N-Way Set Associative Cache

Technical Design Document

issue 2

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PURPOSE

The purpose of this document is to outline and detail the technical design of the N-Way Set Associative Cache and provide an overview for the N-Way Set Associative Cache implementation.

Its main purpose is to -

- Provide the link between the Functional Specification and the detailed Technical Design and Implementation.
- Detail the functionality which will be provided by each component or group of components and show how the various components interact in the design
- Provide a basis for the N-Way Set Associative Cache development
- Provide installation and configuration details of the actual implementation.

As is true with any design document, this document will be updated and refined based on changing requirements.

The document is intended to be used within the project team for reviews and discussions for futher updates and enhancements.

1.2 SCOPE

The cache design outlined in this document builds upon the scope defined in the Coding Exercise document. Which were:

#1 The cache itself is entirely in the memory (ie., it does not communicate with a backing store)

#2 The client interface should be type-safe for keys and values and allow for both keys and values to be of an arbitrary type. For a given instance of a cache, all keys must be of the same type and all values must be of the same type.

#3 Design the interface as a library to be distributed to the clients. Assume that the client does not have the source code for 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.

1.3 SPECIFIC DESIGN CONSIDERATIONS

#1 Java has been used as the language for development, primarily due to programmer's own preferrence and familiarity with the language and it's environment.

#2 The implementation is pure java and has no dependencies on any third party java or non-java libraries and APIs. This approach has been choosen to obliterate the burden of maintaining and updating the third party library changes.

#3 The project uses maven 3.3.9 build system which is simple and open-source and provides easy version and dependency management along with easy integration with unit and integration test libraries.

#4 Java property file has been used as cache configuration mechanism due to its pure Java nature and non-dependency on any library for loading and parsing.

2 SYSTEM OVERVIEW

N-Way Set Associative cache is a cache scheme that is primarily used to design L1 caches in the CPU cache memory hierarchy. It is a hybrid between a fully associative cache and direct mapped cache. It's considered a reasonable compromise between the complex hardware needed for fully associative caches (which requires parallel searches of all slots), and the simplistic direct-mapped scheme, which may cause collisions of addresses to the same slot (similar to collisions in a hash table).

In this cache scheme we group slots into sets. We find the appropriate set for a given address (which is like the direct mapped scheme), and within the set we find the appropriate slot (which is like the fully associative scheme).

This scheme has fewer collisions because we have more slots to pick from, even when cache lines map to the same set.

2.1 SYSTEM ARCHITECTURE

Since this implementation is a constrained version of a cache and is NOT a full-blown cache, the following architecture decisions have been made to make the implementation traceable with the requirements:

#1 The cache is fixed-sized for the life-time of the instance. Once instanciated, it can't be resized. The cache size and number of slots per set as confugured using attributes in cache.properties file.

#2 The cache provides implementations for LRU and MRU cache replacement algorithms / policies which can be opted by setting the chosen class as the value of cache.evictionPolicy.implClass property in cache.properties file.

#3 The cache replacement policy is set at the initialization of the cache instance. The value is read from the cache.evictionPolicy.implClass property of the cache.properties file. If no value is defined for this property, cache uses LRU as default replacement policy.

#4 Client can define their own cache replacement policies by implementing the EvictionPolicy interface and can use them by placing the implementation class on the CLASSPATH and setting the value of cache.evictionPolicy.implClass property of the cache.properties file with the implementation class.

#5 The implementation is stand-alone and non fault-tolerant.

#6 The cache provides easy java property class based confiuration.

#7 The cache provides a very basic statistics feature which can be used for cache performance monitoring.

#8 Since the cache is supposed to be purely in memory and should not talk to a backing store, this implementation does not cover cache write policies for a cache-miss scenario.

#9 The implementation is available in the form of a pure Java API / library which exposes simple cache methods like put / get / remove / clear methods through interface.

