Multilevel Direct-Mapped Cache Simulator

Ntuthuko Mthiyane (MTHNTU003)

1. Introduction (Code overview)

The aim of this experiment is to implement a multilevel direct-mapped cache that supports a read operation for a fictional machine with a 16-bit address space. The cache simulator should be configurable to support any number of blocks and any block size. The simulator should read addresses from a given file, it should be able to run from the command line with at least one command line argument which is the test set.

"A cache is a hardware or software component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere" - Wikipedia.

2. Technical Overview

This code consists of 3 separate classes, all of these classes work together to form a multilevel cache (as discussed on the introduction) with each role playing a unique role. The classes are, SingleL1Cache class, MultilevelCache class, and Main class. The role of each class and its methods are discussed below:

2.1. SingleL1Cache

This class represents a single L1 cache configuration object, this class is used in the simulator (by the main class) when a user selects a single L1 cache configuration. Once the user has specified the cache size (block size and byte per block), those values are passed to this object which does the following (method names highlighted in blue).

When the object is created, the constructor **SingleL1Cache (int numberOfBlocks, int bytesPerBlock)** is executed. The constructor is responsible for calculating the number of bit's required to represent the offset of the address, the amounts of bits for the index and the amount of bit for the tag. It then creates an array representation of the cache, it sets the size of the array to be equal to the number of blocks and initializes the array by inserting zero's (which represent invalid data) in each block.

The method **hexToBin(String s)** is used in the simulator to convert an address that has been read from file from hexadecimal representation to a 16 bits binary representation.

The method **contentChecker1(ArrayList<String> addresses)** plays a vital role in the simulator, it is used to check if the given memory address has been loaded at a particular index in the cache yet. It gets the index and tag of the particular memory address and compares the tags at the index and increments the number of misses or hits.

The method **getCycles()** is used to calculate the total number of cycles it took to read the given instructions.

getHits() is used to get the number of L1 cache hits for the given instructions.

getMisses() is used to get the number of memory fetches there were for the given instructions

getTag(String address) is used to extract the bits that represent the tag for a given memory address and return this as a string.

getIndex(String address) is used to extract the bits that represent the index in the given memory address, convert the bits from binary to decimal and return a decimal representation.

2.2. MultilevelCache

This class is similar to the SingleL1Cache, with just minor differences. It represents a multi-level L1, L2 cache configuration object, this class is used in the simulator (by the main class) when a user selects a multilevel cache configuration. Most methods are similar to those explained in **2.1**, only methods with additional functionality will be explained here, method names highlighted in blue just like before.

The constructor **MultiLevelCache(int numberOfBlocks1, int bytesPerBlock1, int numberOfBlocks2, int bytesPerBlock2)**, used to determine the size of both L1 and L2 caches and set but their array representation. The rest is similar to that of the singleL1Cache constructor**.**

contentChecker1(ArrayList<String> addresses) is used to check for content on the L1 cache and if content is not found on the L1 cache it is than checked at the lower level L2 cache.

2.3. Main

This class is used in the simulator to play a role of being a driver class, it is the one ran on the command-line. A user specifies on this class which configuration they want to use, the cache sizes, and the instructions to be read. It then reads the instructions from the file and selects which of the above classes to run, it is also responsible for displaying the results.

3. Results and analysis

When the simulator was ran with the different configurations, the results were recorded and analysed. The results were recorded in the following tables (CPI was calculated by dividing cycles by number of instructions).

The following table is for a single-level cache configuration with L1: 16 blocks and 16 bytes per block,

File Name extension = addr	No. of Instructions	No. of L1 Hits	No. of L1 Misses	Cycles	Estimated effective CPI
assignment	71602	60098	11504	12104980	170.06
fourier	358595	358164	431	4012640	12.19
heapsort	1120613	975926	144687	154446260	138.82
matmul5x5_loop	1215	1111	104	115110	95.74
small_L2_set	8	2	6	6020	753.50
small_set	9	5	4	4050	451.00

The following table is for a single-level cache configuration with L1: 16 blocks and 32 bytes per block,

File Name extension = addr	No. of Instructions	No. of L1 Hits	No. of L1 Misses	Cycles	Estimated effective CPI

assignment	71602	65420	6182	6836200	96.47
fourier	358595	358414	181	3765140	11.50
heapsort	1120613	1033707	86906	97243070	87.78
matmul5x5_loop	1215	1104	111	122040	101.44
small_L2_set	8	4	4	4040	506.00
small_set	9	5	4	4050	451.00

The following table is for a multi-level cache configuration with L1: 16 blocks and 16 bytes per block, and L2: 64 blocks and 64 bytes per block.

