Design Document

Concurrent Hash Map with Hopscotch Hashing

Team members:
Daniel Perlmutter, Joaquín Ruales, and Wen-Hsiang Shaw

1. Overview

Our CHashMap implements, in addition to the functionality of a regular Hash Map, lock-free concurrency for adds and removals, and wait-free concurrency for lookups. To create this Hash Map, we used the concept of Hopscotch Hashing, which combines techniques from both Linear Probing and Cuckoo Hashing.

Each entry in the Hash Map is associated with a virtual bucket of a fixed neighborhood size. This allows, at the cost of acquiring locks on the virtual buckets during "add" and "remove" operations, for a simple implementation of wait-free lookup, or "get". Below, we take a closer look at these three primary methods we have mentioned. For further details and proofs of their expected runtimes, see Herlihy, Shavit, and Tzafrir (2008), referenced at the end of this document.

2. Description

i. add(key:K, value:V)

add() first gets the virtual bucket of the key that is being added and applies a lock not only there but to all of its 'neighborhood'. After acquiring the lock we add our item and remove the neighborhood's locks. There is one exception to this procedure when we acquire more locks: when we have to 'hop'. If we need to 'hop' (which occurs when there is no space in the current neighborhood; hopping is described in detail in the reference paper and involves moving elements around, while keeping their constraints, in order to make space for a new element), we first acquire additional locks beyond our neighborhood and up until the first empty bucket after our virtual bucket. We then perform the hopping, insert the new item, and release every lock we have acquired. Since add() has to search linearly for an empty space (starting at the virtual bucket), each insertion has a worst case O(n) run time, linear on the amount of elements already in the Hash Map. However, hopping is infrequent enough that add() runs in amortized O(1) time on average (proof provided in the referenced paper).

ii. remove(key:K):V

remove() first resolves the bucket corresponding to the key to remove, or its *virtual bucket*. Then, it tries to acquire this bucket's lock. Once it has acquired the lock, it linearly searches for the key of the item to be removed within the neighborhood of the virtual bucket. If it finds it, it sets the bucket's key to null, updates the virtual bucket's bitmap, releases the lock, and returns the value that was removed. If the item is not found during this search, remove() returns null. Since each remove can search through at most a number of elements equal to the neighborhood size, each remove takes time linear in the neighborhood size (and given that our neighborhood size remains fixed, it is constant time).

iii. get(key:K):V

get() simply looks for the requested key through every bucket in the neighborhood of the virtual bucket, and it returns its corresponding value. Since elements are only shifted by add(), and when that happens it can only shift them to the right—within their neighborhood and into an empty bucket—we are guaranteed to find an existing element within its neighborhood with a left-to-right scan. Thus there is no need to do any locking. If a remove() of the key occurs during a get() and the key is not found, it is because the remove() was linearized before the get(). Thus, each get() is a wait-free operation that takes time linear in the neighborhood size (again, given that our neighborhood size remains fixed, it is constant time).

3. Reference

1. *Hopscotch Hashing*, Maurice Herlihy, Nir Shavit, and Moran Tzafrir