**Muhammad Akhtar Shiekh**
akhhttar@gmail.com

# N-Way Set-Associative Cache

**20th March 2018**

## OVERVIEW

Purpose of the document is to describe What is N-Way Set-Associative Cache? and How I have implemented it?

## Goals

Design and Implement an N-Way Set-Associative Cache with the following features,

1. In memory cache with no communication with backend store.
2. The client interface should be type safe for keys and values.
3. Design the interface as library to distribute to client with no source code access.
4. Provide LRU and MRU replacement algorithms.
5. Provide a way for client to integrate custom replacement algorithm with cache.
6. Ensure that Cache is thread safe. (added goal)

## What is N-Way, Set-Associative Cache?

N-Way Set-Associative cache is a special type of cache which divides the cache into several sub caches called Sets, and each Set contains N number of cache items called Ways. You can think of it as an Array of M number Cache objects where each Cache object contains N number of cache items. So, Set-Associative cache can overall store M*N cache items where M is number of sets and N is number of ways. Set-Associative cache also linked with some replacement policy, which defines which cache item to replace when cache is full and new items needs to be added in cache. The concept came from hardware based set-associative cache mapping which is a hybrid approach to use the flexibility of fully associative cache and the performance and cost efficiency of direct map cache.

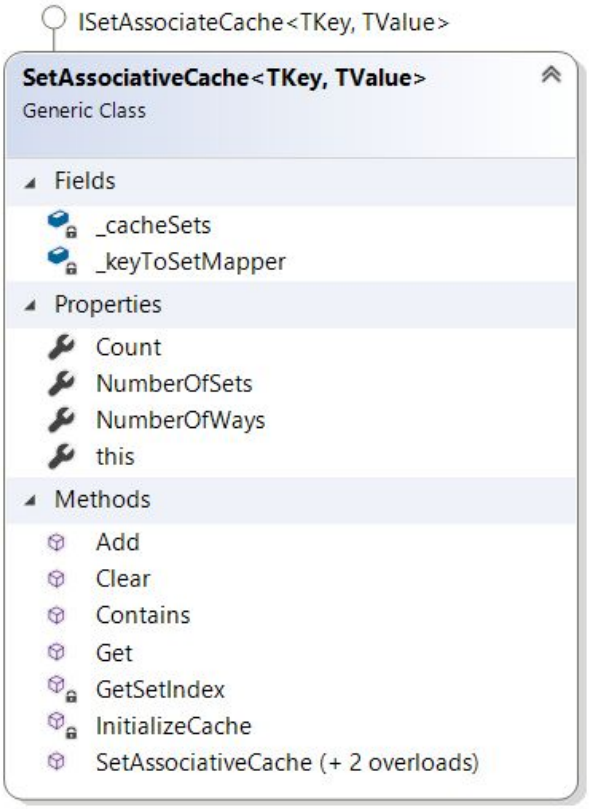
## How N-Way Set-Associative Cache is different than a standard key-value based Cache?

So if N-Way, Set-Associative cache contains M*N number of items ( M is number of sets and N is number of ways) then how its different than having a simple M*N size key-value based cache? That is the question that came to my mind when I started working on this assignment. The way I see is that, Set-Associative cache has a special characteristic that cache items from a particular source can go to particular Cache Set and this can ensure the equal distribution of cache slots. Think about maintaining a fixed size cache for your customers and you want to ensure that each customer should have equal opportunity to store N number of cache items. Without Set-Associative cache, it is possible that all of the cache slots are occupied with the data from few customers (more active ones) and leaving no space for other customers, however, with Set-Associative Cache, you can ensure that Cache items from a particular customer only goes to particular cache set and when that particular Cache Set is full then an existing cache item of the same customer will be removed from Cache in order to add new item, instead of occupying other customer's cache slots.

