucket deletion.
2. We use the same metrics for table migration and size selection that other tables use.
3. We keep a field that provides an insertion timestamp that is unique per-table.
4. We keep the same API.

In refhat, buckets are represented as follows:

typedef struct {

hatrack\_hash\_t hv;

void \*item;

uint64\_t epoch;

} refhat\_bucket\_t;

As mentioned above, hash values are 128-bit values. This is abstracted away by hatrack\_hash\_t: this is implemented as an \_\_int128\_t if available, or else as a struct containing two int64\_t values.

The “item” field is opaque to the implementation. In could be any of the following:

1. A pointer to a dynamically allocated key/value pair.
2. A single ‘key’ only, as would be used in a ‘set’ implementation.
3. A key and value jammed into a single 64 bit value. For instance, our test suite does this, using the upper 32 bits as a key of type int32\_t, and the lower 32 bits as an value of type int32\_t.

Certainly, it would also be possible to store two separate items, a key and a value, but we did not do that in our low-level tables.

The actual refhat\_t object is fairly straightforward, and is essentially:

typedef struct {

uint64\_t last\_slot;

uint64\_t threshold;

uint64\_t used\_count;

uint64\_t item\_count;

refhat\_bucket\_t \*buckets;

uint64\_t next\_epoch;

} refhat\_t;

* last\_slot is the current table size, minus one (we’ll & with this, instead of doing a % against the table size).
* threshold is set to 75% of the table size.
* used\_count is a count of how many buckets are RESERVED, meaning have an associated hash value (which might be different from the number of live items conceptually in the table).
* item\_count is the number of non-deleted items actually in the table.
* buckets is a pointer to all of the buckets; as we migrate stores, this value gets replaced.
* next\_epoch is a counter, that increments by one with every new insertion into the table, to help duplicate Python’s semantics with regard to insertion ordering.

Note that, while we preserve insertion ordering, we take a different implementation approach with refhat. Python keeps two sets of buckets—the traditional hash bucket contains a pointer into a second, ordered array, where the elements are given out in the order inserted.

That has the advantage that, when the ordering needs to be recovered, no special work needs to be done at all. Whereas, with refhat, the items need to go through an O(n log n) sort operation.

However, as we will see later, the naïve Python approach does NOT work in truly parallel environments. Plus, O(n log n) is generally an acceptable cost, especially given how rarely ordering is required. We do explore these tradeoffs with the lohat family of algorithms, discussed later. But, our other algorithms all keep a timestamp of one form or another, and provide the ability to sort items based on that timestamp.

## The “Get” Operation

Here’s pseudo-code for the refhat item lookup, ignoring status indicators (instead, we use the bottom operator to indicate the item is not present in the table).

**function refhat\_get(table, hash\_value):**

bucket\_index = hash\_value % len(table)

for i in range(len(table)):

current\_bucket = table->buckets[bucket\_index];

if current\_bucket.hv == hash\_value:

if not current\_bucket.epoch: # Bucket contents have been deleted

return ⊥ # bottom operator, item not present

else:

return current\_bucket.item

if not current\_bucket.hv:

return ⊥

bucket\_index = (bucket\_index + 1) % len(table)

return ⊥

Since the table is single threaded, and we will enforce our resizing metrics, the loop is guaranteed to end before cycling through the table, because it must have at least one empty item in it. Nonetheless, if magic happens, and the table really is full to the brim, yet we didn’t find the hash value, then the item is not in the table.

## The Put operation

**function refhat\_put(table, hash\_value, item):**

bucket\_index = hash\_value % len(table)

for i in range(len(table)):

current\_bucket = table->buckets[bucket\_index];

if current\_bucket.hv == hash\_value:

if not current\_bucket.epoch: # Bucket contents previously rmd

table.epoch = table.epoch + 1

current\_bucket.epoch = table.epoch

table.item\_count = table.item\_count + 1

current\_bucket.item = item

return

if not current\_bucket.hv:

if table.used\_count + 1 == table.threshold:

refhat\_migrate(table)

return refhat\_put(table, hash\_value, item)

table.used\_count = table.used\_count + 1

table.item\_count = table.item\_count + 1

table.epoch = table.epoch + 1

current\_bucket.epoch = table.epoch

return

bucket\_index = (bucket\_index + 1) % len(table)

Like the get operation, the loop is guaranteed not to run to completion. If the table gets too full when we insert our new item, refhat\_migrate() will get called (see below), and then we start the process over. While this is looks like a recursive call, if a migration happens, the second attempt to insert is guaranteed to succeed (that won’t be the case with some of our parallel hash tables).

