N-way Set Associative Cache Design

1 Overview

Objective of this document is to provide an overview of a java library implementing N-way set associative cache, which allow developers to help reduce the average time to access a memory for given software.

In N-way set associative cache, each set contains N entries. For every R/W operation, all the entries of a given set are scanned. In general, there is always a trade-off achieving fastest hit times against lowest miss rates. N is a control factor on these statistics, for example, in case of direct mapped cache, we obtain best hit times, while higher the value of N, lower we see miss rates.

2 Requirements

1. Cache itself is entirely in memory. No Communication to backing store or I/O.
2. Lib should be type-safe for key and value.
3. Clients should be able to r/w from cache using arbitrary keys. (e.g. strings, integers, classes, etc.)
4. Design should be switchable between LRU & MRU replacement algorithms.
5. Distributed external library.
6. Client flexibility to implement homemade replacement algorithm.

3 Solution Design

N-way set associative cache is very similar to direct mapped cache, in which we hash every key and mapped to a particular set, instead of having single entry per set, in this we have N entries per set. For example in 4-way associative cache, each set has 4 entries. (Refer to 4. Implementation section for more details.)

In case of miss, new data is loaded to cache entries. In case of fully occupied cache entries, single eviction is performed to make room for new entry. By default LRU replacement algorithm is used for this eviction. Replacement algorithm is user configurable. Library also provides MRU implementation for eviction purpose, which could be explicitly selected. Client is also allowed to use completely new custom-made replacement algorithm (Please Refer to 4. Implementation section for more details.)

4 Implementation

This section describes in depth implementation details. Client is allowed to pick `N` and `Number of Sets` of their choice. `N` defines number of Entries per set, while `Number of Sets` defines total number of sets given cache has. This 2 metrics helps evaluate final size of a cache.

```
Size of Cache = N * Number of Sets
```

For unspecified N cases, N=1 used by default. In short direct mapped cache is used by default. (Refer to 7. Cookbook to understand configurability with an example.)

4.1 Scan Policies

This is Array based N-way set associative cache implementation. Each request (r/w) comes along with the arbitrary key, which is mapped to a hashed integer value. (Refer to 4.2 Hashing Policies for more details) This integer key is used for locating `Start Index` of designated set inside a given cache for r/w request.

Depending on value of `N` used, next step is to determine `Index Set` for a given key. `Index Set` is a range of indexes, which belongs to a particular key. Basically, this is number of entries to be scanned for each set. (Which is N)

```
Index Set = {Start Index, End Index}

Where,
Start Index = (IntKey % numSets) * numEntries
End Index = Start Index + N - 1;
```

4.1.1 Read

In case of read operation for a given key, `Index Set` is scanned for key match, if hit, data returned. Also, timestamp field is updated for that cached entry representing recent usage of that entry.

While in case of miss, ideally data is required to be retrieved from main memory as well cache is updated, given the restriction as per requirements no backing store or I/O communication, null is returned. Also, new data is loaded in cache based on Replacement Policies. Dummy implementation is provided for future purpose.

4.1.2 Write

In case of write operation for a given key, `Index Set` is scanned for key match, if hit, data location is updated with new updated data along with timestamp update representing recent usage of cache entry.
In case if miss, Index set is scanned for empty location, first found location within Index Set is used for storing new data.

In case of fully occupied cache entries for a given set, Replacement Policies are used to evict single entry to make room for new one.

4.2 Hashing Policies

Default MD5 uses hashing algorithm for generating unique hash based on unique arbitrary key. This is pretty much essential function which defines/maps every individual key to specific set in a cache. Hashing algorithm is also configurable. User can switch to custom-made hashing if needed. (Refer to 7. Cookbook to learn the example)

4.3 Replacement Policies

In order to make room for new entry in case fully occupied cache entries, one entry needs to be evicted in order to make room for new entry, by default LRU algorithm is used to determine least recently accessed entry location out of `Index Set` to replace with new entry. This identification is obtained based timestamp comparison for every entry in a given set. (Also refer to 4.3 to learn Cache Entry Structure) Alternatively, MRU algorithm is also provided for users, which is configurable. Users are also allowed to provide their own custom-made replacement algorithm for performing eviction. (Refer to 7. Cookbook for detailed example.)

4.4 Cache Entry Structure

Cache Entry has the following structure.

Tag	Data	isEmpty	Timestamp

Description for each individual fields in a structure.

Tag:	This contains part of address. Hashed Integer Key.
Data:	This contains actual data.
IsEmpty:	Under flush condition its true for all entries, representing emptiness. Set to false is location occupied.
Timestamp:	This is field is updated at every time cache entry is accessed for r/w.

5 Deliverables

5.1 Packages

NSetCache-1.0.0.jar

6 API

6.1 Javadoc

Please refer to NSetCache-javadoc.zip package. This contains html version of javadoc.

After downloading source code, javadoc could also be built by following,

```
cd $WORKING_DIR
mvn javadoc:javadoc
```

Javadoc will now be available under,
`$WORKING_DIR/target/site/apidocs/index.html`

7 Cookbook

This cookbook describes how to consume N-way set associative cache using new java NSetCache library. The purpose of this library is to make configurable N-way set associative cache available to developers, further in this cookbook we will discuss steps involved in consuming this cache using NSetCache library.