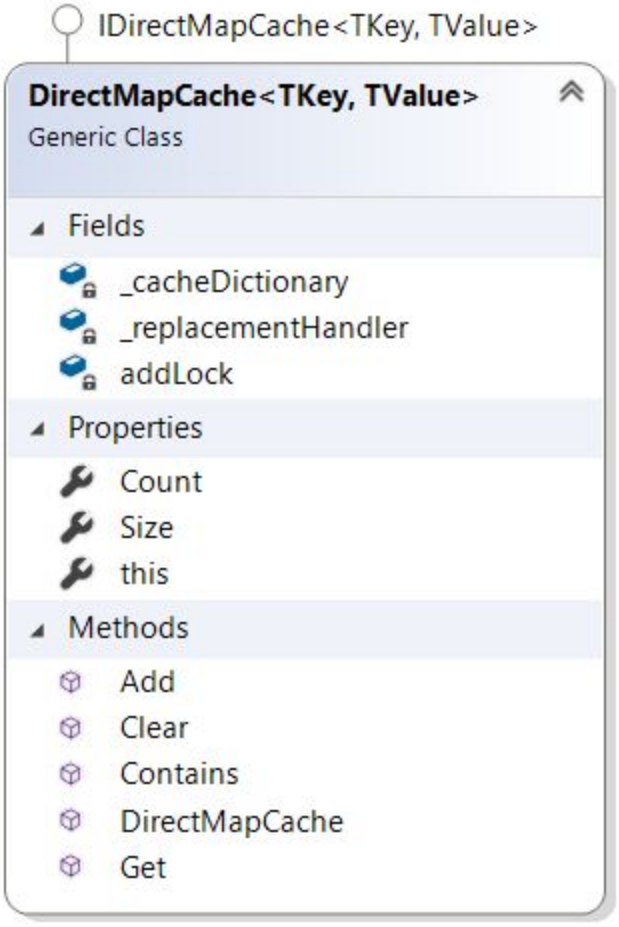
## Implementation Details:

There are four main components of N-Way Set-Associative cache library,

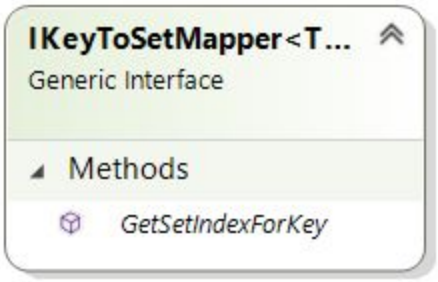
1. **SetAssociativeCache:** This the primary interface to manage cache. **SetAssociativeClass** manages an N size array of **IDirectMapCache,** where N denotes to number of sets in Set Associative Cache. When **SetAssociativeCache** get the request to Retrieve or Add cache item, it uses **IKeyToSetMapper** to find the cache set to perform the action and forward the request to the particular cache set.



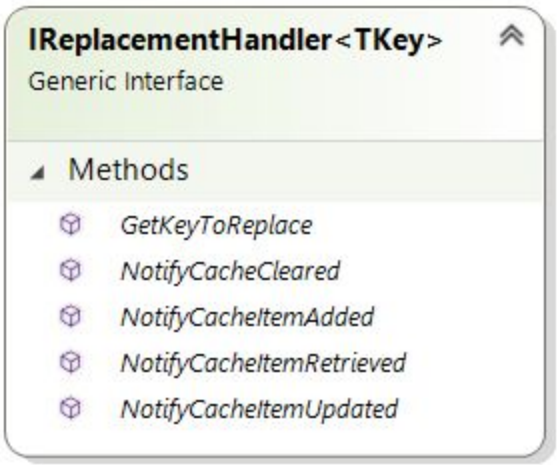

2. **DirectMapCache**: This class represents an individual set inside Set Associative Cache. **DirectMapCache** uses a dictionary to store cache items for faster lookup and Add operation. Whenever cache iterm is retrieved or added, it notifies the **IReplacementHandler** and when Dictionary is full and new item needs to be added, **DirectMapCache** call the **IReplementHandler** to get the key to replace with new item. **DirectMapCache** only stores N number cache items, where N denotes to number of ways in Set Associative Cache.




3. **IKeyToSetMapper:** This is the interface responsible to map cache key to cache set. **IKeyToSetMapper** has only one method **GetSetIndexForKey(Key, numberOfSets)** which takes cache key and number of sets in cache as input and return the 0 baed set index to use for the provided cache key. **SetAssociativeCache** class uses this interface to map keys to sets. I have provided a default implementation of **IKeyToSetMapper** interface which uses *key.GetHashCode() % numberOfSets* to map the cache key to the cache set. You can also write custom **IKeyToSetMapper** implementation and pass it to **SetAssociativeCache** using overloaded constructor e.g.map all cache keys from Customer A to Set 1 etc




4. **IReplacementHandler**: This is the interface responsible to find the cache item which needs to be replaced when cache set is full and new item needs to be added. **DirectMapCache** class notifies the **IReplacementHandler** whenever a cache item is added/updated/retrieved. **ReplacementHandler** uses that information to index the keys based on its replacement policy e.g. keep the list sorted based on usage so that we can replace least or most usage key in O(1) time etc. I have provided two implementation of replacement handlers, **LRUReplacementHandler** which replaces least recently used object (default option) and **MRUReplacementHandler** which replaces most recently used object. You can also write custom replacement handler and pass it to **SetAssociativeCache** using overloaded constructor.




