INF-2200

Assignment 3, Cache

Andreas Berntsen Løvland & Magnus Dahl-Hansen

October 2020

Introduction

In this assignment we describe how we implemented a cache simulator with two levels for a memory subsystem. Level one containing two caches: a read-only instruction cache and a data cache which supports both reads and writes. The level two cache being a unified cache also supporting both reads and writes. The goal for the assignment is to find the best cache design and measure hit and miss rates on a benchmark.

Technical Background

The most relevant topics covered in this assignment are general basics of caches, different cache operations (such as fetch, writes and reads), cache addresses and cache replacement policies.

Cache basics

A cache is a component located near the CPU that stores copies of data that can be accessed faster than data from main memory. Modern CPU's normally contains different levels of caches where the first level is the smallest and fastest while the last cache is the largest but also the slowest. In this assignment we focus on a two-level, set-associative cache. A set-associative cache is a cache that is divided into sets where each set contains a given number of blocks. If we have an 8-way set-associative cache it means that the cache has 8 blocks in each set. The number of sets depends among other things on the cache size.

Cache operations

The cache performs a read operation when the CPU requests data provided with an address. The cache must then check whether the address exists in the cache or not. If the data exists in the cache it results in a hit and the data is returned to the CPU. If the requested address is not in

the cache it results in a miss and the cache must access main memory and allocate space to store the data fetched from memory, and then return this data to the CPU.

A cache write operation occurs when the cache must write data to main memory. It can also occur when the cache is full and needs to free some space to store other instructions. If the cache must write to memory because the cache is full, a problem can occur: Since data in the cache are copies from main memory containing the same address value it becomes a problem if the data in the cache has been modified. This means we have two different values corresponding to the same address. If the data has been modified the cache must first write the modified data back to memory before it can allocate space for new data. Generally, there are two ways of writing back to memory namely, write-back and write-through. Write-through means that the cache writes the modified data instantly back to memory, while write-back means that the cache awaits to write modified data back to memory until it is needed to allocate memory for other instructions.

Cache addresses

As mentioned, a set-associative cache consists of sets and blocks. To navigate through the cache and locate an address the address is divided into three parts: tag, index and block offset. The length needed for each part is decided by the cache size, block size, the associativity of the cache and the binary length of the address. The index is used to locate which set in the cache an address is stored in. The tag is then used to locate which block in the set the address is in and lastly the block offset is used to reference the exact byte in the block. In addition to these the binary address contains two more considerable bits, the valid bit and dirty bit. The valid bit represents if there is data stored at a specific place in the cache and is set to “1” if so. The dirty bit represents if data in the cache has been modified and is set to “1” if so.

Replacement policies

Replacement policies are algorithms that choose which data elements to discard from the cache if the cache is full. In this assignment the cache is implemented with the least recently used policy meaning that the data elements that has been used the least gets discarded. An algorithm like this requires one to keep track of what was used when, and adding some sort of value to each data element most often as an “age bit”.

Design/Implementation

The three caches we created are all based on the same structure. The structure for the caches contains among other things parameters such as cache size, block size and an array. The array that each cache stores simulates the cache blocks which are stored in sets inside the cache. Each element in the array is going to be assigned a specific index value which represents which set it belongs to. The blocks in the array gets assigned a least-recently-used value, which indicates that the cache block with the highest value on each index is the approximated least recently used one and are eventually going to be replaced with new data.

Before we perform a read or a write with a given address, we check if we have a hit in the cache for that address. To check if the address is already in the cache, we iterate over the all the elements of the cache until we find the index that matches the one in the given address. When we have hit the beginning of the index, we search through that index to find the tag provided by the given address. If the tag is not in the cache it results in a cache miss, but otherwise we get a cache hit. If we get a hit with the tag on the index, we go back to the beginning of that index to find which cache block we are going to replace the new data with. We iterate over the index to find the cache block with the highest least recently used (LRU)-value, and check if the valid bit of that block is set to 1. If it is set to 1, we have a cache hit and will increment the LRU-values on the index we were working on. The cache block which had the highest LRU-value gets set to 0, which means that it is the most recently used one. If we get a cache miss for a cache, we go to the next level cache and repeat the same process as for the first cache.