2.2 SOFTWARE DEVELOPMENT TOOLS

Software and development tools used for this project were:

1 Eclipse Mars IDE for Java;

2 Oracle JDK 1.8

3 Maven 3.3.9;

4 Junit 4.12

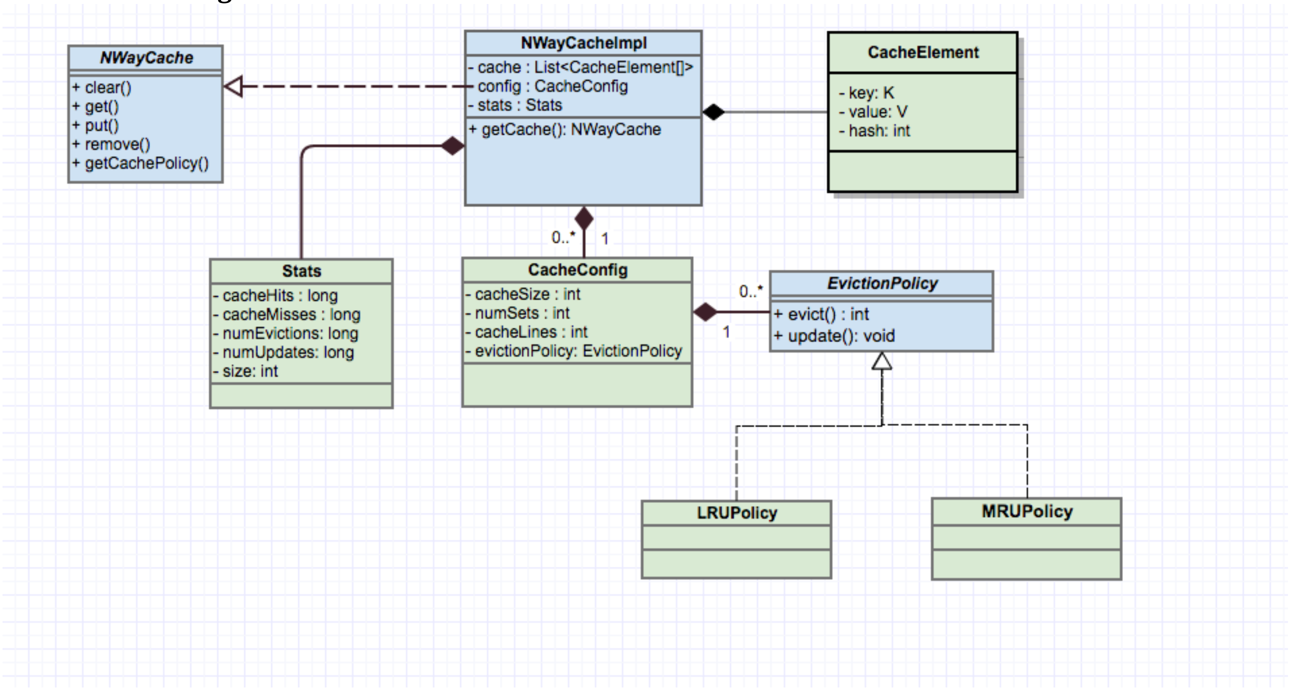
5 LibreOffice Version: 4.3.1.2 for documentation

6 draw.io for UML

3 SYSTEM DESIGN

3.1 DECOMPOSITION DESCRIPTION

3.1.1 Class Diagram



3.2 IMPLEMENTATION

3.2.1 Component Details

#1 **Data Structure** – a List of arrays has been chosen to contain the cache elements because:

1. Each element of the list represents a set of M size.
2. This structure is banked or slotted into separate array objects which can be locked separately by different threads and so, accessed simultaneously in a thread-safe implementation providing fine-grained concurrency.
3. Elements of a set are arrays because a N-Way Set Associative cache has usually a cache line size or set size between 2 – 8 (because bigger value of N is counter productive), there is not much penalty in sequential search among 2 – 8 elements.

#2 **<u>Configuration</u>** – Configuration is available through **cache.properties** file and **CacheConfig** class. The cache.properties class is read during initialization and an object of CacheConfig class is created for use through-out the life-cycle of the cache. Following attributes are available for configuration:

1. **cache.lineSize** – Defines the number of slots per set. Should be a multiple of 2.
2. **cache.size** – Defines the total number of objects in the cache. The value should be a multiple of cache.lineSize
3. **cache.evictionPolicy.implClass** – Defines the class implementing cache replacement algorithm. This class is loaded at cache initialization process using reflection.