## What Data Structure each component uses? and Why?

1. **SetAssociativeCache** uses an N size array to store N number of cache sets. Runtime to find an cache set for any key takes constant O(1) time.

2. **DirectMapCache** uses *ConcurrentDictionary* to store N number of cache items. *ConcurrentDictionary* helps find and add any cache item in constant time O(1) in thread safe manner.
3. **IReplacementHandler:** Each implementation of IReplacementHandler is responsible to use the right data structure to index the keys which best suits its needs to fetch the key to replace based on its replacement policy in an optimal time. **RUReplacementHandler** which is the base class for **LRUReplacementHandler** and **MRUReplacementHandler** uses doubly LinkedList to store cache keys in such a way that most recently used key is always on top of the list and least recently used key at the bottom. It helps finding the key to replace in LRU and MRU cases in constant time O(1). In order to ensure that keys are always sorted based on usage, we need to remove the key from linked list and add it at the top whenever cache item is retrieved (use). Removing a node from LinkedList takes O(N) time, in order to improve this time, I am storing all linked list nodes in a dictionary. It helps find the the node to remove in constant time O(!) at the cost of additional space. This is one of the area to improve to find ways to optimize space requirement for **RUReplacementHandler**.

## How does Set-Associative Cache support Thread Safety?

I am using ConcurrentDictionary to store cache items which provides thread safety at Data store level to ensure that individual CRUD operations are thread safe and on the top of this, I am doing explicit locking at business logic layer with combination of TryGet() , TryUpdate() and TryAdd() methods of ConcurrentDictionary to ensure data is not changed between Get and Add/Update Or to ensure that  Remove and Add operations on cache are performed as one unit. Here is the code snippet to  Add new cache item in a set.

```
public bool Add(TKey key, TValue value)
        {
            bool isAddedOrUpdated = true;

            if (_cacheDictionary.TryGetValue(key, out TValue oldValue))
            {
                if (_cacheDictionary.TryUpdate(key, value, oldValue))
                    _replacementHandler.NotifyCacheItemUpdated(key);
                else
                    isAddedOrUpdated = false; // another thread updated the
same key
            }
            else
```

```
            {
                // Even though Dictionary's TryAdd() and TryRemove()
methods are thread safe but we need to provide explicit locking. E.g.
                // 1) Cache.Count was Size - 1, thread one and two both
evaluated Count >= Size statement as False at same time and Added new item
in cache which can increase cache maximum size to NumberOfWays + 1.
                // 2) Cache.Count was equal to Size, thread one evaluated
Count >= Size statement as True, Removed the Item, meanwhile Thread two
added new item in the cache. This will also increase the cache maximum size
to NumberOfWays + 1
                lock (addLock)
                {
                    if (_cacheDictionary.Count >= Size)
                    {

_cacheDictionary.TryRemove(_replacementHandler.GetKeyToReplace(), out
TValue removedValue);
                    }

                    if (_cacheDictionary.TryAdd(key, value))
                        _replacementHandler.NotifyCacheItemAdded(key);
                    else
                        isAddedOrUpdated = false; // another thread added
the same key before entering the lock
                }
            }

            return isAddedOrUpdated;
        }
```

All change notification send to **IReplacementHandler** are also synchronized so that Cache is not dependent on **IReplacementHandler's** implementation of how it handles concurrency.

