N-Way Set-Associative Cache

10th October 2017

OVERVIEW

This document presents an overview of a .NET library which provides an implementation of n-way set-associative cache.

A n-way set-associative scheme is a hybrid between a fully associative cache and direct mapped cache. It's considered a reasonable compromise between the complex 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).

GOALS

Create a cache library regarding the following specifications:

1. The cache itself is entirely in memory (i.e. it does not communicate with a backing store)
2. The client interface should be type-safe for keys and values and allow for both the 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 all 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 your library.
4. Provide LRU and MRU replacement algorithm
5. Provide a way for any alternative replacement algorithm to be implemented by the client and used by the cache.

SOLUTION DESIGN

For the purpose of this document, we will define a n-way set-associative schema as follows:

set_1: (way_1 [tag, value]) **…** (way_n [tag, value])

……..

set_k: (way_1 [tag, value]) **…** (way_n [tag, value])

Adding elements to the cache:

The element will be mapped directly to a set and associatively to a way in the chosen set. To be able to do this selection for an arbitrary key type, we will first calculate the hash of the key, using the 'GetHashCode' method in .NET, which will provide us with an integer value.

We will then consider the int value as an array of bits of form **[tag_bits | set_bits],** taking a number of **log(k, 2)** bits from the hash to compute the set index, and storing the value along with the tag(the remaining bits of the key) in the first available way of the set. If no way is available in the chosen set, one of them will be freed up with regard to the eviction policy.

Fetching elements from cache:

Retrieving values from cache works in a similar manner as the adding mechanism described above, a set will be selected based on the last bits of the key hash, the remaining tag bits being then matched to the stored elements in the set. If a way which stores the same tag is found, the corresponding value will be returned, otherwise the method will return the default value

SPECIFICATIONS

SetAssociativeCache API:

Constructors:

/// Initializes a new instance of SetAssociativeCache with LRU default replacement strategy

SetAssociativeCache(int **numberOfSets**, int **numberOfWays**)

/// Initializes a new instance of SetAssociativeCache taking an associative cache factory as parameter

SetAssociativeCache(int **numberOfSets**, int **numberOfWays**,

Func<int, IAssociativeCache<TValue>> **associativeCacheFactory**))

Methods:

/// Adds element to cache.

void **Add**(TKey key, TValue value);

/// Gets element from cache.

TValue **Get**(TKey key);

/// Counts the elements in the cache.

int **Count** { get; }

/// Flushes the cache.

void **Clear**();

Built in associative cache implementations: LRUAssociativeCache(default), MRUAssociativeCache, RandomEvictionAssociativeCache.

A **custom eviction policy** can be enforced passing an implementation of the **IAssociativeCache** interface to the constructor through a factory function**(1)** or by passing an instance of **IEvictionPolicy** implementation**(2)**. Examples:

1. new **SetAssociativeCache**<int, int>(2, 4, (size) => new **CustomAssociativeCache**<int>(size)))

2. new **SetAssociativeCache**<int, int>(2, 4, new **CustomEvictionPolicy**<int>());