N-Way Set-Associative Cache

Aditya Deorha

04/15/18

1 Overview

Cache is a hardware component that stores data so future requests for that data can be served faster. Usually the data stored in a cache is duplicated in the main memory.

There are some important terms associated with a cache. A cache hit occurs when the requested data can be found in the cache. A cache miss occurs when the key is not present in the cache.

Most systems have multiple levels of caches but here we will be implementing just one level of cache. The user will be specifying the cache structure which consists of a few pieces of information, which we'll talk about next.

2 Requirements

Design and implement an N-way, Set-Associative cache, with the following features:

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 algorithms.
5. Provide a way for any alternative replacement algorithm to be implemented by the client and used by the cache.

3 Cache Structure

When trying to read from or write to a location in main memory, CPU checks whether the data from that location in already present in the cache. If so, the processor will read from or write to the cache instead of the main memory. The reason for this is that average access time (AAT) to the cache is much less as compared to the AAT of main memory.

3.1 Cache Entries

Data transfer between main memory and caches happen via blocks. These blocks are of fixed size and are called cache lines or simply lines in cache. A block might have many words worth of data. When we attempt to ”get” a specific word, the entire block is lifted from main memory into the cache, in case of a miss.

3.2 Replacement Policy

When we copy data from main memory into the cache, we may have to evict a block which is currently present in the cache. The policy which decides which block to be removed is known as a replacement policy. We are implementing 2 replacement policies and are allowing the client to implement their own custom policy as well.

```
private final static String LRU = "LRU";
private final static String MRU = "MRU";
private final static String CUSTOM = "CUSTOM";
```

LRU or Least Recently Used replacement policy discards the least recently used items first.

```
public class LRUPolicy implements Policy {
  @Override
  public int evictEntryIndex(Entry[] entries, int startIndex,
  int endIndex) {
    int index = 0;
    Timestamp earliestTime = new Timestamp(System.nanoTime());
    for(int i=startIndex; i<=endIndex; i++)
    {
      if(entries[i].occupied && entries[i].timestamp.before(earliestTime))
      {
        earliestTime = entries[i].timestamp;
        index = i;
      }
    }
    return index;
  }
}
```

MRU or Most Recently Used replacement policy discards the most recently used items first.

```
public class MRUPolicy implements Policy {
  @Override
  public int evictEntryIndex(Entry[] entries, int startIndex, int endIndex) {
    int index = 0;
    Timestamp latestTime = new Timestamp(0);
    for(int i=startIndex; i<=endIndex; i++)
    {
      if(entries[i].occupied && entries[i].timestamp.after(latestTime))
      {
        latestTime = entries[i].timestamp;
        index = i;
      }
    }
    return index;
  }
}
```

Custom replacement policy allows the client to define their own replacement policy in ClientCustomPolicy, which implements Policy interface. For now, we have added a default policy there.

```
public class ClientCustomPolicy implements Policy {
  @Override
  public int evictEntryIndex(Entry[] entries, int startIndex, int endIndex) {
    return (startIndex + (endIndex-startIndex)/2); //Default implementation
  }
}
```

3.3 Associativity

If the block to be copied into the cache can be placed anywhere in the cache, it's a full associative cache. If each entry in main memory can go in just one place in the cache, it is a direct mapped cache. We have implemented an N-way set-associative cache which allows us to compromise between the two extremes.

3.4 Other Parameters

Some of the other parameters that the client needs to provide are:

- Number of sets: The client can provide the total number of sets that we have in the cache. Each set holds "associativity" number of lines.
- Cache size: If number of sets isn't given, then the client can provide total cache size and the block size.
- Block size: A block might contain multiple words but for our implementation, we don't need to know at that granurality. We can use cache size and block size along with the associativity to calculate the total number of sets.
- Cache Entry: This corresponds to one line in the cache.
- Array of Cache Entries: This is all space where we are storing the cache entries.

