The University of Texas at San Antonio

Computer Architecture

Report on a Cache Simulator

Submitted by

Group 25

Members

Tanzira Najnin

Date: 30/11/2019

Introduction

A cache is a special high-speed storage mechanism. It is used to improve the average memory access time of slow memory. In computer science, almost everything is a cache. There can be multiple levels of cache and the upper-level cache is the costliest one and the lower levels are cheaper. These days most processors have three levels of caches. They are L1, L2, and L3 caches. There are three types of cache placement policies. For this project set-associative policy has been adapted. As the cache size is small we can not put everything into the cache. So sometimes we need to replace a block of data with new data when there is no free space in the cache. There are different types of cache replacement policies but among them, LRU (Least Recently Used) replacement policy is mostly used. Here in this cache simulator project, for replacement LRU is used and write policy is writeback.

Methodology

To discuss the project in brief, at first __init__() function is called and it initializes all the parameter needed for set associative cache. Then next_frame() function is called with a hexadecimal address. This function then sends the address into translate() function. The translate() function then divides the address into three parts (tag, set_index, offset) and returns them. Then is_hit() function gets a call and checks if the address is a hit or not. If it's a hit then it increases the hit count. If it's not a hit then get_free_frame() function gets call to check if there are any free frame in the set. If there is then it returns the frame number or returns -1. If the frame number is -1 that means no free frame so some frame needs to be replaced. For the replacement policy LRU has been used. So then next_frame() will call get_overwrite_frame(). This function will return the frame number to be overridden. Then next_frame() function calls the place in cache for putting the address into cache and record() function for maintaining the cache replacement policy. Record function basically enqueues the frame into a queue. There is a dirty bit in cache that keeps track of a frame being dirty. If it is dirty then its marked as 1, otherwise 0. And if there is any write operation then next_frame() will call write_back() function. This function writes the data back if any frame is being overridden in cache. Then there is a function for getting the statistics for hit rate and miss rate. The function is named as get_stats(). This will return the hit rate and miss rate.

This project is implemented using the language Python. The version is python 3. There are different methods that implement all the requirements of the project. They are described below:

Class Name: *Cache*

__init__(cache_size, block_size, n, debug, policy)

Arguments

cache_size	(int) Size of the cache in bytes
block_size	(int) Size of each block in the cache in bytes
n	(int) Total number of ways for set-associative cache
debug	(boolean) *True/False*

policy	(string) Cache replacement policy. Can be *lru* (least recently used) or *rr* (randomized replacement)
Returns	None

translate(address)

Arguments	
address	(string) hexadecimal memory address
Returns	(int, int, int) tag, set index and offset for the given address

is_hit(address, op)

Arguments	
address	(string) hexadecimal memory address
op	(string) type of operation. Can be *R* (read) or *W* (write)
Returns	(boolean) *True* if the given address is in cache otherwise *False*

get_free_frame(address)

Arguments	
address	(string) hexadecimal memory address
Returns	(int) frame number if a free slot is available in the corresponding set, otherwise returns -1.

get_overwrite_frame(address)

Arguments	
address	(string) hexadecimal memory address
Returns	(int) frame number to be overwritten according to replacement policy.

write_back(frame, address)

Arguments	
frame	(int) frame to be replaced
address	(string) hexadecimal memory address
Returns	None.

record(address)

Arguments	
address	(string) hexadecimal memory address.
Returns	None. Stores address in a data structure associated with the replacement policy.

place_in_cache(address, free_frame, op)

Arguments	
address	(string) hexadecimal memory address.
free_frame	(int) the frame which will be written.
op	(string) type of operation. Can be *R* (read) or *W* (write)

Returns None. Stores address in cache.

next_frame(address, op)

Arguments
address (string) hexadecimal memory address
op (string) type of operation. Can be *R* (read) or *W* (write)
Returns None.

get_stats()

Returns (float, float) cache hit and miss rate.

Usage

A Separate bash script calls the CacheSimulator.py file and takes a filename and other optional arguments to configure the cache simulator for different settings. The command for running the simulator is:

```
./run_sim.sh 1KB_64B --cache_memory 1KB --block_size 64B --way 16
```

Except for filename other parameters are optional. The default configuration is:
--cache_memory = 1MB
--block_size = 64B
--way = 16
There is an option for help. --help/-h/-help brings out all the command line parameters that can be given for the configuration.

Experiment and Set-Up

For Experimentation, different configurations were tested. Cache size 1KB and extended till 512KB. For set-associative cache, I tried 5 different ways. The 5 different ways are 1, 2, 4, 8 and 16. I have incremented the cache size with a power of 2 and the same for the ways. The block size is fixed here and that is 64B.

To validate the replacement policy that I have used here I compared it with one other replacement policy as well. So with Least Recently Used replacement policy I have also implemented a Randomized replacement policy. Then I have simulated both of the policies with the same text file. For this set-up, different settings were tried as well to see if one performs better than the other in certain settings or not.

For checking the effect of way and cache "*gcc.trace.txt*" file has been used. For checking which replacement policy works better "*openblas_dgemm.trace.txt"* has been used.