The key not being in the table can be represented two different ways:

1. The hash key has not been inserted during the lifetime of the current store, and we find an “empty” hash bucket (hv == 0).
2. The item was most recently REMOVED, and that removal happened in the lifetime of the current store. In that case, we see the hash value, but the associated epoch is set to 0, indicating the item is not present.

Remember also that the used\_count and item\_count fields are not the same thing. used\_count is counting how many buckets have an associated hash value, whereas item\_count could be a smaller number, as it represents only those buckets containing an actual, active item.

## Add and Replace operations

For the sake of brevity, we elide over these operations. They are effectively identical to our put operation, except that add returns bottom if the item already exists in the table, and replace returns bottom if the item DOESN’T already exist in the table. It’s clear from the put code when the item was present.

## The Remove Operation

**function refhat\_put(table, hash\_value, item):**

bucket\_index = hash\_value % len(table)

for i in range(len(table)):

current\_bucket = table->buckets[bucket\_index];

if current\_bucket.hv == hash\_value:

if not current\_bucket.epoch: # Bucket contents previously rmd

return ⊥

table.item\_count = table.item\_count - 1

current\_bucket.epoch = 0

return **success**

if not current\_bucket.hv:

return ⊥

bucket\_index = (bucket\_index + 1) % len(table)

## The Migration Operation

refhat\_put() and refhat\_add() can both trigger our metrics for migrating the table buckets to new memory. We essentially need to allocate a new store with all fields zeroed out, visit every bucket in the old store, perform the equivalent of refhat\_add() operation, except in the NEW store.

Note that, as the number of buckets doubles, there’s a 50% chance that the initial hash index will change for any particular item. We therefore MUST go through the whole bucket acquisition process.

**function refhat\_migrate(table):**

new\_size = hatrack\_new\_size(size, table.item\_count + 1)

next\_store = calloc(new\_size \* sizeof(refhat\_bucket\_t))

for n in range(len(table)):

src\_bucket = table->buckets[n]

if src\_bucket.epoch: continue

dst\_index = src\_bucket.hv % new\_size;

for i in range(new\_size):

dst\_bucket = next\_store[dst\_index]

if dst\_bucket.hv != 0:

dst\_index = (dst\_index + 1) % new\_size

continue

dst\_bucket.hv = src\_bucket.hv

dst\_bucket.item = src\_bucket.item

dst\_bucket.epoch = src\_bucket.epoch

break

free(table->buckets)

table->buckets = next\_store

table.used\_count = table.item\_count // We’ve purged all deleted items.

table.last\_slot = new\_size – 1

table.threshold = new\_size \* .75

Note that calloc() allocates new memory, and zeros it out. We rely on the zero initialization semantics to tell us whether the destination buckets have been used yet.

In some environments, that could be an expensive operation. However, it generally is incredibly cheap in C—about as cheap as a call to malloc(). Calloc will generally memory-map every allocation to the same read-only pages consisting of all zeros, with copy-on-write semantics.

Everything else here should be straightforward.

## The View Operation

Here, we just allocate enough space for the output, cycle through every bucket, and if there is an item in the bucket, copy the contents out to the output array. Since, being single threaded, we have an accurate count of how many items we’re going to return before we start the operation, nothing here is particularly hard. Note that, given we are implementing in C, this function should return both the array of records in the view, plus an indication of how many records we’ve returned. In real code, we pass a pointer to memory, where the number of items is then stored.

In our single-threaded table, that’s not strictly necessary, since the output size is equal to the table’s item count. However, we do it anyway, to maintain consistency across implementations.