File Name extension = addr	No. of Instructions	No. of L1 Hits	No. of L1 Misses	No. of L2 Hits	No. of L2 Misses	Cycles	Estimated effective CPI
assignment	71602	60098	11504	9728	1776	3349780	47.78
fourier	358595	358164	431	402	29	3650840	11.18
heapsort	1120613	975926	144687	121533	23154	45066560	41.22
matmul5x5_loop	1215	1111	104	96	8	28710	24.63
small_L2_set	8	2	6	3	3	3320	416.00
small_set	9	5	4	0	4	4050	451.00

The following table is for a multi-level cache configuration with L1: 8 blocks and 32 bytes per block, and L2: 64 blocks and 64 bytes per block.

File Name extension = addr	No. of Instructions	No. of L1 Hits	No. of L1 Misses	No. of L2 Hits	No. of L2 Misses	Cycles	Estimated effective CPI
assignment	71602	62446	9156	7234	1922	3269860	46.67
fourier	358595	358133	462	433	29	3653630	11.19
heapsort	1120613	979993	140620	117064	23556	45062330	41.21
matmul5x5_loop	1215	1081	134	126	8	31410	26.85
small_L2_set	8	4	4	1	3	3140	393.50
small_set	9	4	5	1	4	4140	461.00

From the CPI's calculated on the above tables, single L1 cache with 16 blocks and 16 bytes per block was the worst performing (with highest CPI). Performance ratio was calculated relative to it.

File Name extension = addr	L1: 16 blocks, 32 bytes per block	L1: 8 blocks, 32 bytes per block. L2: 64 blocks, 64 bytes per block	L1: 16 blocks, 16 bytes per block. L2: 64 blocks, 64 bytes per block
assignment	1.76	3.64	3.56
fourier	1.06	1.09	1.09
heapsort	1.58	3.37	3.37
matmul5x5_loop	0.94	3.57	3.89
small_L2_set	1.49	1.91	1.81
small_set	1.00	0.98	1.00
Average performance	1.31	2.43	2.45

One thing that can be noticed from the above tables is that increasing the number of bytes per instruction decreases CPI significantly.

4. Evaluation

4.1. From the average performances on the Performance table on step 3, it can be noticed that the multi-level cache configuration with L1: 16 blocks and 16 bytes per block, and L2: 64 blocks and 64 bytes per block generally performs best.

4.2. The trade-offs between are size, speed and money as you cannot optimize all three at the same time. It is either fast and expensive or slow and cheap or large, fast and expensive. This is because of the cost associated with making each cache (fast caches are expensive), ideally we want a fast, large and cheap caches.

4.3 For a **fixed size** L1 cache, it is better to increase number of blocks. Though a larger block size takes advantage of spatial locality, tends to reason in a larger miss penalty. It also increases the miss ratio, since it causes the system to fetch a bunch of extra information that is used less than the data it displaces in the cache [https://www.eecis.udel.edu/~sunshine/courses/F05/CIS662/class21.pdf].

For a fixed size cache (example an 8 KB cache), having larger blocks means we are sacrificing something. Than we will have to have **fewer blocks** which results in having fewer destinations for addresses to map to. This will lead to having more collisions, more competition which means having an increased miss rate/ **increased miss penalty** because we will pull things out of the cache more often. Larger blocks Increases miss penalty. For very large block size, may increase miss rate due to pollution [lecture slides].

This does not change for different configurations, example for a multi-level configuration L1 cache should have a smaller block size (for reasons given above) and more number of blocks. And the L2 cache should have a large block size, so it can take advantage of spatial locality and ensure that the miss rate it maximized.

5. Future work

To expand the simulator to support an **n**-way associative cache instead of just a directed-mapped cache, all of the classes with have to be slightly changed in one way or another. The changes for each class will be explained below.

SingleL1Cache class

For the singleL1Cache class to be n-ways associative, a few of their methods will have to be altered slightly. Firstly, the constructor parameters will have to be changed, an additional argument will have to be accepted, this variable N representing the number of sets in the particular cache. Still on the constructor, the way in which the cache is initially created will have to be changed to a 2D array (M*N array, where M is number of blocks and N is number of sets).

Method **contentChecker1(ArrayList<String> addresses)** will also have to change, a random replacement policy will have to be implemented in this method when. The way the search will happen will have to change, there will have to be N- searches now in each block.

The **getIndex(String address)** and **getTag(String address)** methods will have to change as the way to get this values changes with different configurations.

The rest of the methods in the class will remain the same.

MultilevelCache class

MultiLevelCache is similar to singleL1Cache, so they have very similar methods. All of their changes will be the same for the constructor, **getIndex(String address)** method and **getTag(String address)** method, the only difference will be in the be on the **contentChecker1(ArrayList<String> addresses)** method.

contentChecker1(ArrayList<String> addresses) will differ by creating two 2D arrays (for L1 and L2), the replacement policy will also be changed to be random, and searches for each block will be done N times. The rest will remain the same.

Main class class

The main class does not contain any methods, so the only changes that will happen is adding addition variable names. For representing number of sets, the way in which the objects are created will have to be changed as the objects constructors have now been changed.