Results and Discussions

The graph below shows the effect of several ways on cache size.

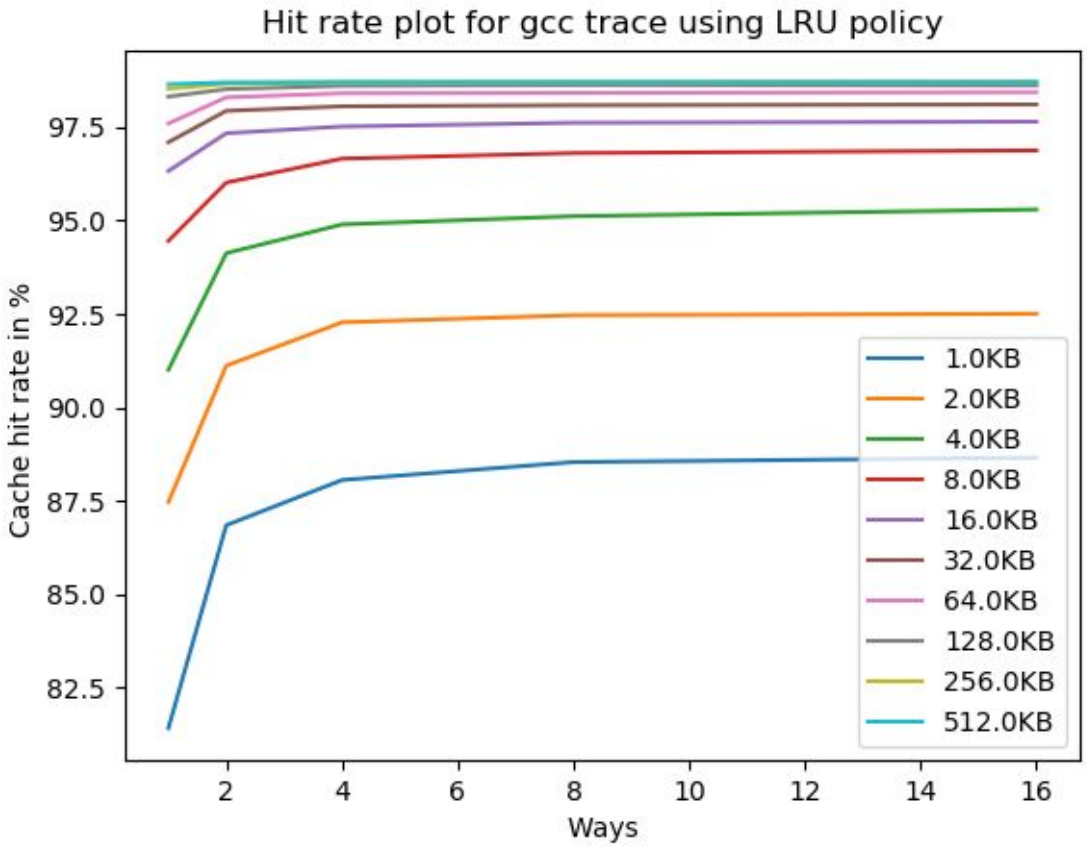

Figure: 1

From figure 1, we can see that after 4 ways there is not much of an improvement in the cache hit rate. If we look at cache size then we can see that after 128KB even if we are increasing cache or ways the accuracy remains almost the same. So for *GCC trace* best configuration would be cache size 256KB and way 4. Because after that increasing the way is kind of pointless.

So after analyzing this figure, we can say that only increasing the way can not solve the miss rate problem.