**function refhat\_view(table, sort?):**

view = malloc(sizeof(view\_record) \* table.item\_count)

view\_ix = 0

for bucket in table.buckets:

if not bucket.epoch: continue

view[view\_ix].item = bucket.item

view[view\_ix].epoch = bucket.epoch

view\_ix++

if (sort): quicksort(view, by\_epoch)

return view, table.item\_count

# Duncecap: Minimal locking readers

With our single-threaded reference table out of the way, let’s start following a typical journey through parallelizing hash functions.

The first, obvious thing we can do is put a lock around our hash table. Scripting languages like Python effectively do this, ensuring that no method is being run for a given dictionary in parallel. That obviously puts an upper bound on performance that’s tied to single-threaded performance (i.e., the performance can’t be better than if a single thread never gets suspended by the scheduler, assuming, of course, that the impact of instruction-level parallelism is identical in each case).

That isn’t much of an advantage, so we can look at solutions that allow single writers, but multiple parallel readers. For instance, we can use a POSIX read-write lock around our operations, which allows multiple parallel readers, but makes reading and writing mutually exclusive.

That is, if a writer is holding the lock, no readers can obtain the lock to read. And, conversely, if any readers are holding the lock, writers just need to wait.

Underlying implementations of R/W locks tend to favor writers to help avoid starvation scenarios.

However, one of our key insights with Hatrack, is that we can, fairly easily, make GET operations in hash tables not just lock-free, but also wait-free, if we can meet the following criteria:

1. We have a way to make sure that any reference the reader gets to the current store (the current set of buckets and its associated meta data), will not be deallocated before the read operation is complete.
2. Within a single store, we never “move” buckets, per the discussion above. Buckets are reserved for a particular hash key, for the life of the store.
3. We can atomically update any bucket state, so that the reader always gets a consistent view.

We will address item #1 with our NEXT hash table. Duncecap only addresses #2 and #3. For #3, we actually weaken this property—not ALL the bucket state has to be updated at once—as long as constraint #2 holds true.

Duncecap’s readers do need to use locks to make sure that the current store doesn’t get pulled out from underneath them, but beyond that, it can just happily read from the current store, as long as writers are atomic.

Even if the write thread migrates the table to a new store, we are fine. From the perspective of linearizing our operations (mapping them to a specific moment where they appear to happen), we just consider the read event to occur after the last write prior to the completion of the migration (assuming that no writer writes to that bucket along the way, in which case we’d linearize the read to before or after the write, depending on what value we read).

So, in the face of a table migration, whether the migration is in progress, or completed by the time the read finishes, it can be safe for readers to read.

We’ll go into detail on the migration approach briefly. First, let’s look at how readers register themselves in Duncecap, and how writers make sure they do not accidentally delete stores that a reader is using, after they complete a store migration.

Instead of using a reader-writer lock, we use a standard mutex. Writers will hold the mutex for their entire operation. Readers, however, will only hold the lock for long enough to grab a reference to the store, and then register their use of the current store. In pseudo-code:

function duncecap\_reader\_enter(table):

lock\_mutex(table.mutex)

store = table.current\_store

atomic\_add(table.readers, 1)

unlock\_mutex(table.mutex)

return store

Here, atomic\_add() instruction will add the second value to the first, returning the old value, and do with sequential consistency. This is how the reader registers their use of the store.

However, the reader does NOT need to grab the mutex to say that it is done. All it needs to do is atomically decrement the counter:

function duncecap\_reader\_exit(table):

table\_add(table.readers -1)

Write threads that update the value of table.current\_store must wait till no readers are reading it, prior to deletion. For the moment, we’ll do this with a spin lock:

while(table.readers): pass

In Duncecap, migration happens whenever a new insert would result in more than ¾ of the buckets in the table being reserved for a hash key, which can only happen during a put or add operation. We perform the migration first, and then do the insert once the migration completes. The writer holds the lock the entire time, including through the migration.

Here’s the pseudo-code for our migration operation:

function duncecap\_migrate(table):

new\_store = duncecap\_new\_store(calculate\_new\_size(table))

for bucket in table.current\_store:

if not bucket.deleted:

duncecap\_store\_add(new\_store, bucket.key, bucket.value)

old\_store = table.current\_store

table.current\_store = old\_store

while(table.readers): pass

free(old\_store)

Note that it’s perfectly safe for the write thread to free the old store once all readers have exited, because any new readers will be blocked waiting for the lock, so that it can register its interest again.

