ARCHITECTURE REPORT

CSC2002S Assignment 4

Student Name: Steffi Baumgart

Student Number: BMGSTE001

24 October 2016

Introduction

The aim of this assignment is to implement a multilevel direct-mapped cache that supports a read operation for a fictional machine with a non-strictly inclusive cache policy. The cache needs to be configurable to support any number of blocks and any block size for a 16-bit address space. These configuration options are passed in as command-line arguments at run-time and the application outputs the number of hits and misses for each level of cache, as well as the number of CPU cycles required.

The configurations tested were as follows:

- L1 cache with:
  - 16 blocks and 16 bytes/block
  - 16 blocks and 32 bytes/block

- L1 cache and L2 cache with:
  - L1: 16 blocks and 16 bytes/block; L2: 64 blocks, 64 bytes/block
  - L1: 8 blocks and 32 bytes/block; L2: 64 blocks, 64 bytes/block

Lists of memory files (with an .addr file extension) were given by the lecturer to be read from. These files have an address on each line and are presented as 16-bit hex values. Namely:

- assignment.addr (Simulates the assignment and reassignment of a large number of variables)
- matmul5x5_loop.addr (Approximation of a 5x5 matrix multiplication)
- fourier.addr (Approximation of the reads needed for algorithm to calculate fourier coefficients)
- heapsort.addr (This is large file that approximates a Heapsort)
- small_set.addr (Very small file)
- small_L2_set.addr (Very small file)

Technical Overview

I have a single class for my application, my_cache_simulator, written in Python and using the dictionary data structure (Python's form of a hash table) for the cache implementation.

The following arguments are taken for the cache configuration:

I. The name of the file to be read
II. The cache setup used
III. The blocksize for L1 cache
IV. The bytes/block for L1 cache
V. The blocksize for L2 cache
VI. The bytes/block for L2 cache

(With parameters V and VI being optional.)

Instructions on how to run the application (more in the README file):

- ✓ Type the following for L1 Cache only:
  python my_cache_simulator L1 <blocksize> <bytesperblock>
  Example: For L1 cache with a blocksize of 16 and 32 bytes per block
  → python my_cache_simulator L1 16 32

- ✓ Type the following for L1 Cache and L2 Cache:
  python my_cache_simulator L1L2 <blocksize> <bytesperblock>
  Example: For L1 Cache with a blocksize of 16 and 16 bytes per block, and a L2 Cache with a blocksize of 64 and 64 bytes per block
  → python my_cache_simulator L1L2 16 16 64 64

Structure of Application:

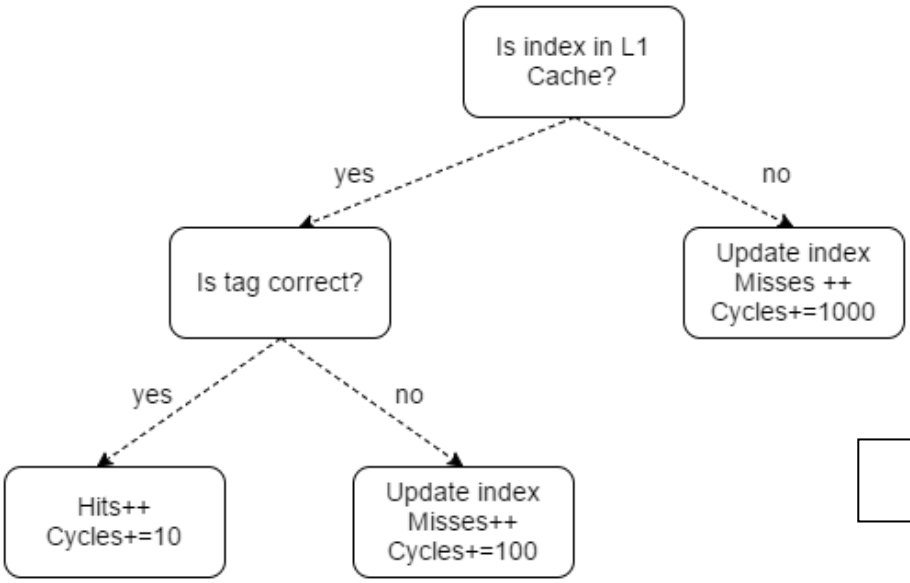
1. Take in the command-line parameters
2. Read in the file
3. Convert each hexadecimal address to binary
4. Pad the address with leading zeros until the length of the address is 16
5. Execute L1 or L1L2 method depending on specified setup
6. Calculate the index size (with the formula *math.log(blocksize, 2))*
7. Calculate the offset size (with the formula *math.log(bytesperblock, 2))*
8. Calcuate the tag cut off (*16 - index size + offset size)*
9. Determine the tag *(address[0:tag cut off])*
10. Determing the index (*address[tag cut off: 16 - offsetsize)*
11. Perform cache operations*
12. Ouput number of hits, misses, and cycles