#3 **<u>Cache client interface</u>** – Cache client has following interface available for interaction with the cache:

1. **NWayCache** is the interface that exposes the cache operation to the client. These operations are:
    1. **get(K key)** – fetches the value of type V from cache if a value is mapped for key. If there is no value mapped, null is returned.
    2. **put(K key, V value)** – puts a key-value pair in the cache. If there are no stots available, eviction policy will be used to determine the victim element to make space available for the new element.
    3. **clear() -** clears the cache of all elements.
    4. **remove(K key)** – removes the element mapped with the key. If the key has a mapping, the mapped element will be removed and the method will return true. If a mapping is not found, method will return false.
    5. **getCachePolicy() -** Returns the EvictionPolicy class for the current instance of Cache.
2. **EvictionPolicy** interface allows client to implement its own customized cache replacement policy. Ideally the client can just extend the AbstractPolicy policy and override the evict() method to define their own algorithm.

#4 **<u>Cache data element</u>** – **CacheElement** class defines an element for the cache. It has following attributes and methods:

1. **key** – represents the key of the cache.
2. **value -** represents the value mapped with key.
3. **hash** – contains the computed hashCode of the key so that hashCode computation is not required repeatedly during cache operations.
4. **hashCode() -** rturns the hashCode of the key.

#5 **<u>Cache implementation class</u> –** NwayCacheImpl class implements the cache. It has following attributes and methods:

1. **cache** – A List of arrays of objects that represents the cache storage.
2. **config** – this is an object of CacheConfig class. This class is created during cache initialization from cache.properties class.
3. **stats** – an object of the class Stats. This is used for collecting and displaying the general statistics of the cache.

4. **getCache() -** this is a static factory method for creating an instance of the cache. It reads the cache.properties file and creates a cache as per configuration provided.

5. Implementations for put(), get(), remove(), getCachePolicy(), and clear() methods.

#6 **<u>Eviction policy algorithm</u> –** Following two classes implement the replacement algorithms:

1. **LRUPolicy** – Implement the LRU replacement algorithm.

2. **MRUPolicy** – Implements the MRU replacement algorithm.

**Client can implement their own cache replecement algorithms by implementing EvictionPolicy interface.

3.2.2 Algorithm

1. **<u>Cache organization</u>**: The N-Way set associative cache is organized as N sets of M slots, where N is a multiple of M and M is a multiple of 2, ideally in the range of 2 – 8, because having more than 4 stots degrades the cache performance and having 8 stots per set has almost similar performance as fully associative caches. M > 8 is detrimental to performance.

The cache size is a product of M and N, that is, Total number of element in cache (size) = M * N.

The index lookup for a tag or key is performed in the following manner:

1.1. Take the key hash (also known as cache entry tag) and calculate hash % N – This gives the set number X.

1.2. cache.get(X) – This gives the starting index of the set number X in the cache.

1.3. Key hash (tag) of each slot in the set needs to be compared and key equality needs to be confirmed to find the index of the element to be inserted, fetched or removed / replaced.

1.4. If the key is not found in the cache, return -1.

2. **<u>get() operation</u>**: The get() operation works as follows:

2.1. Find the element index as described in bullet 1.

2.2. if the index returned was -1, return null.

2.3. If the index is a non-negative value, return the value of the element at index.

2.4. Updated the element and cache statistics.

3. **<u>put() operation</u>**: The put() operation works as follows:

3.1. Compute the set number for the key and the start index for the set.

3.2. Iterate through all the stots in the set and find an index where the slot is empty or the hash of the key matches the hash of the element in the slot.

3.3. If a slot was found, set the element in the slot and update the cache statistics.

3.4. If a slot was not found,

   1. invoke the evict() method of the replacement policy object of the cache to find the index of the victim element.

    2. Put the new element at the index.
    3. Update the cache statistics.

4. **<u>remove() operation</u>**: remove() works in the following way:

    4.1. Find the element index as described in bullet 1.

    4.2. if the index returned was -1, return false.

    4.3. If the index is a non-negative value, remove the element at index.

    4.4. Updated the cache statistics.

    4.5. Return true.