A Concurrent Chaining Hash Table

Linearizable, Fast, Wait-free (on x86)

Contents

Introduction

A hash table is a common data-structure used to implement a map from a set of keys to a set of values. Hash tables have their name because they leverage a *hash function* that maps keys to some subset of the integers. Hashed key values can be used as indices into an array where the values themselves are stored, and standard map operations need only consider values that *collide* with a given key (meaning that they hash to the same value). We are concerned with the following operations:

- insert(k, v): insert k with value v into the table; if k is already present, then update its value to v.
- remove(k): remove the data associated with key k.
- lookup(k): return the value associated with the key k in the table, if it is present. Otherwise, return some specific `nil` value.

Related Work

We consider the broad landscape of *general* concurrent hash table that support resizing. Out of scope are hash tables that do not support arbitrary concurrent operations, or that require exotic hash functions. This leaves out interesting work such as Shun and Blelloch (2014) and Feldman, LaBorde, and Dechev (2013).

Non-blocking, Linearizable, Resizeable Hash Tables

We say a hash table is *non-blocking* if all of its operations satisfy a non-blocking progress guarantee (wait-freedom, lock-freedom, or obstruction-freedom). Linearizability is a strong correctness condition for concurrent data-structures (M. P. Herlihy and Wing 1990; see also M. Herlihy and Shavit 2008) that requires all operations to appear to occur atomically at some point between starting and stopping that operation. It is common to require that such a linear history satisfy a sequential specification for the concurrent object.

There are a number of non-blocking hash table implementations, but most do not support a non-blocking re-size operation. An early non-blocking hash table is the split-ordered list of Shalev and Shavit (2006). The most performant example of a hash table in this genre appears to be @liu2014: a chaining hash table (with lock-free and wait-free variants) that relies on a novel *freezable set* data-structure. While the wait-free variant does not appear to scale effectively, the lock-free table provides good performance.

Blocking, Linearizable Hash Tables

There is a large number of performant concurrent hash tables. We will focus on two examples from the recent literature. Fan, Andersen, and Kaminsky (2013) targets the setting of space-efficient cuckoo hash tables that target high-occupancy and prioritize read-heavy workloads such as memcached. This hash table uses a variant of cuckoo hashing as an eviction mechanism, and uses versions to avoid read-side locking. Concurrent writes are synchronized using a striped locking mechanism.

While their implementation does support resizing, we believe it only does so at a significant throughput cost. To our knowledge, the highest-performance concurrent hash table (absent resizing operations) is M. Herlihy, Shavit, and Tzafrir (2008), employing a novel collision-resolution mechanism called "hopscotch hashing". Their implementation provides blocking write operations and obstruction-free reads. The paper describes a resize operation, though the available code for the data-structure does not provide a resizing implementation, nor do the paper's performance measurements include numbers for resizing. While only M. Herlihy, Shavit, and Tzafrir (2008) makes this explicit, we believe that bot of these hash tables are linearizable.

Read-Copy-Update, Relativistic Programming and Relaxing Consistency

Read-Copy-Update (RCU) is a style of concurrent programming that allows writers to wait for concurrent readers to complete before performing an operation. It is growing in popularity and is widely used in the Linux kernel (see "What is RCU? RCU Linux Overview" 2014). While somewhat subtle, using the RCU API is much less subtle This approach is great for read-mostly workloads, but making this approach performant in the setting of multiple concurrent writers appears to be an area of active research (see Howard 2012; Arbel and Attiya 2014). Our use of an epoch-based memory reclamation system to safely resize the hash table is reminiscent of the `rcu_synchronize` operation, the difference is that we use an asynchronous API and use only best-effort operations such as resizing[1] and freeing memory.

A subgenre of RCU hash tables appears to be ones that support *relativistic programming* (RP). These are RCU algorithms that allow read operations to observe write operations in different orders, hence compromising linearizability[2]. The RP hash table in Triplett, McKenney, and Walpole (2011) provides very high throughput for read-heavy operations, and maintains high throughput in the presence of a concurrent resize operation.

Epoch-based Memory Reclamation

A common pitfall in the design and implementation of concurrent data-structures is the question of memory reclamation. In settings where readers should never block, e.g. for scalability reasons, it is difficult for a thread removing data from a data-structure to be certain that there are no concurrent readers or writers currently using that data. There are several standard solutions to this problem; the hash table in this paper uses an *epoch-based* memory reclamation scheme (EBMRS).[3]

An EBMRS gives each active thread an epoch counter (which is 0, 1 or 2), and an `active` flag. Upon beginning a concurrent operation, threads set their epoch counter to a global epoch counter and set their `active` flag. Logical remove operations where a node is rendered unreachable to new threads append removed nodes to a garbage list corresponding to the current global epoch[4].

Threads will periodically attempt to perform a garbage collection operation by scanning the list of active threads and checking if all of them have counters the same as the current global epoch counter. When this occurs, the epoch counter is incremented modulo 3 and the contents of the oldest epoch's garbage list are freed. The central argument for why this is safe is that all active threads started at a later epoch than when these nodes were unlinked, so none of them can hold a reference to any of these unlinked nodes. Furthermore we need not worry about inactive threads because they will join a later epoch if they become active, and they are (by assumption) not currently in a critical section.

This paper uses an EBMRS for both memory reclamation, as well as for safely growing the hash table. The latter functionality requires extending the reclamation library with the ability to run arbitrary callbacks when the current epoch has quiesced. This functionality bares some resemblance to the *wait-for-readers* function used in RCU, the difference being that it is asynchronous, and we wait for everyone.

[1]Note that our choice of a chaining implementation is *why* resizing can afford to be best-effort only, while maintaining the rest of our progress guarantees.

