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.

Methods for Resolving Collisions

For a serial hash table, the key algorithmic challenge is the manner in which the table manages to resolve collisions. There are two general approacye

Something Something Herlihy and Shavit (2008).

Related Work

- Previous work on lock-free hash tables
  - Open-addressing work from 2000s
  - Split-ordered lists
  - Liu 2014 paper (also provides wait-free implementation)
- Faster (blocking) hash tables
  - CPHash
  - Cuckoo hashing
  - Hopscotch hashing
- RP work? “Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming”
- Phase-Concurrent Hash Tables?

These approaches all have different trade-offs. The first group satisfies strong consistency guarantee (linearizability), with strong theoretical progress guarantees: Neither readers nor writers will block, even during a resizing operation (where one is supported). However, these implementations tend to lag behind the performance of the rest of these implementations.

The second class of hash tables are fast, blocking hash tables that still provide linearizability. These tables provide the same consistency guarantees as the first group, but at the cost of progress guarantees; where some operations (usually `insert`s or `remove`s) must block.

The approach of relativistic programming is optimized to handle read-mostly workloads. The term “relativistic” is used to describe the fact that the causal ordering of events from the perspective of two different reading threads may be inverted at times. Hence, RP is able to achieve very high throughput for read-only workloads, but is not as suited to a setting with more than a small number of writers. RP is related to the “read-copy-update” (RCU) methodology used in (citations!+Linux?). Note that there are some data-structures that are linearizable that still employ RCU (todo: cite); the main contrast with the other groups is the use of a *wait-for-readers* operation to avoid read-size synchronization.

The work in this document combines techniques from the first and third group. It takes the goal of group 1 of providing fast *non-blocking* hash tables with a strong consistency guarantee: this hash table provides wait-free insert, remove and lookup operations while remaining linearizable. It does borrow methodology from the RP/RCU world; we leverage the notion of epochs to not only reclaim memory but also to perform resize operations. In contrast to approaches from the RP/RCU literature we never block writers (though resizing operations can be blocked on long-running reads and writes), and the data-structure is completely linearizable[1].

[1]The precise relationship between RCU, RP and linearizability appears somewhat murky to an outsider (such as

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).[2]

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[3].

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.

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.

the author of this document). There are some RCU data-structures that provide linearizability (e.g. "Concurrent Updates with RCU", by Arbel and Attiya TODO(cite), or some operations in the RP red-black tree thesis). However, RP (as opposed to merely RCU) seems to require some relaxation of linearizability (e.g. the thesis, or the RP hash table paper).

[2] Other solutions to this problem include Hazard Pointers (TODO: citation for hazard pointers and new Herlihy paper on slow-path hazard pointers), and writing all of the code in a language with garbage collection.

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

Notation and code listings

We provide Rust[4] psuedo-code[5] for the code listings.

TODO: expand on this, move to appendix, need to explain:

- Syntax
  - `Owned`
  - `Option`
  - Pointers in Rust.
  - Method notation in Rust
- Atomic operations
  - `load`
  - `store`
  - `fetch-add`
  - Not considering `Ordering`
- Overflow
  - The proofs below assume that a maximum of 2^{63} (or $2^{\text{word size}}$, depending on architecture) fetch-add instructions will be executed on shared `AtomicIsize` counters. See appendix A for 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.

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

[4] 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 minimimal runtime, giving it comparable performance characteristics to the C/C++ family of languages.

[5] 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 `Segment`s to avoid wasted allocations.

```
}
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` `MarkedCell`s, each pointers to actual values in the list. `MarkedCell`s 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)`.

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.

```
struct Stamp(AtomicUsize);
impl Stamp {
    //Note: in Rust `!` is bitwise complement.
    pub fn init(&self, v: usize) { self.0.store(v & ((!0) << 1)); }
    pub fn delete(&self) { self.0.store(self.0.load() | 1) }
    pub fn is_deleted(&self) -> bool { (self.0.load() & 1) == 1 }
    pub 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 (as in TODO Optimistic Cuckoo hashing paper). 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.

```
fn add(&self, key: K, val: Option<V>)  {
    let my_ind = self.head_ptr.fetch_add(1);
```

```
3      let cell = self.search_forward(my_ind)
4          .or_else(|| self.search_backward(my_ind))
5          .unwrap();
6      let h = hash(key);
7      cell.data.store(Some(Owned::new((key, val))));
8      cell.stamp.init(h); // (key, val) only become visible when stamp is set
9  }
10 fn search_forward(&self, ind: isize) -> Option<&MarkedCell<(K, Option<V>)>> {
11     let (seg_id, seg_ind) = split_index(ind);
12     while let Some(seg) = self.head.load() {
13         let cur_id = seg.id.load();
14         if cur_id == seg_id {
15             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` $m = s.\texttt{data}[i]$ *in some* `Segment` s *to be* $s.\texttt{id} \cdot \mathit{SEG\_SIZE} + i$.

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 adds 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));
```

```
15                  current_seg = &*current_seg.next.load().unwrap();}
16              let cell = current_seg.data[current_index];
17              cell.data.store(Some(Owned::new((k, Some(v)))));
18              cell.stamp.init(h);
19              current_index += 1;
20      }}
21      let mut seg_try = Owned::new(seg);
22      let mut cur = self.head.load().unwrap();
23      loop { // add seg_try to end of list. This normally runs once.
24          while let Some(next) = cur.next.load() { cur = next; }
25          match cur.next.cas_and_ref(None, seg_try) {
26              Ok(_) => { return; }
27              Err(seg1) => seg_try = seg1,
28  };}}
```

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 `LazySet`s, 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 (RP, split order? TODO).

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

Resizing

Add, Remove and Lookup

Resize

Wait-Freedom

Linearizability

- Implemented as an array of buckets! Lookups, Adds and Removes are linearized at their linearization points in the bucket implementation.
- Resizing operations are more difficult
  - Leverage epochs to determine when backfills are safe
  - Ensure that "slow" removes are the only ones that occur during a resize.
- Correctness, linearizability with a concurent resize operation

Performance

- 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

- New concurrent hash table that does not compromise speed for progress and consistency
- Should be possible to add transaction to this hash table, in that it can store all of the information necessary to reconstruct past versions of the table.

Appendix A: Overflowing bucket-level counters

It would take 68 years of fetch-adds at a rate of 2^{32} per second to overflow this counter.

Here are some (WIP) notes on why, under some even more modest assumptions than the one above, overflow is not a problem.

The only counters that are susceptible to any overflow issues are the bucket-level `head_ptr` counters. The basic argument is that one would need to trigger a single resize operation, and while descheduling enough other threads to stall a further resize, would need to discover 2^{63} *hash collisions* and add those. Note that such an attack, however implausible, is only feasible on hardware that can address 63 bits of space (this is not true of modern Intel hardware, even with 5-level paging).

But even if it could, one would require a domain where key size is at most two words, because we must store at least 2^{63} of them. If it was only a single word then a reasonable hash function would not allow for anywhere near 2^{63} collisions. This means that cache-padding elements of this data-structure mitigates this issue. Note that this is only necessary to consider if the value type has size 0. In fact we add a word to the value types in this implementation, so the attack does not appear to be possible.

Appendix B: Memory Ordering

Appendix C: Adding callbacks to the `Crossbeam` library

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

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