Whenever we want to add an address to our cache, we do similar operations as when we check for an element in the cache. We iterate over the blocks in our array and check for the correct index and find the position in the cache block with the highest LRU-value. Since we have a write-back policy for the caches, we must check if the cache block we are going to replace is *dirty* or not. If the cache block is not dirty it means that the lower memory has the same values as the cache for that specific address. If the cache block is dirty, we have not written the dirty value to the rest of the lower memory. Therefore, we check if the block is dirty or not, and if it is not dirty, we can just replace the cache block with the highest LRU-value with our new cache block. If the cache block is dirty however we must write the tag on that cache block into the lower memory before replacing it with the new one. For each time we have a write instruction we must set the *dirty bit* of the cache block, which is being written to, to high. We did this by

iterating through the array of the cache blocks until we found the correct index by the given address, and searched for the most recently used cache block on the index, and set its dirty bit to either high or low depending on which one we want for the different scenarios (if we have a read or a write).

We also implemented a write through policy. For each time we are having a memory read with the write through policy we find the correct index in the cache array depending on what the address is and read the data from the memory to that cache block. As the caches can have a mixture of write-back and write-through policies we still have to change the dirty bit. For instance, if the instruction cache has write-through, the level two cache can still have a write-back policy and will need the dirty bit in case if there were multiple levels of caches. Note that we also must update the LRU-values on the index after we have done a read for the write-through policy.

When we have a write instruction to the caches with the write-through policy we find the cache block with the highest LRU-value on the given address' index. We write the new data into the cache block on the correct index before we also write the same data as we just wrote into the next lower memory unit in the hierarchy. With this method we could in theory use write-through on an infinite amount of caches if they were all connected.

As we in this assignment are only counting hits and misses for each of the caches, we need variables in the caches which controls these values. The hit-percentage for each of the caches get printed out in the terminal before the memory the caches allocated gets deallocated.

Evaluation/discussion

After running our simulator on a *mergesort* algorithm that sorts 1000 elements, we got some interesting results. The next page shows different test results based on different parameters such as cache size, block size and associativity. Every result shows the hit percentage for each cache level with both write-back and write-through policies:

Parameters as specified in the assignment

L1 instruction cache:	
Size	32
Blocksize	64
Associativity	4
Policy	write-back
Hit rate	99,49 %

VS

L1 instruction cache:	
Size	32
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	99,49 %

L1 data cache:	
Size	32
Blocksize	64
Associativity	8
Policy	Write-back
Hit rate	2,50 %

VS

L1 data cache:	
Size	32
Blocksize	64
Associativity	8
Policy	write-through
Hit rate	91,96 %

L2 unified cache:	
Size	256
Blocksize	64
Associativity	8
Policy	Write-back
Hit rate	99,25 %

VS

L2 unified cache:	
Size	256
Blocksize	64
Associativity	8
Policy	write-through
Hit rate	79,40 %

Increased cache size

L1 instruction cache:	
Size	256
Blocksize	64
Associativity	4
Policy	write-back
Hit rate	99,70 %

VS

L1 instruction cache:	
Size	256
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	99,70 %

L1 data cache:	
Size	256
Blocksize	64
Associativity	8
Policy	Write-back
Hit rate	22,89 %

VS

L1 data cache:	
Size	256
Blocksize	64
Associativity	8
Policy	write-through
Hit rate	99,32 %

L2 unified cache:	
Size	256
Blocksize	64
Associativity	8
Policy	Write-back
Hit rate	98,50 %

VS

L2 unified cache:	
Size	256
Blocksize	64
Associativity	8
Policy	write-through
Hit rate	2,40 %

Only 4-way associativity

L1 instruction cache:	
Size	32
Blocksize	64
Associativity	4
Policy	write-back
Hit rate	99,49 %

VS

L1 instruction cache:	
Size	32
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	99,49 %