The function duncecap\_new\_store() should not only allocate enough memory to hold all the required buckets, it should zero out the memory it allocates. It can use the system allocator.

The function calculate\_new\_size() implements the metrics we discussed above. If the number of active items in the table is 50% of the old store size or above, it returns two times the old store size. If it’s 25% of the old store size or less, it returns half the old store size. Otherwise, it returns the same store size.

# Swimcap: Wait-free readers

Duncecap’s readers require locks, but only to make sure they get a reliable reference to a store, and that it does not get deallocated during a read operation. We can solve this problem with memory management systems.

For instance, if the reader can atomically load a pointer to the current store, then garbage collection would ensure that the store is not deallocated until there are no held references to it. However, garbage collection is rarely used in system languages like C, C++ and Rust, where performance is generally valued highly.

There are many specialized memory management schemes that aim to solve similar problems in parallel programming. For instance, Reference Counters and Hazard Pointers are two well-known approaches, but both are somewhat costly, and difficult to implement.

We base our memory management capabilities in this work on epoch-based memory reclamation techniques. We call our scheme MMM, short for Miniature Memory Manager. Much of what differentiates MMM from other epoch based schemes applies to later algorithms. For now, our operations represent nothing novel.

In MMM, as with many other epoch-based memory management schemes, there is a global timer called the epoch, that monotonically increases. Additionally, there is a global array of “reservations”, where each thread has its own entry, to which it can write, but no other threads can.

The key idea is that memory allocations under MMM management, as with many schemes, is to defer memory reclamation until we can prove no other thread has a reference to the memory.

Instead of doing accounting on a per-pointer basis, epoch-based schemes do it differently. Before threads begin operations on memory under management, they “reserve” the epoch in which the operation starts. For instance:

function mmm\_begin\_basic\_op(): reservations[get\_thread\_id()] = global\_epoch

The read of the global epoch needs to be both atomic, and sequentially consistent, and should become apparent, soon.

And when a thread no longer holds any reference to any items, it removes its reservation:

function mmm\_end\_op(): reservations[get\_thread\_id()] = NO\_RESERVATION

In Hatrack, the global epoch is a 64-bit unsigned integer, and, NO\_RESERVATION is the maximum integer value, as it’s impractical for real applications to ever use 2^64 epochs.

The “reservation” process will help ensure that any MMM-managed memory object that is alive during any reserved epoch will stay alive at least until no reservation for that epoch exists.

To implement this, threads that wish to delete an object call mmm\_retire(). Instead of freeing the memory, mmm\_retire marks the memory, by writing the epoch of retirement into a hidden field in the allocation. The thread then sicks the object on a thread-specific “retirement list”. Periodically, the thread runs the operation, mmm\_empty(), which first scans through the array of thread reservations, looking for the LOWEST active reservation. Then, it scans its retirement list, freeing any data object whose retirement epoch is lower than the lowest active reservation.

The metric for triggering the mmm\_empty() operation can be important, but in the context of Duncecap, it’s sufficient to call mmm\_empty() every n mmm\_retire() operations. Also, on thread exit, threads should wait to exit until all objects they’ve retired are safe to deallocate, so that those objects can be freed, instead of leaking them.

Also, with mmm\_end\_op(), we may need to take care about the compiler re-ordering this call before a thread’s last use of a protected pointer (mmm\_begin\_op requires an atomic read, which will generally result in both a memory fence and a compiler fence). Any sort of fence should do here. Hatrack uses atomic\_signal\_fence() from the C11 atomic standard, which can be used to create a compiler fence.

Note that we have not yet talked about increasing the global epoch value. It is not strictly necessary to change the value if mmm\_empty() is only called when a thread is not holding a reservation (or, alternatively, if a thread is willing to ignore its own reservation). In such a case, mmm\_empty() will only free objects if NO thread is holding a reservation.

While never increasing the epoch would be safe, in the sense that no objects will get freed unnecessarily. However, without incrementing epochs, it’s easy to imagine a scenario where some thread ALWAYS has a reservation, and thus nothing using the scheme ever gets freed.