[2]The precise relationship between RCU, RP and linearizability appears somewhat murky to an outsider (such as the author of this document). There are some RCU data-structures that provide linearizability. However, RP (as opposed to merely RCU) seems to require some relaxation of linearizability.

[3]Other solutions to this problem include hazard pointers Michael (2004), mixed approaches between EBMR and hazard pointers Dice, Herlihy, and Kogan (2016), and writing all of the code in a language with garbage collection.

[4]These per-epoch garbage lists are sometimes called “limbo lists” because they represent “dead” nodes whose memory has yet to be freed.

A Lazy, Wait-Free Hash Bucket

Here we detail the design of the bucket for the hash table: the `LazySet` data-structure. It is essentially a chunked linked-list; we say it is "lazy" because elements are only ever *logically* deleted, with separate garbage collection routines ensuring that the memory overhead of a set with sufficient `remove` calls does not grow without bound. We provide Rust[5] psuedo-code[6] for the code listings. For a more detailed overview refer to Appendix C. All of the proofs below assume that the bucket-level `head_ptr` counters *do not* overflow. We refer readers to Appendix A for a justification of why this is reasonable.

Data Layout

A `LazySet` takes two type parameters `K` and `V` for key and value types respectively. The various trait bounds on the `K` parameter express that the table requires keys to support hashing, equality comparison and copying. Lastly, the `Atomic` type is an atomic pointer type, `AtomicIsize` is an atomic `intptr_t` type.

```
struct LazySet<K: Eq + Hash + Clone, V> {
    head_ptr: AtomicIsize,
    head: Atomic<Segment<(K, Option<V>)>>,
    last_snapshot_len: AtomicUsize,
}
struct Segment<T> {
    id: AtomicIsize,
    data: [MarkedCell<T>; SEG_SIZE],
    next: Atomic<Segment<T>>,
}
struct MarkedCell<T> {data: Atomic<T>, stamp: Stamp}
```

A `LazySet` is essentially a pointer to the linked-list structure called a `Segment`. `Segment`'s form a linked list of chunks, where each chunk holds `SEG_SIZE` `MarkedCells`, each pointers to actual values in the list. `MarkedCells` also include a `Stamp`, which we will cover below.

The `LazySet` `head` pointer points to a `Segment` with the highest `id`. A bucket gets initialized to point to an empty `Segment` with `id` 0; values are added to the list by creating a new segment with `next` pointer set to the current head, with an `id` of one greater than the head's `id`.

In this way, a `Segment` represents a lazily initialized infinite array. Items are added to a `Segment` by incrementing the `LazySet`'s `head_ptr` value, and the indexing `head_ptr % SEG_SIZE` into the `Segment` with `id` `floor(head_ptr / SEG_SIZE)`.

[5]Rust is a fairly recent language aiming at providing low-level control over memory, a reasonable level of abstraction, and memory safety. It has a minimal runtime, giving it comparable performance characteristics to the C/C++ family of languages.

[6]While the psuedo-code is valid Rust syntax, it differs from the actual implementation in a few ways. There are several small optimizations that inhibit readability for the code that have been omitted for this document, e.g. replacing integer division and mod with bitwise operations. The major simplifications are in the omission of explicit lifetime parameters, and of explicit calls into the EBMR subsystem; these aspects are necessary to avoid memory leaks, but only add noise when discussing the algorithm's correctness and liveness properties. We also omit threading through thread-local caches for `Segments` to avoid wasted allocations.

One subtlety is that we store `Option`al values in the cells. These will become more important when describing the process of removing elements when there is a concurrent resize operation on the hash table.

Stamps

A `Stamp` is a word of memory used to store metadata about a given key-value pair in a `MarkedCell`. It is implemented as an `AtomicUsize` in Rust, where the least significant bit is used to indicate if a given cell has been *logically deleted*, and the remaining bits are the word size − 1-most significant bits of the hash of the relevant key. Cells that are logically deleted will not be considered by lookup operations.

```rust
struct Stamp(AtomicUsize);
impl Stamp {
    //Note: in Rust `!` is bitwise complement.
    fn init(&self, v: usize) { self.0.store(v & ((!0) << 1)); }
    fn delete(&self) { self.0.store(self.0.load() | 1) }
    fn is_deleted(&self) -> bool { (self.0.load() & 1) == 1 }
    fn matches(&self, h: usize) -> bool { (self.0.load() | 1) == (h | 1) }}
```

Storing a portion of the key's hash reduces the number of indirections for a given lookup (Fan, Andersen, and Kaminsky (2013) is the first implementation of a technique such as this; they do it for similar reasons). While this incurs a space overhead of a single word per key, there are independent reasons why a "logically deleted" bit that is updated atomically should be word-aligned. In that way, the space overhead is a cost we have to pay, given the overall design.

Insertion

Adding to a `LazySet` involves atomically incrementing (with a fetch-add instruction) the `head_ptr` index, and then searching for this index in the array.

```rust
fn add(&self, key: K, val: Option<V>)  {
    let my_ind = self.head_ptr.fetch_add(1);
    let cell = self.search_forward(my_ind)
        .or_else(|| self.search_backward(my_ind))
        .unwrap();
    let h = hash(key);
    cell.data.store(Some(Owned::new((key, val))));
    cell.stamp.init(h); // (key, val) only become visible when stamp is set
}
fn search_forward(&self, ind: isize) -> Option<&MarkedCell<(K, Option<V>)>> {
    let (seg_id, seg_ind) = split_index(ind);
    while let Some(seg) = self.head.load() {
        let cur_id = seg.id.load();
        if cur_id == seg_id {
            return Some(seg.data[seg_ind]);
16          } else if seg_id < cur_id {
17              return None;
18          }
19          let new_seg = Owned::new(Segment::new(cur_id + 1, Some(seg)));
20          self.head.cas(Some(seg), Some(new_seg));
21      }
22  }
23  fn search_backward(&self, ind: isize) -> Option<&MarkedCell<(K, Option<V>)>> {
24      let (seg_id, seg_ind) = split_index(ind);
25      let mut cur = &self.head;
26      while let Some(seg) = cur.load() {
27          let cur_id = seg.id.load();
28          if cur_id == seg_id {
29              return Some(seg.data.get_unchecked(seg_ind));
30          }
31          cur = &seg.next;
32      }
33      None
34  }
35  fn split_index(ind: isize) -> (isize, usize) {
36      let seg = ind / SEG_SIZE;
37      let seg_ind = (ind as usize) % SEG_SIZE;
38      (seg, seg_ind)
39  }
```

