A Concurrent Chaining Hash Table

Wait-Free, Linearizable, Fast

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].

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.

[1] 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 (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.

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.

Notation and code listings

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

TODO: expand on this, need to explain:

- Syntax
  - `Owned`
  - `Option`
  - Pointers 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. Showing why this is reasonable takes longer, and is less important: it would take 68 years of fetch-adds at a rate of $2^3 2$ per second to overflow this counter.

  TODO(ezr) add to an appendix explaining why enabling growing mitigates the (remote) possibility of such an overflow. 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 de-scheduling enough other threads to stall a further resize, would need to discover $2^6 3$ *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 $2^6 3$ of them. If it was only a single word then a reasonable hash function would not allow for anywhere near $2^6 3$ collisions. This means that cache-padding elements of this data-structure mitigates this issue. Note that this is only necessary to consider if the

[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. 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.

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.

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<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)`.

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.

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);
    let cell = self.search_forward(my_ind)
        .or_else(|| self.search_backward(my_ind))
        .unwrap();
6      cell.data.store(Some(Owned::new((key, val))));
7  }
8  fn search_forward(&self, ind: isize) -> Option<&MarkedCell<(K, Option<V>)>> {
9      let (seg_id, seg_ind) = split_index(ind);
10     while let Some(seg) = self.head.load() {
11         let cur_id = seg.id.load();
12         if cur_id == seg_id {
13             return Some(seg.data[seg_ind]);
14         } else if seg_id < cur_id {
15             return None;
16         }
17         let new_seg = Owned::new(Segment::new(cur_id + 1, Some(seg)));
18         self.head.cas(Some(seg), Some(new_seg));
19     }
20 }
21 fn search_backward(&self, ind: isize) -> Option<&MarkedCell<(K, Option<V>)>> {
22     let (seg_id, seg_ind) = split_index(ind);
23     let mut cur = &self.head;
24     while let Some(seg) = cur.load() {
25         let cur_id = seg.id.load();
26         if cur_id == seg_id {
27             return Some(seg.data.get_unchecked(seg_ind));
28         }
29         cur = &seg.next;
30     }
31     None
32 }
33 fn split_index(ind: isize) -> (isize, usize) {
34     let seg = ind / SEG_SIZE;
35     let seg_ind = (ind as usize) % SEG_SIZE;
36     (seg, seg_ind)
37 }
```

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.

An `add` operation starts by acquiring an index into the `LazySet` using a fetch_add operation (line 2). It then searches for the cell corresponding to that index. In the common case, this will be an index into the current segment pointed to by `head`. There are, however, other cases to consider.

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) 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 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.

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. □

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. □

Removal

Lookups

Backfilling

Optimizations

- Decide on psuedo-code
- Detail add and both kinds of remove operation
- Describe “backfill” operation
- Show correctness, wait-freedom, linearizability of the add and remove operations.
    - Leave backfill to the full hash table description

The Hash Table

- 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.

References

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