## How to use custom Replacement Handler?

Write new Replacement Handler by implementing IReplacementHandler and instantiate the Set-AssociativeCache class with your custom implementation by using overloaded constructor.

```
var cache = new SetAssociativeCache<CacheKey, CacheValue>(numberOfWays,
numberOfSets,
```

```
                () => new MyCustomReplacementHandler<CacheKey>());
```

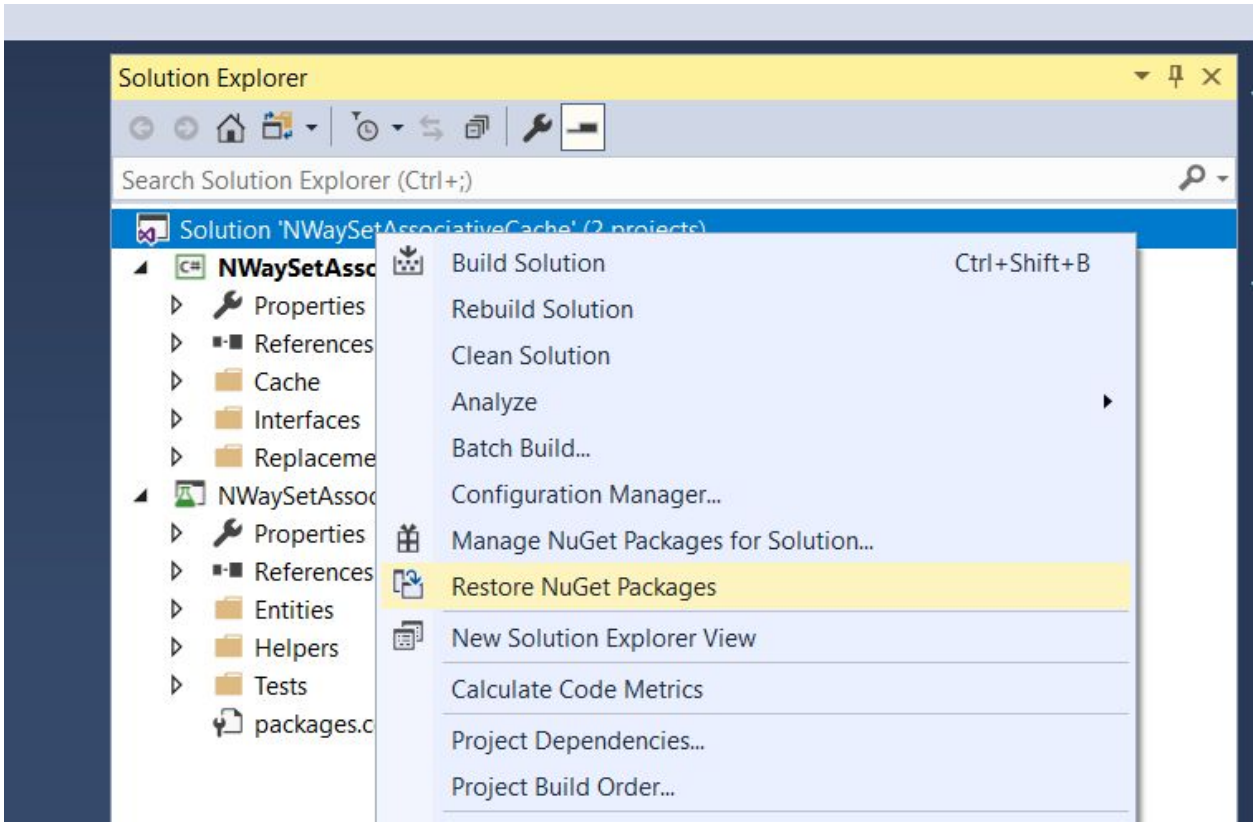
## Areas for improvement?

I would like to improve following areas,

1. Find ways to reduce space requirement for LRU and MRU Replacement Handlers.
2. More unit tests to test thread safety and large data.
3. Integrate with with DataStore to get the item from Store if not available in Cache. ( only if there is a requirement to match the implementation as it works with Hardware based Set-Associative cache)
4. Improve how concurrency is being handled. By default locking doesn't guarantee FIFO order if multiple threads are waiting to acquire the lock. In our situation, we want to ensure the ordering to correctly build recently used cache item list in **RUReplacementHandler.** Something similar to this can help.

## Dependencies:

This library is built using .NET Framework 4.6.1 with Visual Studio 2017. Library itself doesn't have any external dependencies, however, test project is dependent on FluentAssertions 5.2.0 nuget package so please restore nuget packages before compiling the solution.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'NWaySetAssociativeCache' (2 projects)
NWaySetAssc
Properties
References
Cache
Interfaces
Replaceme
NWaySetAssoc
Properties
References
Entities
Helpers
Tests
packages.c
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Analyze
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Restore NuGet Packages
New Solution Explorer View
Calculate Code Metrics
Project Dependencies...
Project Build Order...