7.1 Build

This is a maven based java project. After downloading project, in order to build the project, do following,

```
cd $WORKING_DIR
mvn clean install
```

7.2 Setup/Installation

Once project is successfully build you should obtain jar file or this is also package is also provided.

You can locate the built jar under `$WORKING_DIR/target/NSetCache-1.0.0.jar`

7.3 Getting Started

For writing maven based java projects most developer prefer to work in a IDE like Netbeans or Eclipse rather then testing by direct deployment due to the improved efficiency it brings.

Lets start writing custom N-way set associative cache using NSetCache Library.

7.3.1 Maven Configuration file (pom.xml)

We will use maven for handling compiling and building of a the bundle, and it requires single configuration file as well. Store this file project home DIR. (referred as $WORKING_DIR)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany</groupId>
    <artifactId>ClientCache</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>ClientCache</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.mycompany</groupId>
            <artifactId>NSetCache</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>5.8</version>
            <classifier>jdk15</classifier>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-javadoc-plugin</artifactId>
                <version>2.8.1</version>
            </plugin>
        </plugins>
    </reporting>
</project>
```

We will not spend much time in understanding whole configuration here, but note that 2nd <dependency> imported is the one we are interested in. This import will make sure, NSetCache Library is available for java project.

Testng artifacts are imported for unit tests purpose, which will discuss further in the cookbook.

7.3.2 Writing N-way set associative cache

Here is the example of complete cache.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.mycompany.clientcache;

import com.mycompany.nsetcache.NSetCache;

/**
 * Sample client cache implementation based on NSetCache library.
 *
 * @author panchal
 */
public class ClientCache extends NSetCache {

    /**
     * Constructor. Initializes N-way associative cache
     * of numSet*numEntry size.
     * where, N = numEntry
     * with CUSTOM  replacement algorithm.
     *
     * @param numSet
     * @param numEntry
     * @param replacementAlgo
     */
    public ClientCache(int numSet, int numEntry,
                       String replacementAlgo) {
        super(numSet, numEntry, replacementAlgo);
    }

    /**
     * Custom hash algorithm written for demo.
     * This uses standard java Object hashcode() function for hashing.
     *
     * @param key Key.
     * @return Hashed integer Key.
     */

    @Override
    public int getHash(Object key) {
        return Math.abs(key.hashCode());
    }

    /**
     * Custom replacement algorithm written for demo.
     * Simple algorithm, evicted index is always last index of
     * Index Set.
     *
     * @param startIndex Start index.
     * @param endIndex End index.
     * @return Index to be evicted.
     */
    @Override
    public int customReplacementAlgo(int startIndex, int endIndex) {
        return endIndex;
    }
}
```

`ClientCache` Object now has access to all basic cache functionality and methods like `get(key), put(key, value), size(), clear()`.

Note that in above class is using constructor which allows replacement algorithm to be overwritten. If you want to customize replacement alogorithm, simply override `customReplacementAlgo` method. As seen above very basic custom algo is written which evicts last index no matter what as example. Once, done `ClientCache` object could be initialized with "CUSTOM" passed as value for `replacementAlgo` parameter to the constructor. Refer to unit test in next section for example.

Similarly, if you want to customize/change underneath hashing algorithm, override `getHash` method as shown above. For example purpose, I have just used standard Object `hashcode()` method as an alternative to default MD5 hash algorithm.

7.3.3 Unit testing N-way set associative cache

Below I am, documenting few basic unit tests using testing framework for understanding purpose.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.mycompany.clientcache;

import org.testng.Assert;
import org.testng.annotations.Test;

/**
 * Unit testing class for ClientCache.
 *
 * @author panchal
 */
public class ClientCacheTest {

    /**
     * Test for basic read/write operation
     */
    @Test
    public void testBasicRW() {
        ClientCache cache = new ClientCache(8, 2, "LRU");
        cache.put("Lionel", "Messi");
        cache.put("Christiano", "Ronaldo");
        Assert.assertEquals(cache.get("Lionel"), "Messi");
        Assert.assertEquals(cache.get("Christiano"), "Ronaldo");
    }

    /**
     * Testing CUSTOM eviction algorithm, making sure, last index entry
     * is evicted in case of fully occupied entries of a given set.
     *
     * @throws InterruptedException
     */
    @Test
    public void testCUSTOM() throws InterruptedException {
        ClientCache cache = new ClientCache(8, 4, "CUSTOM");
        cache.put(16, "Christiano");
        Thread.sleep(50);
        cache.put(32, "Ronaldo");
        Thread.sleep(50);
        cache.put(48, "Lionel");
        Thread.sleep(50);
        cache.put(64, "Messi");
        Thread.sleep(50);
        cache.put(80, "Kaka");
        Assert.assertEquals(cache.get(16), "Christiano");
        Assert.assertEquals(cache.get(32), "Ronaldo");
        Assert.assertEquals(cache.get(48), "Lionel");
        Assert.assertEquals(cache.get(64), null);
        Assert.assertEquals(cache.get(80), "Kaka");
    }
}
```

This test cases demonstrates how configurable N-set associative cache could be created programmatically & tested for client usage purpose.

8 Dependencies

No 3rd party dependencies except for standard java deps.

9 Review/Comment