*detailed in the diagrams on page 3

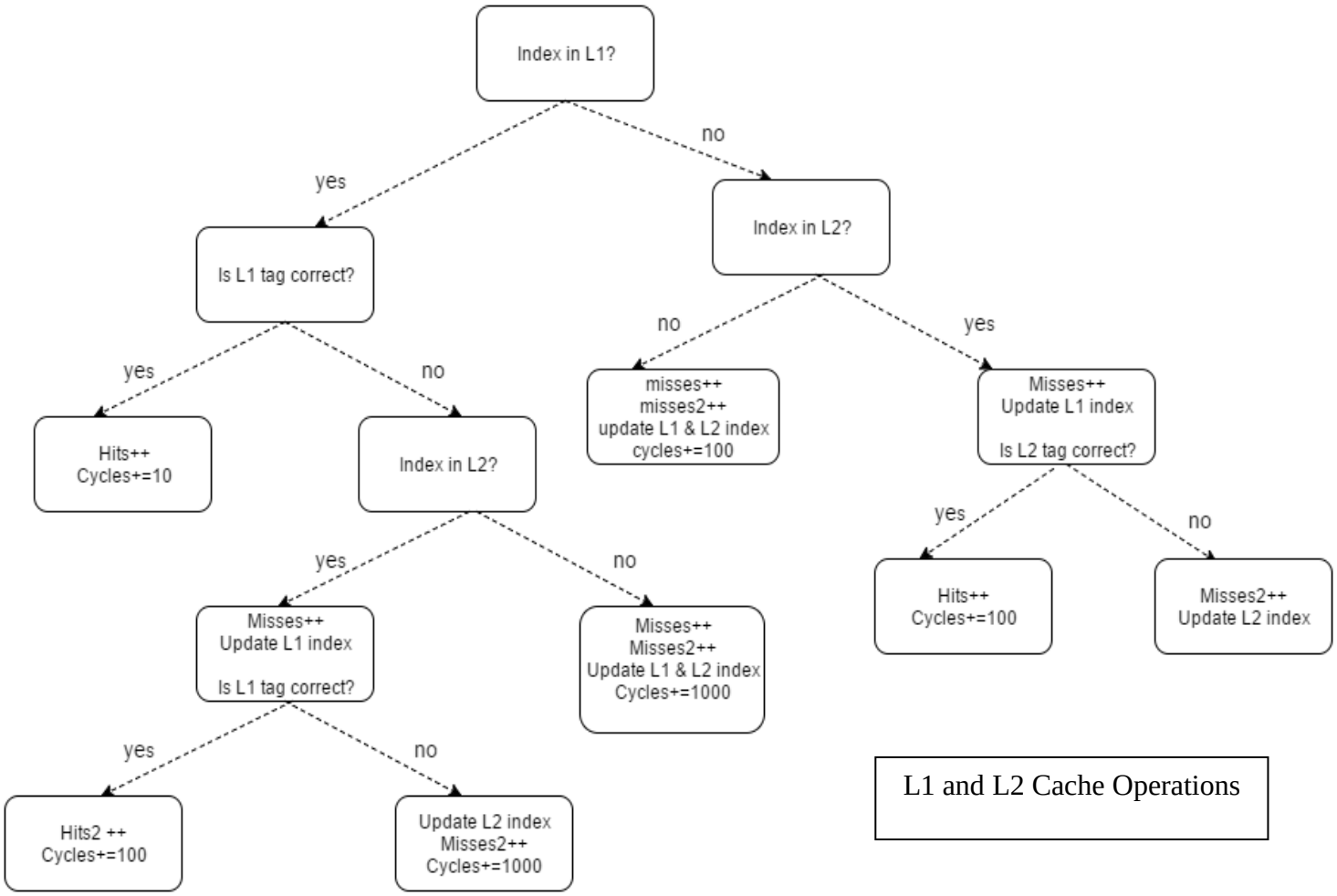
How the CPU cycles are calculated

- L1 hit: 10 cycles
- L2 hit: 100 cycles
- Main memory fetch: 1000 cycles

The results can be verified by performing 2 simple checks after every run:
For L1 and L1L2: The sum of the hits and misses = the number of addresses in the file
For L2 only: The sum of the the L2 hits and the L2 misses = the number of L1 misses



L1 Cache Operations



L1 and L2 Cache Operations

[3]

Results and Analysis

Memory File	L1 Cache		L1 Cache and L2 Cache	
	16 Blocks 16 Bytes/Block	16 Blocks 32 Bytes/Block	L1 : 16 Blocks 16 Bytes/Block L2: 64 Blocks 64 Bytes/Block	L1 : 8 Blocks 32 Bytes/Block L2: 64 Blocks 64 Bytes/Block
assignment.addr				
L1 Hits	60098	65420	60098	62446
L1 Misses	11504	6182	11504	9156
L2 Hits	-	-	9728	7234
L2 Misses	-	-	1776	1922
Cycles	12104980	6836200	3349780	3269860
matmul5x5_loop.addr				
L1 Hits	1111	1104	1111	1081
L1 Misses	104	111	104	134
L2 Hits	-	-	96	126
L2 Misses	-	-	8	8
Cycles	115110	122040	28710	31410
fourier.addr				
L1 Hits	358164	358414	358164	358133
L1 Misses	431	181	431	462
L2 Hits	-	-	402	433
L2 Misses	-	-	29	29
Cycles	4012640	3765140	3650840	3653630