We therefore should increment our epoch periodically. We could choose to do so every x calls to mmm\_retire(), or we do choose to do so every y calls to our allocation function, mmm\_alloc()[[5]](#footnote-5). And certainly, we could do both, or use some alternate scheme, such as bumping it periodically, based on a timer.

Note that the more often the increment the epoch, the less memory will be sitting around that is not in use, but is not freed. However, the global epoch needs to be incremented atomically, so threads contending to increment the counter could become a bottleneck.

For the purposes of Duncecap, we will increment the counter on every call to mmm\_alloc()

The writer still needs to put a lock around all its activity, but no longer needs a spin lock for deallocation. It does need to call mmm\_retire() on the old store when it is done with it, instead of free(). The writer also may not call mmm\_retire() until the previous store is installed, to avoid any race condition.

With swimcap, as with future algorithms, we have lower-level functions that operate on a store, and the public interface generally will just wrap the call to the lower-level store operation. For instance, swimcap’s get operation is effectively just:

function swimcap\_get(table, hash):

mmm\_start\_basic\_op()

ret = swimcap\_store\_get(table.store\_current, hash)

mmm\_end\_op()

return ret

Similarly, write operations like swimcap\_put() wrap a call to their associated store operation (e.g., swimcap\_store\_put), with code that acquires and releases the table’s write mutex:

function swimcap\_put(table, hash, item):

table.write\_mutex.lock()

ret = swimcap\_store\_put(table.store\_current, table, hash, item)

table.write\_mutex.unlock()

return ret

This approach helps avoid issues like having to deal with recursion when acquiring locks. For instance, when swimcap\_store\_put() decides that a migration is necessary, it can run the migration function, then recursively call itself without having to worry about locking—it will continue to hold the mutex, and does not need to worry about either releasing it or recursively acquiring it, until the entire put() operation is done.

Whenever we allocate a Swimcap instance’s store, we use an MMM allocation function, which allocates memory, while creating the hidden field to store the retirement epoch. That includes when we first initialize a Swimcap object, as well as when the migration operation happens.

In pseudo-code, the migration operation is identical to the duncecap operation, except for the use of MMM operations to protect the store’s memory:

function swimcap\_migrate(table):

new\_store = mmm\_alloc(calculate\_new\_size(table))

for bucket in table.current\_store:

if bucket.hash and not bucket.deleted:

duncecap\_store\_add(new\_store, bucket.key, bucket.value)

old\_store = table.current\_store

table.current\_store = old\_store

mmm\_retire(old\_store)

Due to our use of MMM, the migration function also does NOT busy-wait in the way that Duncecap did.

Now with that sketch of the Swimcap algorithm (and a basic version of MMM), we can discuss how the read operation is wait free. First, note that the read operation is not dependent on any activities of writers. Swimcap acquires whatever store is current at the time of the read, and reads from it, even if writers go on to create a newer store in the process. Even if a writer is concurrently updates the bucket that a reader is reading from, that is transparent to a reader thread. Their read operations on the bucket will be ordered either before or after the writer’s, but they will succeed. And, of course, the reads certainly don’t make use of the lock.

The only loop in the read operation is the linear probe. It only iterates when the hash passed to the operation doesn’t match the hash in a bucket, and the bucket’s associated hash is NOT zero. Each bucket is checked at most once per operation, and the number of buckets checked is bounded by the size of the current store. Therefore, the read operation will always complete in a bounded number of operations.

# Newshat: Multiple readers, multiple writers

It’s nice to know that our read operations are wait free, and thus will not stall, even if writer threads are suspended while holding a lock. But, it’s pretty unfortunate that Swimcap can only handle one simultaneous writer. If we have write-heavy workloads, we will not make good use of multiple processors.