A note on Rust constructs: the `or_else` method runs the closure it takes as an argument if its receiver is `None`. The `unwrap` method asserts an `Option<T>` is non-null, returning its contents or halting the program.

The core idea behind `add` is to use an atomic fetch-add operation (line 2) to acquire an index into the "infinite array" represented by a segment. Once such a cell is acquired, the only thing left to do is to search for the cell at that index (lines 3–5), and then store `key` and `val` in that cell (line 6). We can make this more explicit.

Definition 1 *We define the* logical index *of a* `MarkedCell` $m = s.\mathtt{data}[i]$ *in some* `Segment` s *to be* $s.\mathtt{id} \cdot \mathit{SEG\_SIZE} + i$.

Given this definition, along with the fact that `add` is the only `LazySet` method that modifies the `head` pointer, we can reason about some important properties of `add`.

Lemma 1 *Lines 3–5 in the* `add` *function store a reference to of a unique* `MarkedCell` *reachable from* `self` *with logical index* `my_ind` *into* `cell`.

To show uniqueness, it suffices to show that (positive) `Segment` IDs are allocated contiguously (i.e. the order of segments is $0, 1, 2, \ldots$) without duplicates. No code modifies an ID once a segment is successfully CAS-ed into `head` (line 18), which guarantees that a new `Segment` will always point to a `Segment` with an `id` one less than its own (unless it is completely full, in which case it may be garbage collected in the future — see below). This means there is a bijective correspondence between logical index and (`Segment` id, `Segment` index pairs), this bijection is exactly the one computed by `split_index`. The uniqueness of values of `my_ind` is guaranteed by the implementation of fetch-add; the cells corresponding to `my_ind` are therefore unique on a per-thread basis.

The `search_forward` operation starts by examining this segment and testing if it has the proper `id` (line 12). If the `id` is correct, it suffices to index into the current segment and return its contents (line 13). The two remaining cases in the search are if the `id` is too small, and if it is too large.

If `id` is too small, a new segment is allocated (this is what `Owned::new` accomplishes) pointing to the current head, and the thread attempts to `cas` this new segment into the `head` pointer (line 18). If this succeeds, then the search continues, as there is a new segment to which we can apply the same checks. If the `cas` fails, there is no need to retry because another thread must have performed the same operation and succeeded.

If the `id` is too large, `search_forward` immediately returns (line 15). This could occur if between the fetch-add and the load, multiple `search_forward` operations succeeded, thereby installing a later segment at `head` and causing the cell corresponding to `my_ind` to lie behind `head`. This is the only possible scenario for this condition to hold, as `add`s in the "too small" path are the only operations to re-assign to `head`. In this case the cell corresponding to the caller's logical index must be reachable from `head`, and `search_backward` merely follows `next` pointers until it finds the proper index. Note that the two values loaded from `head` in the two search methods may not be the same, but the first value must be reachable from the second.□

Lemma 2 *If fetch-add is wait-free, then* `add` *is wait-free.*

To show wait-freedom, we need only show that `search_forward` and `search_backward` are wait-free, as fetch-add and store operations are wait-free (by assumption).

`search_forward` consists of a loop that is will terminate after a thread finds the proper segment. If the initial load of `head` (line 10) corresponds to an `id` of x, and `ind` corresponds to a segment with `id` y, then the loop will break after at most $y - x$ iterations. This follows directly from the argument for `cas` failures still guaranteeing progress above.

`search_backward` is simply a linked-list traversal. The only way that it could not terminate would be if an unbounded number of additional nodes were added *below* head. While more nodes *can* be added below `head` as part of a `back_fill` operation (see below), the number of `back_fills` concurrent with a given operation on a hash table is guaranteed to be bounded by the EBMRS. We conclude that both search methods will always terminate in a finite number of steps, and hence that `add` is wait-free. □

Note

Modern Intel x86 machines provide a atomic fetch-add instruction. Such an instruction always succeeds and is typically much faster than implementing fetch-add in a CAS loop. On architectures that do not natively support atomic fetch-add, implementing fetch-add in terms of CAS (or any equivalent primitive that guarantees progress) is lock-free and can still be quite fast, though we have not examined performance of these data-structures on non-x86 architectures.

