University of Tromsø

INF-2200

Assignment 3 - Memory simulator

Tengel Ekrem Skar and Alexander Einshøj

Department of Computer Science

November 4, 2015

1 Introduction

In this paper, a cache simulator for a memory system with a level-1 instruction cache, a level-1 data cache and a unified level-2 cache is implemented. The goal is to find the best cache design for a chosen benchmark.

The chosen benchmark is the sorting algorithm heapsort. As input, an array of 10000 data points in descending order was used. Heapsort consists of two parts: heapify and sift-down, where the heapify function seems to have a somewhat random memory access pattern, while sift-down accesses the same part of memory a lot.

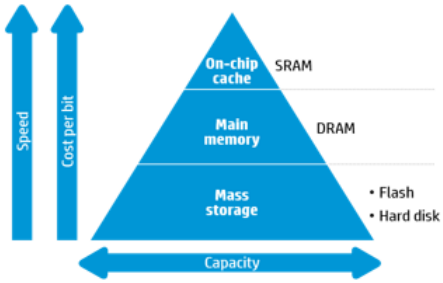
source:

2 Technical Background

2.1 Cache

The memory hierarchy of a computer consists of different layers, L1 instruction cache, L1 data cache and L2 unified cache, which is considerably larger, but slower than the former two.

A cache is necessary given the rapid development of computer technology. The cache acts as the middle-point between the CPU and the RAM, by storing frequently requested addresses in the cache, making it much more available to the CPU. The CPU can access data a lot faster than the RAM can send it, which creates a bottleneck between the two.



Simplified Memory Hierarchy

2.2 Memory requests

The cache handles different kinds of requests from the CPU. When the cache receives a read request, it can arrive either from the PC(for an instruction), or from the ALU(for data).

2.2.1 Read

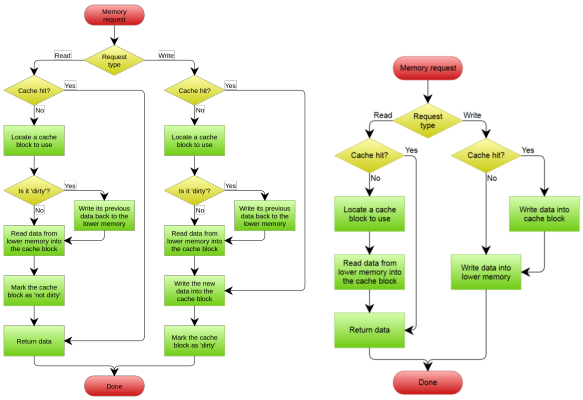
If the request comes from the ALU, it is evaluated as a read request, and the address is sent to the data cache. If the valid bit is 1 and the tag calculated on the address is equal to a tag in the cache, it is considered a hit. This means that the requested word is available in the given block. There can be a number of unique words in a block, and the right one needs to be selected, by using the block index field. If the cache request is a miss, the address will be pushed down the memory hierarchy to the unified cache looking for a match, and possibly all the way down to RAM. The address will then be written in to both the unified cache and the data cache.

2.2.2 Fetch

When a request arrives from PC, it's considered a fetch, and handled by the L1 instruction cache.

2.2.3 Write

There are two options for handling a write request; write-through and write-back. Write-through always updates both the cache and the next lower level of the memory hierarchy, ensuring that data is always consistent between the two. Write-back handles the request by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced. A commonly used implementation is LRU(Least Recently Used), which replaces the least recently used block with the written data, if the block is not already in memory. The LRU algorithm also applies to reads and fetch instructions on the different caches.



3 Design

This solution uses write-through as its cache write policy. When using this as policy, there's no need of keeping record over which lines have been written, because the data will always be consistent throughout the levels of the memory hierarchy. The linesizes, setsizes and cachesizes are all constructed as global variables, so they are easily changeable. When creating the caches, one needs to consider the requirements for reading from and writing to different cache levels. The solution is a general approach for designing any cache of any shape, because they all behave alike.

When creating the cache write, you need to know whether a given block has a valid bit, if the tag is equivalent to the requested write tag, and also which block has the highest LRU value. This is the minimal information required to always update the correct block.

3.1 Implementation

The assignment is written in C.

4 Methodology

Whether a request is a call or miss, is decided when the cache gets a read signal. The given cache will loop through the set located at an index. While inside the set, it determines if a given block has a valid bit or not. If a block is valid, there's made a comparison to confirm that the

tag in the block is equal to the tag at the requested address. If all of this evaluates to true, it's a hit. In all other cases, it's a miss.

The first correctness trace is designed to read two addresses, each twice in a row. A cache with a size of only 4 bytes should then get a 50% hit rate, because it can only hold one value at once. It will always miss the first time it's read, before writing the new address to cache, making the read a hit the second time around.

The second correctness trace is designed with the case of writing, then reading. In a cold 4 byte cache, it writes data to an address, then attempts to read from that same address. This is performed for two different addresses, and should yield a hitrate of 100%.

The benchmark is performed with two different cache sizes, the first with 32KiB, the second with 4KiB. The associativity varied between 16 and 1, and the block size between 264, 64 and 16 bytes.