L1 data cache:	
Size	32
Blocksize	64
Associativity	4
Policy	Write-back
Hit rate	6,20 %

VS

L1 data cache:	
Size	32
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	96,49 %

L2 unified cache:	
Size	256
Blocksize	64
Associativity	4
Policy	Write-back
Hit rate	96,63 %

VS

L2 unified cache:	
Size	256
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	14,53 %

4-way plus increased cache size

L1 instruction cache:	
Size	256
Blocksize	64
Associativity	4
Policy	write-back
Hit rate	99,49 %

VS

L1 instruction cache:	
Size	256
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	99,70 %

L1 data cache:	
Size	256
Blocksize	64
Associativity	4
Policy	Write-back
Hit rate	2,50 %

VS

L1 data cache:	
Size	256
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	99,51 %

L2 unified cache:	
Size	256
Blocksize	64
Associativity	4
Policy	Write-back
Hit rate	99,25 %

VS

L2 unified cache:	
Size	256
Blocksize	64
Associativity	4
Policy	write-through
Hit rate	0,60 %

Decreased block size

L1 instruction cache:	
Size	32
Blocksize	32
Associativity	4
Policy	write-back
Hit rate	99,49 %

VS

L1 instruction cache:	
Size	32
Blocksize	32
Associativity	4
Policy	write-through
Hit rate	99,49 %

L1 data cache:	
Size	32
Blocksize	32
Associativity	8
Policy	Write-back
Hit rate	14,80 %

VS

L1 data cache:	
Size	32
Blocksize	32
Associativity	8
Policy	write-through
Hit rate	93,51 %

L2 unified cache:	
Size	256
Blocksize	32
Associativity	8
Policy	Write-back
Hit rate	98,10 %

VS

L2 unified cache:	
Size	256
Blocksize	32
Associativity	8
Policy	write-through
Hit rate	77,55 %

Decreased block size + Increased cache size

L1 instruction cache:	
Size	256
Blocksize	32
Associativity	4
Policy	write-back
Hit rate	99,68 %

VS

L1 instruction cache:	
Size	256
Blocksize	32
Associativity	4
Policy	write-through
Hit rate	99,68 %

L1 data cache:	
Size	256
Blocksize	32
Associativity	8
Policy	Write-back
Hit rate	41,19 %

VS

L1 data cache:	
Size	256
Blocksize	32
Associativity	8
Policy	write-through
Hit rate	99,35 %

L2 unified cache:	
Size	256
Blocksize	32
Associativity	8
Policy	Write-back
Hit rate	97,86 %

VS

L2 unified cache:	
Size	256
Blocksize	32
Associativity	8
Policy	write-through
Hit rate	2,08 %

Based on the test results, we found that a higher cache size results in a higher hit rate except for the L2 cache. The hit percentage for the L2 cache decreased when the cache size increased for both L1 caches. Our theory for the decrease in hit percentage in L2 cache is that when the L1 caches are bigger they are likely to get more hits and thus we won't check for hits in the L2 cache. Since the L2 cache will be checked fewer times than the L1 caches will, the hit percentage is more likely to decrease.

After some debugging of our implementation, we found that there was something wrong with how we add elements to the cache. We found that we always replace only the top element in each block leaving in some cases (based on the associativity) three unused spaces in each block. In this case this means we only use ¼ of our cache's capacity… Therefore, based on the results, a lower cache associativity results in a higher hit percentage. If we lower the associativity in each cache, we get more sets and fewer blocks in each cache, and fewer blocks in each cache means we use more of the cache's capacity.

Based on the results we concluded that decreased block size meant insignificant changes in hit percentage.

Conclusion

In this assignment we have described our implementation of a two-level cache simulator. To complete the simulator and for the cache to perform the specific operations *read* and *write* we had to create auxiliary functions. Some of these functions created a cache with specific traits, added elements to it, set dirty and valid bits and checked for elements within the cache. We have also described some parts of the how the cache works that may be important to understand before looking at the actual implementation.