Lookups

A lookup operation simply loads the value of `head` and traverses backward through the `Segment` until it finds a cell with the requisite key, or has reached the end of the list. Lookup operations distinguish between failures to lookup that observed the key as deleted, and ones that do not observe the key.

```
struct SegCursor<T> { ix: usize, cur_seg: Option<Shared<'a, Segment<T>>>}
impl<T> Iterator for SegCursor<T> {
    type Item = &MarkedCell<T>;
    fn next(&mut self) -> Option<Self::Item> {
        match self.cur_seg {
            Some(ptr) => {
                let cell = ptr.data[self.ix];
                if self.ix == 0 {
                    self.cur_seg = ptr.next.load();
                    self.ix = SEG_SIZE - 1;
                } else {
                    self.ix -= 1;
                }
                Some(cell)}
            None => None}}}
enum LookupResult<T> {Found(T), Deleted, NotFound}
fn lookup(&self, key: &K) -> LookupResult<&V> {
    let h = hash(key);
    match self.search_kv(h, key) {
        Found(cell) => {
            let v_opt = &cell.data.load().unwrap().1;
            let v_unwrap = &v_opt.as_ref().unwrap();
            Found(v_unwrap)
        }
        Deleted => Deleted, NotFound => NotFound,
}}
```

The core lookup logic occurs in `search_kv`

```
fn search_kv(&self, hash: usize, k: &K) -> LookupResult<&MarkedCell<(K, Option<V>)>> {
    for cell in self.head.load().unwrap().iter_raw() {
        if cell.stamp.matches(hash) {
            if let Some(data) = cell.data.load() {
                if data.0 == *k {
                    return if cell.stamp.is_deleted() {
                        Deleted
                    } else {
                        Found(cell)
    };}}}}
    return NotFound; }
```

It is fairly clear that, for some state of a given set over time, this does perform a lookup of a given key if it is present. The more challenging correctness concern with `lookup` is with linearizabilty.

Lemma 3 *`lookup` is wait-free.*

This follows from `back_fill` only executing a bounded number of times concurrently with any other operations, mentioned above. □

Removal

There are two algorithms for removing a key from a `LazySet`. One (the *standard*) method flips the "deleted" bit in a key's stamp if the key is present, and the other (the *backup*) method appends a record with value `None` to the set. In practice, the standard method performs better so it is called as the default. The backup method is used to ensure the remove operation is not lost in the midst of a concurrent resize operation.

Standard Removal

The standard remove operation is a straight-forward traversal of the `LazySet`, followed by a possible store operation on a relevant `Stamp`. It is effectively a `lookup` followed by a modification of the relevant cell's `stamp`.

```
fn remove_standard(&self, key: &K) {
    if let LookupResult::Found(n) = self.search_kv(&guard, hash, key) {
       if !n.stamp.is_deleted(Relaxed) {
         n.stamp.delete(Acquire, Release);};}}}
```

Where wait-freedom follows from the fact that `search_kv` is wait-free.

Backup Removal

The implementatino of backup removal is the same as `add` except the stamp is deleted before it is initialized (line 8). Wait-freedom therefore follows from the wait-freedom of `add`.

Linearizability

We now show that an arbitrary history composed of `add`, `remove` and `lookup` operations is linearizable; furthermore it is always possible to linearize these operations in a history that corresponds to a correct sequential specification for such an object. We begin by considering only `add` and `lookup` operations. For each class of operation will have "pending" and "commit" events. These events correspond to locations in the code for these operations, and hence correspond to valid linearization points The linearized event is given its own notation.

- `lookup` operations begin when they load the contents of a `head` pointer. If a lookup is searching for an element e this event is denoted l^e. The corresponding commit operation is denoted $l^e_{e',i}$ where e' is the last observed element and i is its logical index. If e and e' match, then the lookup either returned "deleted" or the corresponding value. We denote the linearized operation as l^e_*.

- An `add` operation for element e is pending at the fetch-add in `add`. If fetch-add returns a logical index i this event is denoted $a_{e,i}$. The operation commits at the store operation on the record's new `Stamp`, this event is represented as $a'_{e,i}$. The linearized operation is notated as a_e.

We use the notation $x \prec y$ to specify that, in some history of events, the event x precedes the event y.

The Linearization Procedure

We consider each pending operation and consider the points at which its corresponding final operation lands in the linearized history. We justify why these linearization points are valid when it is not obvious.

- *Standard Remove*: A standard remove satisfies the same conditions as `lookup` operation, except the commit operation occurs when the `Stamp` is marked as deleted. See the `lookup` case for details on linearizing a standard remove operation.
- *Backup Remove*: A standard remove behaves exactly like an add operation, except the commit operation occurs when the deleted `Stamp` is loaded into the requisite cell.
- *Lookups*: Given a lookup pending operation l^e, the only case where we do not linearize at its commit operation is when there is an add operation satisfying the following, given $i < j$, and any event e'

$$l^e \prec a_{e,j} \prec a'_{e,j} \prec l^e_{e',i}$$

  In this case, we linearize l^e_* just before the relevant a_e operation. Because we (inductively) assume linearizability, and the execution of this add operation occurs entirely during the traversal of this lookup, this point in time overlaps the execution of the lookup and is therefore a valid linearization point. For *all* lookups of the same element for which this history applies, the question remains how they themselves should be ordered. Any permutation of these lookup operations is permitted, so long as any non-repeat remove operations are ordered before any lookups that observe the element with the deleted bit set.
- *Add*: Given a pending $a_{e,i}$, there are two cases:

  1) If there is a $j > i$ such that
  $$a_{e,i} \prec a_{e,j} \prec a'_{e,j} \prec a'_{e,i}$$
  Then a_e is linearized as the latest event in the history that precedes any lookups that see its element and also before the commit corresponding to $a'_{e,j}$. This event overlaps the execution of the add with pending operation $a_{e,i}$ because the operation $a_{e,j}$ will (inductively) be linearized at some point during its execution, and that execution time is a sub-span of the time spent on $a_{e,i}$.
  2) Otherwise, linearize $a_{e,i}$ at $a'_{e,i}$. This relies on the fact that for all $e, e', j > i$ the semantics of fetch-add guarantee that $a_{e,i} \prec a_{e',j}$

Theorem 1 (Linearizability) *An insertion or modification of a cell precedes any lookups that observe that action in this linearizable ordering.*

This follows straight-forwardly from the definition of the *Add* and *Lookup* rules. □

Backfilling

Resizing a hash table involves moving elements from an old set of buckets to their corresponding buckets in a new set. In order to accomplish this without blocking any other operations, a resizing thread takes all live elements from an old bucket and *prepends* them to the `LazySets` to which they correspond. One strange aspect of this issue is that backfilled `Segments` have negative `ids`. No attempt is made to ensure successive backfilled segments have different `ids`, though this would be possible to add.