heapsort.addr				
L1 Hits	975926	1033707	975926	979993
L1 Misses	144687	86906	144687	140620
L2 Hits	-	-	121533	117064
L2 Misses	-	-	23154	23556
Cycles	154446260	97243070	45066560	45062330
small_set.addr				
L1 Hits	5	5	5	4
L1 Misses	4	4	4	5
L2 Hits	-	-	0	1
L2 Misses	-	-	4	4
Cycles	4050	4050	4050	4140
small_L2_set.addr				
L1 Hits	2	4	2	4
L1 Misses	6	4	6	4
L2 Hits	-	-	3	1
L2 Misses	-	-	3	3
Cycles	6020	4040	3320	3140

In order to find the CPI (Cycle per Instruction) for each test result (with a Base CPI of 1) the following formulas were used:

L1 Cache: $CPI = Base\ CPI + \frac{(10)(L1\ Hits) + (1000)(L1\ misses)}{No.of\ Instructions}$

L1 and L2 Cache: $CPI = Base\ CPI + \frac{(10)(L1\ Hits) + (100)(L2\ Hits) + 1000(L2\ Misses)}{No.of\ Instructions}$

CPI for each configuration	**L1 Cache**		**L1 Cache and L2 Cache**	
	16 Blocks 16 Bytes/Block	16 Blocks 32 Bytes/Block	L1 : 16 Blocks 16 Bytes/Block L2: 64 Blocks 64 Bytes/Block	L1 : 8 Blocks 32 Bytes/Block L2: 64 Blocks 64 Bytes/Block
assignment.addr	170	96	47	46
matmul5x5_loop.addr	95	101	24	26
fourier.addr	12	11	11	11
heapsort.addr	138	87	41	41
small_set.addr	451	451	451	461
small_L2_set.addr	753	506	416	393