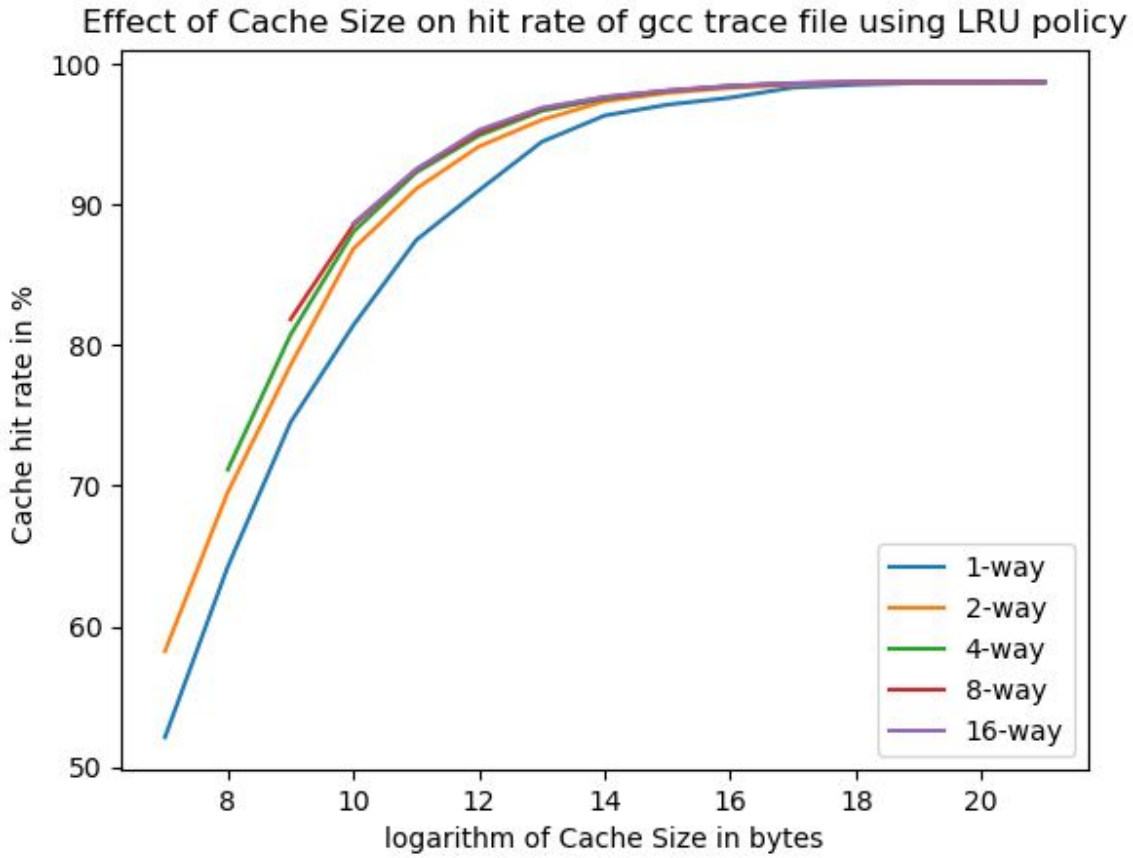

Figure: 2

Now if we look at figure 2, we can see the effect of cache size on ways. At first, way is dependent on cache size. Because the total number of sets in the set-associative cache is dependent on cache size and block size. If the cache size is not big enough we can not have a total number of sets more than a certain number (*cache size/block size * way*). So for this second experiment for a different number of cache size I have different numbers of ways. From analyzing the figure we can see that when the cache size is around 256KB it has no effect on the hit rate. It is almost the same even if we go beyond that size.

So from this experiment, we can conclude that cache hit does not only depend on cache size.

Again I have tested the replacement policy that works better. I have implemented the *Least Recently Used* replacement (LRU) Policy and *Randomized Replacement* (RR) policy. The table for hit rate with different cache size and ways are given below for both of the policies. For this experimentation, I have used "*openblas_dgemm.trace.txt*" file.

Cache Size (KB)	***Associativity (LRU)***					***Associativity(RR)***				
	1-way	**2-way**	**4-way**	**8-way**	**16-way**	**1-way**	**2-way**	**4-way**	**8-way**	**16-way**
1	**46.55**	**37.25**	**38.28**	**38.24**	**38.22**	32.83	31.46	34.25	36.12	37.0
2	**52.91**	**59.11**	**56.55**	**60.16**	**60.2**	36.41	44.89	43.9	46.19	47.07
4	**58.8**	**64.24**	**67.33**	**67.07**	**67.3**	39.92	48.42	54.94	54.62	56.07
8	**62.74**	**66.1**	**67.58**	**67.6**	**67.66**	42.22	50.99	57.48	61.35	61.35
16	**68.43**	**67.15**	**67.6**	**67.67**	67.75	50.92	56.78	62.53	66.16	**68.08**
32	**83.04**	**81.57**	**86.56**	**83.67**	77.87	72.98	72.91	75.48	77.32	**78.41**
64	**87.2**	**90.79**	**90.56**	**90.33**	**90.33**	78.94	82.75	84.36	85.4	85.99
128	**89.39**	**91.56**	**91.63**	**91.64**	**91.64**	82.23	85.61	87.6	88.54	89.04
256	**90.64**	**91.66**	**91.7**	**91.7**	**91.7**	84.21	86.93	88.84	89.82	90.32
512	**91.17**	**91.7**	**91.7**	**91.7**	**91.7**	86.09	88.76	90.37	91.25	91.68

Table: 1

From the above experimentation, we can see that LRU performance is not much increasing after 128KB cache size. For the memory and way, it is taking the outcome is not that significant. On the other hand, if we look at the other table for RR policy as we are increasing the way the output is getting better. It is the same for the cache as well. As bothe cache size and way are increasing the output keeps getting better. The more cache size we provide for RR the less cache miss will occur because it is choosing a frame randomly.

So we can say that from these two Least Recently used policy will be better. Because if one file is being used then in future the probability is higher that the same file will be used again.

After seeing the effect of both cache and number ways we can say that after a certain point the accuracy will be the same. And if we increase the cache size more then the latency will increase too. So we will have to come to a balanced situation where we can have both less miss rate and less latency. When the cache size is small the replacement policy helps the best in reducing the miss rates. If we had a perfect replacement policy like if we could see future then cache miss rates would be much lesser. These experiments help us in determining those balanced cases that increase hit rates.