This may appear to invalidate linearizability arguments that rely on a well-defined and unique notion of logical index. However, all of the arguments that use this notion (e.g.~for comparisons) are used when there is an overlapping add operation. All that is required to have this continue to work is for any node in a backfilled segment to be considered to have a lower logical index than a new one.

```
fn back_fill(&self, mut v: Vec<(usize, K, V)>) {
    if v.is_empty() { return; }
    let mut count = -1;         // backwards-moving segment id
    let mut current_index = 0; // index into data array of current Segment
    // Head of new segment list
    let seg: Segment<(K, Option<V>)> = Segment::new(-1, None);
    let target_len = v.len();
    {   // Current segment being filled
        let mut current_seg = &seg;
        while let Some((h, k, v)) = v.pop() {
            if current_index == SEG_SIZE {
                count -= 1;
                current_index = 0;
                current_seg.next.store(new_seg(count));
                current_seg = &*current_seg.next.load().unwrap();}
            let cell = current_seg.data[current_index];
            cell.data.store(Some(Owned::new((k, Some(v)))));
            cell.stamp.init(h);
            current_index += 1;
    }}
    let mut seg_try = Owned::new(seg);
    let mut cur = self.head.load().unwrap();
    loop { // add seg_try to end of list. This normally runs once.
        while let Some(next) = cur.next.load() { cur = next; }
        match cur.next.cas_and_ref(None, seg_try) {
            Ok(_) => { return; }
            Err(seg1) => seg_try = seg1,
};}}
```

We briefly observe that `back_fill` is lock-free, though we only require it be obstruction free for the purposes of the hash table. There are two loops to consider. The first loop (line 10) operates only on thread-local data and only executes for as long as there are elements remaining in `v`. The CAS loop (line 23) first traverses to the end of the list. It then attempts to CAS the end of the list from a null pointer to `seg`. If this fails, it can only be because another `back_fill` operation succeeded.

Lastly, the resize operation guarantees that only one thread performs a back fill at once, and that only a constant number of back fills occur concurrently with any given operation on a hash bucket.

Garbage Collection

While the implementation of the bucket thus-far is correct in the sense that it provides a wait-free, linearizable map data-structure, it has the drawback that it leaks memory: remove operations only logically delete values, leaving reclamation to other methods.

We use the standard technique of an EBMRS to unlink and then reclaim a segment consisting entirely of deleted nodes; this garbage collection process is triggered every few `add` operations, and only one thread is permitted to perform this operation at a time. While this GC will always complete in a bounded number of steps, there is no non-blocking guarantee that the hash bucket will not leak memory, as the chosen GC thread could be de-scheduled for an arbitrary amount of time.

The Hash Table

Most of the complexity of the overall data-structure is contained in the bucket design. The hash table itself if essentially an array of `LazySets`, where individual operations are propagated to the proper set by first hashing the input key, modding that hash by the current array length, and performing the operation on that index in the array. Such a design would provide good performance for a while, but throughput would eventually degrade as linear lookups dominate execution time.

In order to provide good performance without prior knowledge of maximum table size, it is common to dynamically resize a hash table to accommodate more or fewer elements without substantially degrading the data-structure's performance or space overhead. Performing such a resizing operation in a concurrent setting is a rather fraught task. While most concurrent hash tables support some form of resizing, fewer do so without blocking other operations, and fewer still actually include resizing operations in their published suite of benchmarks (Triplett, McKenney, and Walpole 2011 is an exception).

Because resizing is not an operation that impacts correctness so much as performance, we do not provide a resizing operation with a non-blocking progress guarantee. Note that this is not the case for some non-chaining hash tables, where a lack of space or an insufficient collision resolution mechanism can effectively force a resize operation. We *do* guarantee that resizes do not block any add, lookup or remove operations; in practice resize operations occur and complete reliably even under high-load scenarios where there is a greater risk of resizes being preempted.

Data Layout

As noted above a `HashTable` is essentially an array of buckets. To facilitate resizing, we also keep track of an old array of buckets `prev_buckets`, a counter to estimate table load `grow_votes` and a flag `growing` to indicate if the hash table is currently undergoing a resize operation.

```
struct HashTable<K: Eq + Hash + Clone, V: Clone> {
    buckets: Atomic<Vec<LazySet<K, V>>>,
    prev_buckets: Atomic<Vec<LazySet<K, V>>>,
    grow_factor: usize,
    grow_votes: AtomicIsize,
    growing: AtomicBool,
}
```

Resizing

The key insight in implementing resizing is to use the EBMRS to also run arbitrary callbacks. The high-level algorithm for resizing is as follows:

(1) Allocate a new bucket array. Copy `buckets` to `prev_buckets` move the new bucket array into `buckets`.

(2) In a closure executed after all current operations have returned (`deferred_closure` in the psuedo-code), re-hash all members of `prev_buckets` into `Segments` broken down by hash value modulo the length of the new `buckets`, and `back_fill` them into those `Segments`.