5 Results & Discussion

```
XXXXXXXXXXXXXX
BENCHMARK ON L1_DATA WITH CACHE SIZE 32768 BYTES
XXXXXXXXXXXXXX
================================================================================================
associativity:       | 16
linesize(bytes):     | 256      |   64      |   16      |
------------------------------------------------------------------------------------------------
HITS:                | 6811898  |   6807065 |   6794706 |
MISSES:              | 2821     |   7654    |   20013   |
TOTAL:               | 6814719  |   6814719 |   6814719 |
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.000414 |   0.001123|   0.002937|
================================================================================================
associativity:       | 8
linesize(bytes):     | 256      |   64      |   16      |
------------------------------------------------------------------------------------------------
HITS:                | 6812010  |   6807399 |   6796024 |
MISSES:              | 2709     |   7320    |   18695   |
TOTAL:               | 6814719  |   6814719 |   6814719 |
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.000398 |   0.001074|   0.002743|
================================================================================================
associativity:       | 4
linesize(bytes):     | 256      |   64      |   16      |
------------------------------------------------------------------------------------------------
HITS:                | 6811813  |   6808473 |   6798346 |
MISSES:              | 2906     |   6246    |   16373   |
TOTAL:               | 6814719  |   6814719 |   6814719 |
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.000426 |   0.000917|   0.002403|
================================================================================================
associativity:       | 2
linesize(bytes):     | 256      |   64      |   16      |
------------------------------------------------------------------------------------------------
HITS:                | 6812089  |   6809629 |   6801065 |
MISSES:              | 2630     |   5090    |   13654   |
TOTAL:               | 6814719  |   6814719 |   6814719 |
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.000386 |   0.000747|   0.002004|
================================================================================================
associativity:       | 1
linesize(bytes):     | 256      |   64      |   16      |
------------------------------------------------------------------------------------------------
HITS:                | 6801186  |   6806444 |   6802085 |
MISSES:              | 13533    |   8275    |   12634   |
TOTAL:               | 6814719  |   6814719 |   6814719 |
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.001986 |   0.001214|   0.001854|
================================================================================================
XXXXXXXXXXXXXX
BENCHMARK ON L1_DATA WITH CACHE SIZE 4096 BYTES
XXXXXXXXXXXXXX
================================================================================================
associativity:       | 16
linesize(bytes):     | 256          |   64          |   16
------------------------------------------------------------------------------------------------
HITS:                | 6567583      |   6800924     |   6774114
MISSES:              | 247136       |   13795       |   40605
TOTAL:               | 6814719      |   6814719     |   6814719
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.036265     |   0.002024    |   0.005958
================================================================================================
associativity:       | 8
linesize(bytes):     | 256          |   64          |   16
------------------------------------------------------------------------------------------------
HITS:                | 6565979      |   6795468     |   6774084
MISSES:              | 248740       |   18251       |   40635
TOTAL:               | 6814719      |   6814719     |   6814719
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.036500     |   0.002678    |   0.005963
================================================================================================
associativity:       | 4
linesize(bytes):     | 256          |   64          |   16
------------------------------------------------------------------------------------------------
HITS:                | 6552087      |   6775338     |   6773449
MISSES:              | 262632       |   39381       |   41270
TOTAL:               | 6814719      |   6814719     |   6814719
------------------------------------------------------------------------------------------------
MISSRATE:            | 0.038539     |   0.005779    |   0.006056
```

In the first picture, we assign a cache size of 32KiB. Across the board, there are very low miss rates. A general trend is that when set associativity is lowered, the miss rate lowers as well. An exception is when the associativity is lowered from 2 to 1 and the line size is either 256 or 64 bytes. In this case, it rises significantly. The principle of temporal locality explains this, by stating that data which has been accessed earlier, will be accessed again sometime soon. Since the set size is 1, the LRU cannot be exploited.

With a cache size of 4KiB, the tendencies seem to be a bit different than for a cache size of 32KiB. All the different line sizes have an ascending miss rate the lower the associativity is. It's worth taking notice of the fact that a cache with a line size of 256 bytes will perform worst of the alternatives, as opposed to in the 32KiB cache. This can be explained by the relation between line size and cache size, since line size is big in proportion to the cache size.

Since the cost per MiB of cache is linear, the different alternatives in each of the two cache sizes approximately cost the same. The best choice for a 32KiB cache specifically designed for this benchmark would have a line size of 256 bytes and an associativity of 2, while the best choice for a 4KiB cache would have a line size of 64 bytes and an associativity of 16.

The 32KiB cache is 8 times as expensive as the 4KiB cache, since the cost is linear with cache size. At the same time it only reduces miss rate by 80%. This yields a cost efficiency for the 32KiB cache of about 65% of the 4KiB cache's cost efficiency. This means that if the 32KiB cache was to be more efficient than the 4KiB cache, it would need almost double the performance it currently shows.

6 Conclusion

The results from the testings show that a 4KiB cache is more cost effective than a 32 KiB cache if implemented specifically for the heapsort benchmark with 10000 elements. Due to this, a 4KiB cache may possibly a better choice.

Bibliography

Memory hierarchy, November 2015. URL http://www8.hp.com/hpnext/sites/default/files/Simplified%20Memory%20Hierarchy.PNG.

Write-back, November 2015a. URL https://upload.wikimedia.org/wikipedia/commons/thumb/c/c2/Write-back_with_write-allocation.svg/2000px-Write-back_with_write-allocation.svg.png.

Write-through, November 2015b. URL http://wiki.expertiza.ncsu.edu/images/4/48/Writethrough.png.