N-Way Set Associative Cache Design

Varun Sukesan

Overview

This document is meant to detail the design and implementation decision for an N-Way Set Associative cache library. It will describe the basic functionality as well as the meeting of the following requirements.

Generally, the utility of a cache is to mitigate the sometimes high cost of retrieving data from the larger main memory. The cache is divided into sections or sets which correspond to larger sections of the main memory. “N” is the size of each set, and balances the speed of reading from the cache and the likelihood of getting a cache hit. This is because for each request for data from the cache, the entirety of the requested data’s designated set could be searched. If N is lower, that increases the speed of retrieving a value from the cache. It also means that requested data is less likely to be available, which generally would trigger a call to main memory.

This implementation is modified from the general case to fit the requirement.

Specific Requirements

1. The cache itself is entirely in memory (i.e. it does not communicate with the backing store).
2. The client interface should be type-safe for key sand values and allow for both keys and values to be of an arbitrary type (e.g. strings, integers, classes, etc.) For a given instance of a cache all keys must be the same type and al values must be the same type.
3. Design the interface as a library to be distributed to clients. Assume that the client doesn’t have source code to the library.
4. Provide LRU and MRU replacement algorithms.
5. Provide a way for any alternative replacement algorithm to be implemented by the client and used by the cache.

Implementation

Classes

NSetCache.java

CacheEntry.java

The NSetCache class is essentially an object-oriented data structure for key and value pairs. Functionality includes configurable initialization, get, put, remove, setEvictionAlgorithm, size, clear, and the option to override evictCustom. It contains objects of type CacheEntry, which is a wrapper class for the key and value pairs inserted into the cache by the user. CacheEntry also tracks a timestamp to be used for eviction algorithms.

Initialization

Since there is no backing store, the user is required to specify the total size of the cache and the size of each set (“N”) on initialization. If the user does not enter numbers that split the total cache size into a whole number of sets of size N, the cache size is scaled up to do so. I decided to manipulate the size of the cache instead of N because the size of the cache is variable anyway, and it seems logical that in an N-Way Set Associative cache, the user should be able to determine what N is.

Backend Structure

NSetCache uses an ArrayList of HashSets as a backend; I considered using a single array because the size of the cache is fixed on initialization, but decided that the cleaner implementation and increased speed in some cases of using existing data structures was a more effective solution. Specifically, a set is a logical choice because order is unimportant and duplicate entries are not allowed. Furthermore, the function to check if an entry is contained is utilized in multiple places and works in constant time in a HashSet.

Eviction

When a user attempts to put a key and value pair into a set that is already full, a single eviction based on the specified replacement algorithm is called. The cache will remove the Least Recently Used (LRU) entry by default, but the user can configure the cache to remove either the Most Recently Used (MRU) or a custom (CUSTOM) entry. This can be done at initialization or at any point after that via a set method. The time based eviction methods are based on the timestamp within CacheEntry, which is implemented with an atomic counter to handle inconsistencies with getting the current time within the computer system. The evictCustom method is included with the intention to be overridden. An algorithm to remove a random entry is implemented just as an example. The relevant set is passed in as a parameter, so writing the function is just a matter of iterating through the set and returning the entry to be evicted. The relevant data for each entry is available through public get methods in the CacheEntry class.