(3) Null out the pointer to `prev_buckets`, free all memory it contains once all active operations have completed (`deferred_delete` in psuedo-code).

```
fn rebucket<K: Hash + Eq + Clone, V: Clone>(
    bucket: &LazySet<K, V>, new_mod: usize)
    -> Vec<(usize, Vec<(usize, K, V)>)> {
        let mut res: Vec<(usize, Vec<(usize, K, V)>)> = Vec::new();
        for (_, (k, v)) in bucket.iter_live(Relaxed, guard) {
            let hash = hash(k);
            let bucket_ind = hash % new_mod;
            let mut to_insert = Some((hash, k.clone(), v.clone().unwrap()));
            for &mut (bkt, ref mut vec) in res.iter_mut() {
                if bkt == bucket_ind {
                    vec.push(to_insert.take().unwrap());
                    break;}}
            if let Some(t) = to_insert { res.push((bucket_ind, vec![t]));}}
        deferred_delete(bucket)
        res
}
fn try_grow<F: Fn(usize) -> usize>(&self, trans: F) {
        if self.growing.compare_and_swap(false, true) { return; }
19          let start_votes = self.grow_votes.load();
20          let cur_buckets = self.buckets.load().unwrap();
21          let new_mod = trans(cur_buckets.len());
22          let new_buckets = HashTable::make_buckets(new_mod);
23          //these always succeed
24          self.prev_buckets.cas_shared(None, Some(cur_buckets));
25          self.buckets.cas(Some(cur_buckets), Some(new_buckets));
26          let current_grow_votes = self.grow_votes.load();
27          deferred_closure(Box::new(move || {
28              for i in self.prev_buckets.load().unwrap().iter() {
29                  let mut new_buckets = rebucket(i, new_mod);
30                  while let Some((hash, vals)) = new_buckets.pop() {
31                      self.buckets.load().unwrwap()[hash].back_fill(vals);
32                  }}
33                  deferred_closure(Box::new(move || {
34                      self.prev_buckets.store(None);
35                      self.grow_votes.fetch_add(-current_grow_votes);
36                      self.growing.compare_and_swap(true, false);
37                      free(self.prev_buckets);
38                  }));
39              }));
40  }}
```

The `iter_live` iterator is a wrapper on top of `iter_raw` that de-references the key-value pairs and keeps a set of seen elements (added or deleted) and yields only elements it has not yet seen. `make_buckets` is a function that creates and initializes a new bucket array. Note that the `deferred_closure` in `try_grow` itself enqueues deferred operations (both as part of `rebucket` and as a `deferred_closure`. The first deferral is to ensure that there are no active mutating operations acting on the old bucket array. As we will see, after the CAS on line 25, all new `add` or `remove` operations will operate on the new bucket array. The fact that the old array has quiesced guarantees that the `back_fills` (line 31) do not miss any nodes.

We briefly establish that the "bounded `back_fill`" property assumed in arguing for the wait-freedom of the hash buckets is satisfied.

Lemma 4 *For any `add`, `remove`, or `lookup` operation on a `LazySet` within a `HashTable`, the number of `back_fills` concurrent with that operation is bounded.*

`back_fills` are only executed during a `try_grow` operation. Furthermore, `try_grow` can only be executed by one thread at a time; this is enforced by the CAS attempt on the `growing` flag (line 18). Any `add`, `remove`, or `lookup` operation is executed in a particular epoch of the EBMRS; if that epoch overlaps the epoch in which the initial deferred closure (line 27) is run, then there may be some number of concurrent `back_fills`. As we will see later, `trans` will only ever set modify the size of the table by a power of two. If `trans` shrinks the table by a factor of k then the rebucketing loop (line 28) will potentially exececute k `back_fills`. If `trans` grows the table, then the number of concurrent `back_fills` is at most 1, as buckets are actually split by the same factor.

While we have established that there is a bounded number of `back_fills` for any given bucket during a resize operation, we must still argue that no additional `back_fills` will execute concurrently with any concurrent bucket-level operations. For this, observe that the next possible `try_grow` call that succeeds in acquiring the `growing` bit must occur after the execution of the `deferred_closure` at line 33. But that closure is run in a later epoch than the `back_fills`, and the only way it can be run is if all operations concurrent with the loop at line 28 have completed. This gives us the claim. □

Add, Remove and Lookup

Compared with the mechanics of resizing, the standard table opearations are fairly simple. All operations hash their keys and perform the corresponding operation on the bucket corresponding to this hash modulo the length of `buckets`. `lookup` operations search `prev_buckets` if their initial lookup returns `NotFound`. `add` and `remove` operations only touch `buckets`; any modification of `prev_buckets` would risk losing those updates as there is no synchronization with concurrent `back_fill` operations. Whether or not to use `remove_standard` or `remove_backup` is determined by the current value of the `growing` flag. We must use `remove_backup` if there is a concurrent grow operation because `remove_standard` cannot `delete` an element that is not in the bucket. `remove_backup` has the effect of deleting not only previous ocurrences of a key in the table, but also any such ocurrences that are later added as a result of a `back_fill` operation.

```
fn remove(&self, key: &K) {
    let bucket = get_bucket(self.buckets.load().unwrap(), key);
    if self.growing.load() { bucket.remove_backup(key) }
    else { bucket.remove_standard(key) }
}
fn add(&self, key: K, val: V) {
    let mut bucket = get_bucket(self.buckets.load().unwrap(), &key);
    if self.grow_factor != 1 && bucket.last_snapshot_len.load() > SEG_LOAD_FACTOR {
        if self.grow_votes.fetch_add(1) as usize > self.grow_threshold() {
            self.try_grow(&guard, |n| n * self.grow_factor);
            bucket = get_bucket(self.buckets.load().unwrap(), &key);}}
    bucket.add(key, val)
}
fn lookup(&self, key: &K) -> Option<&V> {
    let bucket = get_bucket(self.buckets.load().unwrap(), key);
    match bucket.lookup(key) {
        Found(n) => Some(n), Deleted => None,
        NotFound => {
            if let Some(buckets) = self.prev_buckets.load() {
                let bucket = get_bucket(buckets, key);
                if let Found(elt) = bucket.lookup(key) { return Some(elt); }
            }
            None
}}}
fn get_bucket(buckets: &Vec<LazySet<K, V>>, key: &K) -> &LazySet<K, V> {
    let hash = hash(key);
    buckets[hash % buckets.len()]
}
```