4 Initializing the Cache

We've provided 2 constructors to take care of the ways to instantiate our cache. The first one uses cache size and block size to calculate the number of sets.

```
public NWayCache(int associativity, int cacheSize, int blockSize,
String evictionPolicy)
{
  this.associativity = associativity;
  this.cacheSize = cacheSize;
  this.blockSize = blockSize;
  this.numSets = cacheSize/blockSize;
  this.entries = new Entry[associativity * numSets];
  for(int i=0; i<entries.length; i++)
  {
    entries[i] = new Entry();
  }
  this.replacementPolicy = evictionPolicy;
}
```

The second constructor takes the number of sets directly from the client.

```
public NWayCache(int associativity, int numSets, String evictionPolicy)
{
   this.associativity = associativity;
   this.numSets = numSets;
   this.entries = new Entry[associativity * numSets];
   for(int i=0; i<entries.length; i++)
   {
      entries[i] = new Entry();
   }
   this.replacementPolicy = evictionPolicy;
}
```

We have a Policy interface which LRUPolicy, MRUPolicy and ClientCustomPolicy are implementing. We have a Cache interface which NWayCache is implementing. We also have a Cache Entry class which provides the building block of the class using the following data members and constructors.

```
public class Entry {
  Object value;
  Timestamp timestamp;
  int tag;
  boolean occupied;

  public Entry() {
    this.value = null;
    this.timestamp = null;
    this.tag = 0;
    this.occupied = false;
  }

  public Entry(Object value, Timestamp timestamp, int tag,
  boolean occupied) {
    this.value = value;
    this.timestamp = timestamp;
    this.tag = tag;
    this.occupied = occupied;
  }
}
```

5 Cache Implementation

The get and put functions are implemented in the NWayCache class.

5.1 Get

Get function looks for the block in the cache. If it finds the block and the key matches, it returns the value and updates the timestamp. If it doesn't find the block, it generates a null Object in lieu of a value being retrieved from the memory. The null object is then "put" in the cache, which might end up replacing an existing entry in the cache.

```
public Object get(Object key) {
  Utility util = new Utility();
  int hashKey = util.hash(key);
  int startIndex = util.getStartIndex(hashKey, numSets, associativity);
  int endIndex = util.getEndIndex(startIndex, associativity);

  for(int i=startIndex; i<=endIndex; i++)
  {
    if(entries[i].occupied && entries[i].tag == hashKey)
    {
      Timestamp currentTime = new Timestamp(System.currentTimeMillis());
      entries[i].timestamp = currentTime;
      return entries[i].value;
    }
  }
  Object object = null;
  put(key, object);

  return object;
}
```

5.2 Put

Put function first looks for the same block to be already there in the cache. If it's present, it updates the existing entry. Else, it creates a new entry and adds it to the cache on the first empty line or evicts a line and adds it there.

We also added a check to make sure that the value and the key that we are trying to put, has the same class type as the key and value we tried to add for the first time.

```
public void put(Object key, Object value) {
  if(value!= null && firstPutCall)
  {
    keyType = key.getClass();
    valueType = value.getClass();
    firstPutCall = false;
  }

  if(key.getClass() != keyType)
  {
    System.out.println("The key type is inconsistent");
    return;
  }

  if(value!=null && value.getClass() != valueType)
  {
    System.out.println("The value type is inconsistent");
    return;
  }
  ....
}
```

5.3 Hash

We need to calculate the hash of the given key to return an integer, which we will use to calculate the start and the end index of the cache where we might get or put a cache line. We kept it quite simple, using the default hashCode function that Java provides along with some extra modification.

```
public int hash(Object key)
{
  return (key.hashCode()*37 + 17);
}
```

If it would've been a real memory/cache system where the key was supposed to be an address, there we wouldn't have to hash, but use the most significant bits as tag. That's not the case here though.

5.4 Calculating start and end index

We need to calculate the start and end index to specify where to get the cache entry from or put a new cache entry at.

```
public int getStartIndex(int hashKey, int numSets, int associativity)
{
  int startIndex = (hashKey % numSets) * associativity;
  return startIndex;
}
public int getEndIndex(int startIndex, int associativity)
{
  int endIndex = startIndex + associativity - 1;
  return endIndex;
}
```

6 Conclusion

The n-way set-associative cache has been implemented here with multiple replacement policies and was tested for different associativities and number of sets. The custom replacement algorithm provided to the user can be modified to LFU (Least Frequently Used) or any of the other replacement algorithms that exist today.