If we’re willing to trade off space, we can use multiple locks to allow multiple simultaneous writers. Let’s start by considering our approach in the absence of table migration. We can add a lock to each bucket. Instead of writers locking the hash table at the beginning of an operation, they attempt to lock buckets they wish to update, before updating them. Specifically, here is pseudo-code for Newshat’s store put operation:

function newshat\_store\_put(store, table, hash, item):

bix = hv & (len(store) – 1)

for (i in range(0, len(store)):

bucket = store.buckets[bix]

bucket.lock()

if bucket.migrated:

bucket.unlock()

newshat\_store\_put(table.store\_current,table, hash, item)

return

if not bucket.hash:

if (table.used\_buckets >= threshold):

bucket.unlock()

newshat\_store\_migrate(store, table)

newshat\_store\_put(table.store\_current, table, hash, item)

bucket.hash = hash

bucket.item = item

bucket.unlock()

return

if bucket.hash != hash:

bucket.unlock()

continue

bucket.item = item

bucket.deleted = false

bucket.unlock()

return

Here, to make our life as easy as possible, we lock buckets before reading or writing any values from them. We first check an additional per-bucket flag, that indicates to writers that the version of the store they’re operating on is no longer the current store, in which case it retries the operation.

In all control paths, the bucket is unlocked. When a migration is deemed necessary, the bucket is also unlocked, to make lock management more straightforward.

The migration function uses an additional lock, making sure that only one thread does the migration work. When a thread realizes a migration is necessary, it first acquires this lock. Once it does, it checks to see if some other thread beat it to the migration.

If not, it then the migration thread locks every bucket, before performing further work. Other writers certainly can update individual buckets in the interim, as update operations do not check table size metrics.

Once all buckets are locked, any new writers to the current store will block on locked buckets. The thread is then free to migrate buckets one-by-one, without concerns of interference.

Once the migration is done, the thread installs the new store, opening it up for readers and new writers. Finally, it unlocks all the buckets in the old store, and then calls mmm\_retire() on the old store.

The pseudocode:

function newshat\_store\_migrate(store, table):

table.mutex.lock()

if store != table.store\_current:

table.mutex.unlock()

return

for bucket in store: bucket.lock()

new\_store = mmm\_alloc(calculate\_new\_size(table))

for bucket in store:

if bucket.hash and not bucket.deleted:

newshat\_store\_add(new\_store, table, bucket.item)

table.store\_current = new\_store

for bucket in store: bucket.unlock()

mmm\_retire(store)

Note that, unlike with Swimcap, write operations are, if we don’t take care, at risk for having the store deallocated out from under them. Therefore, like readers, they need to start and end operations with calls to mmm\_begin\_basic\_op() and mmm\_end\_op().

For newshat\_put, for instance, the top-level operation has pseudocode akin to the read operation:

function newshat\_put(table, hash, item):

mmm\_begin\_basic\_op()

ret = newshat\_store\_put(table.store\_current, table, hash, item)

mmm\_end\_op()

return ret

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

# Oldhat: Extending Hihat to support single-word CAS

# Order-preserving hash tables

Ballcap: Order-preserving, with locks

Lohat: Order-preserving, lock-free

# Exploring time/memory tradeoffs with Lohat-a and Lohat-b

# Witchhat: A fully wait free hash table

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

# Tophat: An approach for general-purpose programming languages

# A High-level Dictionary Implementation

# A High-level Set Implementation

# Performance Analysis

# Related Work

# Conclusions

# Appendix A: The Contention Test

This simple test case shows that, if one wishes to avoid extreme outliers in the face of contention, one should consider wait-free algorithms over lock-free algorithms. In the program (contend.c), eight threads race to update a single 128-bit global variable with unique values as quickly as possible.

#include <pthread.h>

#include <stdatomic.h>

#include <stdio.h>

#include <stdint.h>

typedef \_Atomic int64\_t gate\_t;

static inline void

gate\_init(gate\_t \*gate)

{

atomic\_store(gate, 0);

return;

}

static inline void

gate\_open\_when\_ready(gate\_t \*gate,

int64\_t num\_threads)

{

while (atomic\_load(gate) != num\_threads)

;

atomic\_signal\_fence(memory\_order\_seq\_cst);

atomic\_store(gate, -1);

return;

}

static inline void

gate\_thread\_ready(gate\_t \*gate) {

atomic\_fetch\_add(gate, 1);

while (atomic\_load(gate) != -1)

;

return;

}

#define OPS\_PER\_THREAD 1250000

#define NUM\_THREADS 8

#define MAX\_FAILS 100000

\_Atomic int64\_t total\_fails[MAX\_FAILS];