Performance Ratios for assignment.addr	**L1 Cache**		**L1 Cache and L2 Cache**	
	16 Blocks 16 Bytes/Block	16 Blocks 32 Bytes/Block	L1 : 16 Blocks 16 Bytes/Block L2: 64 Blocks 64 Bytes/Block	L1 : 8 Blocks 32 Bytes/Block L2: 64 Blocks 64 Bytes/Block
16 Blocks 16 Bytes/Block	1	0.5647…	0.2764…	0.2705…
16 Blocks 32 Bytes/Block	1.7708…	1	0.6811…	0.4791…
L1 : 16 Blocks 16 Bytes/Block L2: 64 Blocks 64 Bytes/Block	3.617…	2.0425…	1	0.9787…
L1 : 8 Blocks 32 Bytes/Block L2: 64 Blocks 64 Bytes/Block	3.6956…	2.0869…	1.0217…	1

From the results it can be deduced that larger cache size leads to faster data transfer, which results in better CPU performance. It also decreases the number of misses.

Increasing block size should reduce the miss rate due to spatial locality. On the contrary, because of pollution, *very* large block sizes could potentially lead to an increased miss rate. In our findings, it can be observed that the higher the bytes/block, the lower the miss rate (i.e. inversely proportional.)

Lowering the miss rate is of vital importance as it makes a huge difference to the time taken to process the list of instructions. If there is a hit, the CPU proceeds as normal. However, a cache miss results in the CPU pipeline being stalled, the block having to be fetched from the next level of memory hierarchy, and the instruction fetch having to be restarted.

Evaluation

Which configuration generally performs best?

Generally the L1 cache with 16 Blocks and 16 Bytes/Block with L2 cache: 64 Blocks and 64 Bytes/Block has the fewest number of CPU cycles, and the other L1L2 cache configuration follows closely behind (L1 : 8 Blocks, 32 Bytes/Block ; L2: 64 Blocks, 64 Bytes/Block)

Given that L1 cache is much more expensive than L2 cache, what are the tradeoffs between them?

If the price of caches didn't matter, it would definitely be more beneficial to have more levels of cache (ie increasing the number of caches would boost performance.) However, if cost were an issue (as it is for the majority of home users and businesses,) the tradeoff would be to purchase sufficient primary cache, but leave the rest of the budget for L2 cache because of it's ability to be ''shared'' among multiple cores.

L2 exists to service an L1 cache miss. If the size of L1 was the same/bigger than the size of L2, then L2 could not accommodate for more cache lines than L1, and would not be able to deal with L1 cache misses.

The purpose of caching is to speed up access to the slower hardware by adding a medium that has improved performance, and yet is cheaper than the faster hardware. It prevents the main memory from slowing down the performance of the faster hardware.

If L1 cache is faster, what other purpose does L2 serve besides caching the primary cache's misses? L1 cache is bound to a single core. This means that increasing the L1 size by a fixed quantity will have that cost multiplied by 4 in a dual-core processor, or by a factor of 8 in a quad-core processor. Meanwhile L2 has the ability to be shared among some, if not all, cores of the processor, therefore the cost of increasing L2 would be less.

For a fixed size L1 cache, do you recommend increasing block size or the number of blocks? Does this change for different configurations?

I would increase the number of blocks, as the results have shown that an increase in block number lead to a decrease in CPI. This does not change for the different configurations.

Future Work

The way in which I wrote my code is not suitable for expansion. I have 2 methods specifically for L1 and L1L2 but to extend this to N-levels of cache would involve a very complex and lengthy rewriting process. To allow this application to test more than 2 cache's, I would redesign the program with OOP principles (with each cache as an object.) Then a system could be implemented with more caches (L3, L4, etc) as well as different cache types being tested (such as a LRU cache.)