We trigger a growing operation by checking a `last_snapshot_len` value after each `add` operation, and incrementing a table-level counter if it is over some threshold. Once this counter (`grow_votes`) is over the `grow_threshold` for a given table size, a grow operation is attempted. `last_snapshot_len` is updated after each bucket-level GC. This psuedo-code does not have a complementary heuristic for shrinking the table, though `try_grow` is entirely agnostic to whether the table grows or shrinks.

Theorem 2 *Assuming a wait-free fetch-add operation, add, lookup, and remove are wait-free.*

All of the above operations only execute a small number of wait-free bucket-level operations. The only exception to this rule is an `add` that successfully executes a `grow` operation, but the grow operation (and all of its closures) only execute a bounded number of operations. The only loops resulting from `grow` are in the rebucket loop, which only operates on a quiesced (hence bounded in size) `prev_buckets` array. We conclude that these operation are all wait-free. □.

Theorem 3 *The add, lookup and remove operations are linearizable, and all lookups that observe the results of an add or remove operation are linearized after that operation.*

The fact that these operations are linearizable follows directly from the linearizability of the corresponding bucket-level operations, and the fact that linearizable structures are *compositional* (see M. Herlihy and Shavit 2008, chap. 3). The correctness properties largely follow from the correctness of the underlying bucker operations, because causally related operations will interact with the same bucket. The only added layer of difficulty comes from the presence of resize operations. However, the fact that `add` and `remove` operations only modify buckets in the current `buckets` array, combined with the fact that old buckets only cease to be visible after all `back_fills` become visible, means that no `lookup` operation can fail to see relevant `adds` or `removes` linearized before it. Nor can it see any operations linearized after it, as those operations will all take effect in the new bucket. □

Performance (TODO: produce graphs, analysis)

- Use all the tables we can get our hands on
- Modify key ranges, %reads/writes/removes
- Scaling up to 32 threads on the Xeon.

Conclusion And Future Work

We have described the implementation of a fast, scalable concurrent hash table; faster than any other general hash table that we know of. Furthermore we provide very strong progress and consistency guarantees, along with support for concurrent resize operations.

For future work, in addition to implementing additional low-level optimizations (see Appendix D), we believe that it is possible to implement fast concurrent, serializable transactions on top of this hash table design. This would help to reduce the limitations of this data-structure compared to a more conventional hash table, such as iteration operations and consistent bulk updates.

Appendix A: Overflowing bucket-level counters

The proofs above tacitly assume that there will be no overflow of the bucket-level counter `head_ptr`, currently stored as a 64-bit signed integer allowing for 63 increments before an overflow occurs. While such an overflow, if possible, would take a long time to accomplish – it would take 68 years of fetch-adds at a rate of 2^{32} per second to overflow this counter – we can go further. Here, we argue that such an overflow is impossible. We consider the case where we use a 63 bit counter, but this argument generalizes to any setting where there a (word size $-$ 1)-bit counter is in use.

Consider the setting in which there are a large number of `add` operations in the same bucket of a `HashTable`. These will eventually trigger a `try_grow` that will effectively reset a bucket-level counter. Now, if an attacker had control over the scheduler (or was just very very lucky) they could starve this grow operation by perminently preempting an active thread, preventing any re-bucketing and hence any future grows from ocurring. In this setting, however, all `remove` operations will append a record to the relevant bucket and there is no mechanism to perform a bucket-level GC. Therefore, every increment to the bucket-level counter incurs *at minimum* a cost of 2 words of memory (one for the `Stamp` in the `MarkedCell` and one for the `Optional` bit in the `(K, Optional<V>)` tuple). But then $2^{63}+1$ operations on the bucket (causing an overflow) would consume $2^{64}+2$ words of memory, which is beyond the maximum addressable space of any 64-bit architeture.

Appendix B: Memory Ordering

The desciription of this algorithm has the notable limitation that it eschews discussions of memory ordering, i.e. the extent to which certain barriers are necessary on atomic loads and stores in order to maintain correctness. This is not an aspect of the algorithm that has been completely fleshed out; the author is a relative novice when it comes to these ordering constraints, and development of the algorithm was performed on an x86 machine, which has relatively forgiving ordering semantics by default. It is safe to say that some barriers are required, e.g. re-assigning `prev_buckets` and `buckets` in `try_grow` involves two CAS operations which must be observed in program order, necessitating a barrier between them. Rust currently uses a subset of the LLVM memory orderings[7] to synchronize atomic operations, so fine-grained control of the placement of these barriers is possible.

Appendix C: Rust Background

This is intended primarily as an introduction to the notation used in this paper, it is *not* intended as an introduction to the Rust language. For that, see the various recommended options from the Rust community. One of the most exciting Rust features is how it deals with memory safety. We largely omit discussion of that here: we have removed lifetime parameters on references, and also removed unsafe blocks (there are plenty in the current implementation).

[7]Rust supports `Relaxed`, `Acquire`, `Release`, `AcqRel` and `SeqCst` orderings; these are similar to the ones provided by C++ (post C++11), though the `consume` ordering is omitted.

