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].

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

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.

[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.

Data Layout

- 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."