N-Way Set Associative Cache Design Document

Requirements

1. The cache is in-memory.
2. The client interface is type-safe for keys and values, for a given instance, all keys and values must be the same respective type.
3. The interface is a library for clients who may not have access to the source code
4. Provide LRU and MRU replacement algorithms.
5. Provide a way for the user to create an alternative replacement algorithm that can be used for the cache.
6. The cache will have n sets and in each set are m blocks, n and m are set by the user.
7. The replacement algorithm for a cache spans a set.
8. The values for n and m as well as the replacement algorithm may not be changed after constructing the cache.

Design

The client facing interface of the n-way set associative cache is an implementation of a Java templated interface called `Cache<K, V>`. The cache implementation contains an `ArrayList` of sets. The cache behaves like a pass-through layer and only deals with finding the correct set for the key as well as interpreting the cache set's return values. As a result, the sets that the cache uses has very similar signatures when compared to the outer cache.

Outer Cache

The outer cache uses an array list to hold the inner sets. But given any Key, has O(1) access to the correct set that would contain the key. This is due to the cache explicitly hashing a given key and performing a modulus on the value to obtain the index of the set for the key. `ArrayList` access to a given index is O(1).
The `NWaySetAssociativeCache` can be customized with any implementation of `HashFunction` as well as any implementation of a list of `CacheSets`.
There are two builder classes for the `NWaySetAssociativeCache`:

1. `Builder` which allows the user to quickly build an instance of `NWaySetAssociativeCache` using a selection of the packaged Replacement Algorithms and Cache Set from the enumeration classes.
2. `AdvancedBuilder` which allows the user to pass in a user defined implementation of the `HashFunction` as well as a user implementation of the list of `CacheSet`, which can have any combination of replacement algorithms and entry types. Note that the `List` implementation is also not necessarily an `ArrayList`.

The user can also extend the `NWaySetAssociativeCache` and leverage the protected functions such as `onCacheHit()` to augment the reporting capabilities of the cache.

Inner Set

The inner sets have functions matching closely to the outer cache. These functions match one to one with the outer cache and are called by the outer cache's functions of the same name. The default implementation of a set is the `HashMapCacheSet`, though by following the JavaDoc of the `cacheSet` class, any implementation can be used by the main outer cache. The `HashMapCacheSet` uses a Java `HashMap` to maintain the key value pairs as well as have O(1) access for any key, just like the outer cache. It uses Java's `ReentrantReadWriteLock` to help with concurrency, following a real database cache locking behavior. The read lock is used for methods `get()` and `containsKey()` because they do not have the cache change the entry data while the write lock is used for `clear()`, `remove()`, and `put()` since they do use the cache to change entry information.

Entries

The entry class `CacheBlockEntry` maintains metadata about the entry such as the access time or whether the information has been modified. It is somewhat coupled with the `HashMapCacheSet` in order to give the user more flexibility in what information to modify in the entry should they choose to extend from either class and leverage the protected methods.

Replacement Algorithms

The cache's replacement algorithm should be set at creation time and the `HashMapCacheSet` does not allow an algorithm to be reset mid way, though it is an option if the user implements it that way. The `ReplacementAlgorithm` interface tries to be flexible in providing different ways for the user to implement a custom algorithm. The `evict()` method is giving the algorithm all of the entries in the set and expecting the algorithm to remove one. The two `notify()` methods may be implemented if the user wishes an to follow an observer style pattern.

The basic MRU and LRU algorithms do not use the observer methods and simply run through the collection of entries and removes the one that meets the criteria. This means they are dependent on the entries' metadata. They also have a runtime of O(N), N being the number of entries given. This could impact performance negatively if there are many evictions happening. The cache would attempt to operate at O(1), but `put()` calls may slow down to O(N).

The Observer versions of MRU and LRU maintain their custom linked hash map called a `HashQueue`, where each time a change in the cache occurs, they modify their own data structures to ensure the next to be evicted is always ready at the next call of `evict()`, note that the argument is never used. Since the custom data structure performs in O(1), evictions happen at O(1). The trade off is that space usage increases due to the extra data structure.

Though the `HashMapCacheSet` does not support replacing a replacement algorithm randomly during runtime, the basic MRU and LRU algorithms are safe to switch out, whereas the Observer versions will not work when switched out randomly.

Hash Queue

The specialized data structure is a combination of a Java `HashMap` and linked list with an interface like a deque. The `HashMap` keys are the keys of the cache, and the values are the nodes of the list. Removing a node representing a given k or moving it to the front/back happens at O(1) due to the instant access of the node through the map.

Testing

Aside from basic unit and integration tests that check for branching as well as regression, I implemented performance tests to measure the true performance of the various replacement algorithms under real, high stress situations. In the package integrationTest.concurrency, I created builders that would spawn threads that made repetitive get/containsKey/put/remove calls in parallel and built an instance of a cache for each of the four replacement algorithms and put them to the test and measured their average speed per method call (same cache implementation besides the algorithms).
My goal was to perform different tests with different cache sizes and algorithms as well as different access order to perform some sort of statistical analysis.

Unfortunately my computer is 7 years old and has 1 CPU so the results were unexplainable magic. Sometimes the Observer replacement algorithms performed worse than the Basic algorithms by a factor of 10 and sometimes the other way around. Sometimes, for the same algorithm, the `get()` calls would take twice as long as the `containsKey()` calls.
I have encapsulated some of those performance tests in the PerformanceTest.java class in case someone with a btter machine wanted to confirm the results.

Build

This project is build using Gradle and comes along with the Gradle wrapper. All of the dependencies are handled by Gradle connecting to the Maven repository. To build a .jar file for this library, navigate to the root of this project and run

`gradlew build` on Linux or `gradlew.bat build` on Windows

The unit tests are automatically run on build, though the jar alone can be built with the argument `jar` instead of `build`. The cache.jar library is located in build/libs/ while the test reports are located in build/reports/tests/test/index.html, which should be opened on a browser.
The integration tests are run with:

`gradlew integrationTest` on Linux or `gradlew.bat integrationTest` on Windows

and the results can be seen atbuild/reports/tests/integrationTest/index.html. The PerformanceTest class will have output representing the average speeds if you navigate to that section (build/reports/tests/integrationTest/classes/concurrency.PerformanceTest.html) though I do recommend opening

the project and running the performance tests in Eclipse to allow instant changes of the cache variables.

`gradlew clean` on Linux or `gradlew.bat clean` on Windows

Will clear out all built jars as well as test reports.

Usage

To use the cache in real code, import all of the library classes and declare a `Builder` or `AdvancedBuilder` with the Key and Value templates set and follow the procedures. Note that the `Builder` comes with default values for the members so the user does not have to set them all if they are confident enough.

Basic:

```
NWaySetAssociativeCache.Builder<String, String> builder =
        new NWaySetAssociativeCache.Builder<>();
builder.setTotalSets(10)
    .setSubCacheType(SubCacheType.HASH_MAP_CACHE_SET)
    .setBlockSize(5)
    .setReplacementAlgorithmType(ReplacementAlgorithmType.LRU)
    .setHashFunction(new NullSafeHashFunction());

Cache<String, String> cache = builder.build();
```

Advanced:

```
NWaySetAssociativeCache.AdvancedBuilder<String, String> builder =
        new NWaySetAssociativeCache.AdvancedBuilder<>();
List<CacheSet<String, String>> cacheSets = new ArrayList<>();
cacheSets.add(set0);
cacheSets.add(set1);
builder.setHashFunction(new NullSafeHashFunction()).setCacheSet(cacheSet);
Cache<String, String> cache = builder.build();
```

More examples can be seen in the test classes.