Values

Rust values, declared using `let`, are immutable by default and must be declared as `mut` to mutate. New data-types are declared with `struct` or `enum` syntax (such as `LazySet` and `LookupResult` above). `structs` are values like in Go or C/C++, `enums` may contain arbitrary fields and hence are essentially variant types along the lines of ADTs in OCaml or Haskell. Enums and structs can be destructed through *pattern matching*, which is what the `match` experessions accomplish.

Another aspect of Rust that may throw off C, C++ or Java programmers is that Rust is more "expression-oriented", for example `return` statements are only necessary for early return semantics. In general, the entire body of a function is an expression: this is a function which adds one to a 32-bit signed integer:

```
fn add_one(i: i32) -> i32 { i + 1 }
```

Common control constructs form expressions as well, for example this is a function that only adds one to even numbers, otherwise it returns its argument:

```
fn add_one_even(i: i32) -> i32 {
  if i % 2 == 0 {
    i + 1
  } else {
    i
  }
}
```

Options

Rust pointers (types with prepended `&`) may not be null. To mark a value as optional, the `Option` enum is used. Here is an example definition:

```
enum Option<T> {
  Some(T),
  None
}
```

If the programmer is certain that an `Option` is indeed a `Some` variant, there is an `unwrap()` method with type `Option<T> -> T` that returns the contents of a `Some` variant and halts the program if the value is actually `None`.

Atomic Types

The Rust standard library provides a few basic atomic types. All of these types provide `compare_and_swap` methods which perform a CAS operation on the type, and either return a boolean indicating success or failure or return the previous value of the location in question. The provided types that we use are:

- `AtomicIsize`: a word-sized signed integer.
- `AtomicUsize`: a word-sized unsigned integer.
- `AtomicBool`: an atomic boolean value.

Note that the atomic numeric types also have a `fetch_add` method that adds a value to the atomic integer and returns the previous value of the integer.

`crossbeam`

We also use the atomic pointer type `Atomic<T>` provided by the `crossbeam` library and its EBMRS. `Atomic` provides a few CAS methods: `cas`, `cas_and_ref` and `cas_shared`. These operations all perform a CAS operation on the pointer, but they consume and return different references to the CAS'ed object in order to provide sufficient information to Rust's borrow checker.

For details on `crossbeam`'s EBMRS, see this blog post. The psuedo-code in this document leaves out the various `Guard` references that we create and pass to different functions. The one modification that we perform to the memory reclamation system itself is to add the `deferred_closure` method, which keeps a separate collection of closures that it runs before freeing the memory that has been enqueued by `unlinked`. Note that because `crossbeam` stores thread-local garbage lists, there are very few guarantees on the relative ordering of these closures being run, the only ordering that is guaranteed is that later epochs' closures run after earlier ones.

Appendix D: Further Optimizations

- *Stamps to avoid re-hashing.* The idea here is to use the values stored in a `Stamp` as a hash, thereby avoiding re-hashing all elements during a resize operation. We currently just mod to determine the has bucket, but store the deleted flag in the low-order bit, so it doesn't quite work yet, but it would be a small change.
- *SIMD instructions for lookups.* The algorithmic change here would be to store (perhaps truncated) stamps in a `Segment` in a separate array from the elements, essentially bifurcating `MarkedCell` into parallel arrays. These elements can then be loaded directly into SIMD registers and compared using a single instruction, using fast masking operations to determine which (if any) cells contain a potential match. This is the diciest one, as it decreases portability and packs more `Segment`s into a cache line.
- *Unboxed values.* The fact that nodes only become visible after the stamp is assigned, coupled with the fact that their values only change when the segment is unlinked and their values are freed, means that keys and values can be stored unboxed in a `MarkedCell`. This optimization will reduce allocations, the number of destructors enqueued on the memory system, and the number of indirections required even further. It will also space out stamps in memory, meaning fewer will share a cache line.

References

Arbel, Maya, and Hagit Attiya. 2014. "Concurrent Updates with RCU: Search Tree as an Example." In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, 196–205.

PODC '14. New York, NY, USA: ACM. doi:10.1145/2611462.2611471.

Dice, Dave, Maurice Herlihy, and Alex Kogan. 2016. "Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures." In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, 36–45. ISMM 2016. New York, NY, USA: ACM. doi:10.1145/2926697.2926699.

Fan, Bin, David G Andersen, and Michael Kaminsky. 2013. "MemC3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing." In *Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 371–84.

Feldman, Steven, Pierre LaBorde, and Damian Dechev. 2013. "Concurrent Multi-Level Arrays: Wait-Free Extensible Hash Maps." In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, 155–63. IEEE.

Herlihy, Maurice P, and Jeannette M Wing. 1990. "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (3). ACM: 463–92.

Herlihy, Maurice, and Nir Shavit. 2008. "The Art of Multiprocessor Programming." Morgan Kaufmann Publishers Inc.

Herlihy, Maurice, Nir Shavit, and Moran Tzafrir. 2008. "Hopscotch Hashing." In *International Symposium on Distributed Computing*, 350–64. Springer.

Howard, Philip William. 2012. "Extending Relativistic Programming to Multiple Writers." PhD thesis, Portland State University.

Michael, Maged M. 2004. "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects." *IEEE Transactions on Parallel and Distributed Systems* 15 (6). IEEE: 491–504.

Shalev, Ori, and Nir Shavit. 2006. "Split-Ordered Lists: Lock-Free Extensible Hash Tables." *Journal of the ACM (JACM)* 53 (3). ACM: 379–405.

Shun, Julian, and Guy E Blelloch. 2014. "Phase-Concurrent Hash Tables for Determinism." In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, 96–107. ACM.

Triplett, Josh, Paul E McKenney, and Jonathan Walpole. 2011. "Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming." In.

"What is RCU? RCU Linux Overview." 2014. https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt.