N-Way Set Associative Cache

Sameer Punjal

2/26/2018

Overview

This file serves as documentation for the accompanying implementation of an N-way set associative cache. Information regarding the design of the implementation and how it is to be used can be found here.

An N-way set associative cache is a cache data structure that combines the policies of a Directly Mapped cache with the policies of a Fully Associative cache. When a key value pair is added to to this cache, it is “directly mapped” to one of the N sets, and each of the N sets act as Fully Associative caches, which could incorporate an arbitrary replacement policy, like LRU, MRU, or any other.

Requirements

1. The cache should exist entirely within memory, without directly communicating with backing store
2. The cache should allow for keys and values of arbitrary types, with type safety enforced, so that all keys and all values must be of the same type
3. The implementation should be distributed as a library, and the end users should be able to use the library without access to the library’s source code
4. The implementation should include LRU and MRU replacement algorithms for the subcaches (the N sets)
5. The user should be able to provide an arbitrary replacement algorithms that they can implement themselves for the subcaches

Building and Usage

This project was written in Java version 1.8. It uses JUnit version 4.8.2 as a unit testing framework, and Maven version 3.5.2 to manage the project’s build. This project can be built using any Java IDE that integrates with Maven, or through the command line by issuing the command `$mvn clean install` in the `NWaySetCache` directory. This command will remove output from previous builds (as per the `clean` command), compile the source code and tests, produce a `.jar` file (located in the target directory), and run the unit tests located in `NWaySetCache/src/test/java/com/sameer/nwaysetcache`.

To include the cache in another Java program, include the import statement `import nwaysetcache.NWaySetCache;` or `import nwaysetcache.*;` Before compiling the program, build the NWaySetCache project using Maven, and add `NWaySetCache-1.0-SNAPSHOT.jar` (located in the `target` directory) to your project's classpath.

Public Methods

`public NWaySetCache(int n, int subcapacity)`

Constructor - initializes empty cache with `n` subcaches, each with a maximum capacity of `subcapacity`. Should be used as `new NWaySetCache<K,V>(n,subcapacity)` where `K` is the type of the keys, and `V` is the type of the values

`public int size()`

Returns the number of key value pairs currently stored in the cache

`public containsKey(K key)`

Returns true if `key` is contained in the cache and false if not

`public V get(K key)`

Returns the value associated with `key`. If `key` is not associated with a value, the return value will be null

`public void put(K key, V value)`

Inserts the key value pair (`key`,`value`) into the cache. Another key value pair may be evicted if necessary

`public void remove(K key)`

Removes the key value pair associated with `key`, if such a pair exists

`public void clear()`

Removes all key value pairs from the cache.

Using Alternative SubCache implementations

By default, this implementation implements the N sets as LRU caches. This policy can be replaced with the supplied MRU cache class or any user class that implements the Cache interface (found in the nwaysetcache folder) by using Java's Anonymous Inner Class feature to override the following method:

```
protected Cache<K,V> createSubCache(int subcapacity)
```

By default, this method returns a new instance of LRUCache with capacity `subcapacity`. When the constructor for `NWaySetCache` is invoked, createSubCache is called N times, and the returned Cache objects serve as the N subcaches. If `createSubCache` is to be overridden, it should return a new empty Cache object with capacity `subcapacity`. An instance of `NWaySetCache` can be instantiated using the supplied MRUCache class instead as shown below:

```
NWaySetCache<Integer,String> ex =

new NWaySetCache<Integer,String>(4,16){

@Override
protected Cache<Integer,String> createSubCache(int subcapacity){
     return new MRUCache<Integer,String>(subcapacity);
}

};
```

In the code above, the variable `ex` is assigned to a new instance of `NWaySetCache` with 4 sets, each of which has a maximum capacity of 16. The new `createSubCache` method now returns instances of MRUCache with a maximum capacity of `subcapacity`.

Implementation Details

LRUCache

The LRUCache implemented by maintaining a LinkedHashMap whose internal Linked List was set to maintain access order (this is not the default for Java's LinkedHashMap, but it can be set in the constructor). The removeEldestEntry method was overridden (using Anonymous Inner Class) to ensure that when the capacity of the LRUCache is exceeded, the head key value pair of the internal Linked List would be removed from the map.

MRUCache

Although MRU caches and LRU caches are similar in design to each other, the MRU cache could not be simply adapted from the LinkedHashMap class in a way that maintained that each of the operations could be performed in O(1) time due to the fact that it is impossible to get access to the most recently accessed key in Java's LinkedHashMap class in O(1) time.

The MRUCache class was implemented internally as a linked hashmap. This reimplementation of the linked hashmap would allow for access to the most recently accessed key in constant time. Internally, there exists a plain HashMap that maps keys to doubly linked list nodes that store the key value pair, along with references to the predecessors and successors in access order. To simplify the implementation there exist two dummy list nodes to represent the head and tail of the list as well - these nodes are not mapped to by the hashmap.

Any time an access or assignment is made in the cache, the corresponding key value pair node is placed in the list just prior to the tail. When the cache has reached its capacity and a new element is added, the element prior to the tail is deleted from the list and its key is removed from the map before the new key value pair is added.

To remove a key value pair from the cache, the corresponding key's list node is removed from the linked list and its key is removed from the map. When a key is overwritten to a new value, the key is first removed from the cache, then added as normal. The internal linked list exists to support the remove functionality - if the remove operation was not to be supported, a standard hashmap and a variable that tracked the most recently used element would suffice.

NWaySetCache

The NWaySetCache maintains an ArrayList of N Caches (implemented as LRUCaches by default). A `key value` pair is added to the NWaySetCache by placing the pair in the subcache with index `key.hashCode()` modulo N. Assuming evenly distributed `hashCode()` values, it is likely that the key value pairs will be evenly distributed through the N subcaches. All of the major cache functionalities are handled through the functionalities of the subcaches - when a put, removal, or a check for a key is made, the corresponding check is made for the correct subcache for the input key.

The `createSubCache` function and the `Cache` interface were both included in the design to allow for it to be users to incorporate their own caching data structures in to this design. For a user to implement their own caching policy into the NWaySetCache, they would need to create an implementation of the Cache interface, which requires only some basic functionality to be included.