\_\_thread int64\_t thread\_fails[MAX\_FAILS];

\_Atomic \_\_int128\_t value = ATOMIC\_VAR\_INIT(0);

gate\_t gate;

void \*

run\_thread(void \*my\_id)

{

int64\_t id\_as\_int;

\_\_int128\_t expected, my\_ctr;

uint64\_t i;

uint64\_t retries;

for (i = 0; i < MAX\_FAILS; i++) {

thread\_fails[i] = 0;

}

id\_as\_int = (int64\_t)my\_id;

my\_ctr = ((\_\_int128\_t)id\_as\_int) << 64;

expected = atomic\_load(&value);

gate\_thread\_ready(&gate);

for (i = 0; i < OPS\_PER\_THREAD; i++) {

retries = 0;

while (!atomic\_compare\_exchange\_strong(&value, &expected, my\_ctr)) {

retries++;

}

thread\_fails[retries]++;

my\_ctr++;

}

for (i = 0; i < MAX\_FAILS; i++) {

atomic\_fetch\_add(&total\_fails[i], thread\_fails[i]);

}

return NULL;

}

int

main(void)

{

pthread\_t threads[NUM\_THREADS];

int64\_t i;

uint64\_t total\_ops;

double cur\_pct;

double total\_pct;

gate\_init(&gate);

for (i = 0; i < NUM\_THREADS; i++) {

pthread\_create(&threads[i], NULL, run\_thread, (void \*)(i + 1));

}

for (i = 0; i < MAX\_FAILS; i++) {

total\_fails[i] = 0;

}

gate\_open\_when\_ready(&gate, NUM\_THREADS);

for (i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

}

total\_ops = NUM\_THREADS \* OPS\_PER\_THREAD;

printf("Total ops: %llu\n", total\_ops);

total\_pct = 0;

for (i = 0; i < MAX\_FAILS; i++) {

if (!total\_fails[i]) {

continue;

}

cur\_pct = 100. \* (total\_fails[i] / ((double)total\_ops));

total\_pct += cur\_pct;

printf("%llu fails: %llu (%3f%%; total %3f%%)\n",

i, total\_fails[i], cur\_pct, total\_pct);

}

return 0;

}

I compiled and ran the above program on 1 13” 2020 MacBook Pro with an 8-core M1 chip and 16GB of memory, running on battery power.

The results of a run took up 30 pages, with 1,537 unique fail chain lengths, but we summarize the data here:

Total ops: 10,000,000

0 fails: 1 (0.000010%; total 0.000010%) – Necessarily, the first successful operation.

1 fails: 3,256,907 (32.569070%; total 32.569080%)

2 fails: 1,963,231 (19.632310%; total 52.201390%)

3 fails: 239,343 (2.393430%; total 54.594820%)

4 fails: 2,162,483 (21.624830%; total 76.219650%)

5 fails: 236,307 (2.363070%; total 78.582720%)

6 fails: 143,757 (1.437570%; total 80.020290%)

7 fails: 568,293 (5.682930%; total 85.703220%)

8 fails: 278,309 (2.783090%; total 88.486310%)

9 fails: 90,315 (0.903150%; total 89.389460%)

10 fails: 195,844 (1.958440%; total 91.347900%)

11-25 fails: 568,689 (5.68689%; total 97.03479%)

26-100 fails: 230,207 (2.30207%; total 99.33686%)

101-500 fails: 62,942 (.62942%; total 99.96628%)

501-1000 fails: 2,647 (.02647%; total 99.99275%)

1001-5000 fails: 722 (.00722%; total 99.99997%)

5000+ fails: 3 (.00003%; total 100%)

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)
4. In a case where memory is freed and then reallocated, and in both cases is a store for the same hash table, we still consider that conceptually different stores, and thus conceptually different buckets, even though memory addresses may get reused. [↑](#footnote-ref-4)
5. Actually, for the benefit of other algorithms, MMM has two different allocation functions, and we actually use mmm\_alloc\_committed() in duncecap, which is explained later. Really, depending on the implementation details, either version could be used. [↑](#footnote-ref-5)