ES215 Computer Organization and Architecture
Project Report
Cache Simulator

Aryan Gupta
aryan.gupta@iitgn.ac.in
20110026

Ruchit Chudasama
ruchit.chudasama@iitgn.ac.in
20110172

Rahul Lalani
rahul.lalani@iitgn.ac.in
20110154

26 APRIL, 2022

Contents

Abstract

We have implemented a cache simulator for a single level cache hierarchy from scratch in python.

Inputs:

- Cache size (in Bytes)
- Block size (in Bytes)
- File containing memory traces (each entry containing 8-digit Hex-decimal number).
- Associativity

Outputs:

- A csv file A csv file containing 26 parameters for all kinds of cache configuration out of which some have been listed below:
  - Number of tag bits
  - Number of Cache Blocks
  - Number of Cache Accesses
  - Number of Read Accesses
  - Number of Write Accesses
  - Number of Cache Misses
  - Number of Compulsory Misses
  - Number of Capacity Misses
  - Number of Conflict Misses
  - Number of Read Misses
  - Number of Write Misses
  - Number of Dirty Blocks Evicted
  - Program Execution Time

  - Number of NAND Gates
  - Total Hit Rate

- A separate window displaying the Total Cache Misses, Total Execution Time for various cache configurations and breakdown of instruction types

Introduction

2.1 The goal of this project

The objective of this course project is to simulate the behaviour of a cache hardware. This project aims to study a single-level cache hierarchy system where the user can configure the parameters of the cache such as Cache Size, Block Size and Associativity as well as input the program trace file and outputs the best cache mapping and cache policy for a given cache configuration and trace file. The output will be decided by analysing the program execution time corresponding to different cache mapping and policies. We also hope to visually plot bar charts and pie graphs and a table containing all the parameters such as cache hits, cache misses, read hits, write hits, read misses, write misses corresponding to different cache mappings and policies.

2.2 The rationale of the project

We were always mesmerised to see a web page taking lesser time to load if it had been recently opened. We observed similar behaviour for profile pictures on Whatsapp. This made us keen to explore how an individual cache is managed, what data to bring, where it gets placed and what happens if it is already present in the cache when we do future look ups.

2.3 The importance of the project

Cache memory is important because it improves the efficiency of data retrieval. It stores program instructions and data that are used repeatedly in the operation of programs or information that the CPU is likely to need next. Via this project, we can analyse which cache mapping and policy is suitable for a particular cache organization which overall will increase the efficiency of the computer processor. We can also understand how the memory accesses are made while running programs on the processor which ultimately will help us to optimize the execution time of running a computer program.

2.4 Our contributions to the project

There are multiple cache mappings and cache policies which can give a variety of combinations to choose from. We will be designing our cache simulator which will output various performance metrics such as additional hardware used, program execution time, hit rate etc. for all kinds of cache configuration based on which the user can decide which kind of cache is suitable. for a particular program corresponding to a fixed cache configuration and a computer program.

Literature Review

- We started working on the project before the topic was covered in the class. So, we took help from online lectures of CS3810 (University of Utah) to get understanding of caches. We also refereed to the lecture slides to get insights into the topic.
- We faced some challenges while we were working on the code. We asked to Prof. Sameer Kulkarni whenever we had trouble in understanding some concepts. For example, we had doubts regarding the writing policies and also regarding the miss penalties for different policies. These doubts were regularly addressed and were cleared by the faculty.
- We minutely studied "Paracache", an existing web-based simulator. Studying this simulator made us understand how a cache simulator works and indeed laid a solid foundation on which we could build our own cache simulator from scratch.

Project Idea

The growing difference between processor and memory speeds makes the cache memory and its efficient utilization a crucial factor in determining program performance. With increasing demand in computation and large set of data involved, understanding the memory mapping techniques would be the baseline for student to further improve efficiency in resources utilization.
Since there are multiple cache mappings and various writing and replacement policies, we aim to design a cache simulator which outputs various performance metrics such as additional hardware used, program execution time, hit rate etc. for all kinds of cache configuration based on which we can comprehend which kind of cache is optimal for a particular program corresponding to a fixed cache configuration and a computer program.

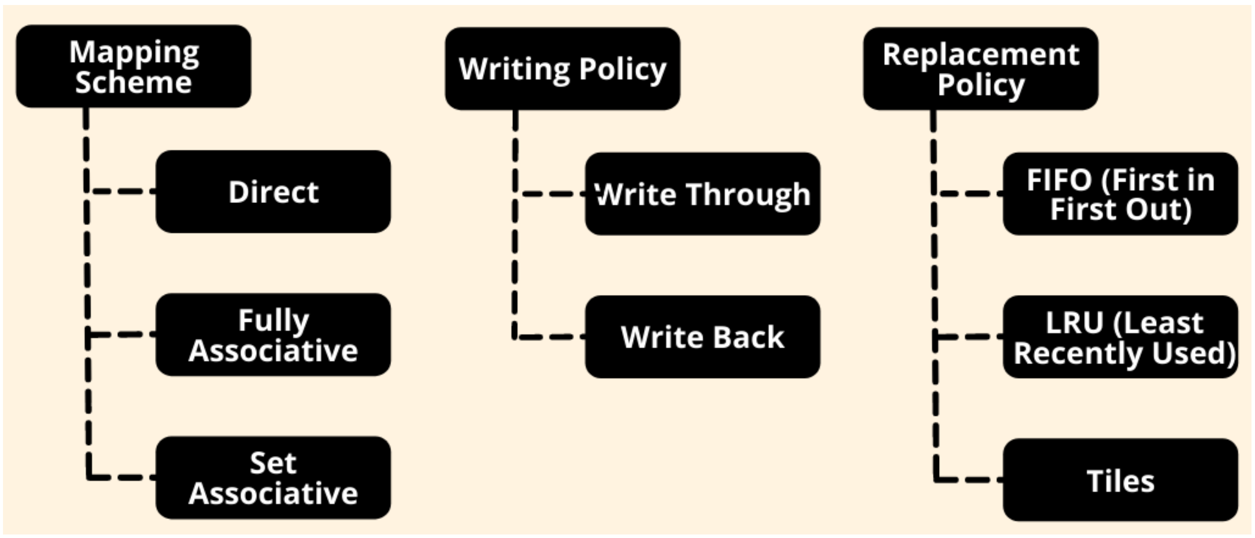


Figure 4.1: Cache Mapping and Cache Policies

Assumptions

After doing an intensive literature review, we felt that it would not be feasible to implement each and every detail that we had studied. So, we make the following assumptions:

- We are not concerned with the type of data that we are writing or reading from memory, we are only focusing on the memory LOCATION on which the data is being read/wrote.
- As we are not dealing with the content of memory, so we don't know memory bits. And hence we don't know overhead.
- The trace file that we have considered has a specific format. The trace file will specify all the data memory accesses that occur in the sample program. Each line in the trace file will specify a new memory reference.
  Each line in the trace cache will therefore have the following three fields:

  Access Type: A single character indicating a load ('l') or a store ('s') instruction.

  Address: A 32-bit integer (in unsigned hexidecimal format) specifying the memory address that is being accessed. For example, "0xff32e100" specifies that memory address 4281524480 is accessed.

  Instructions since last memory access: Indicates the number of instructions of any type that executed between since the last memory access (i.e. the one on the previous line in the trace). For example, if the 5th and 10th instructions in the program's execution are loads, and there are no memory operations between them, then the trace line for with the second load has "4" for this field.
- We have assumed write allocate policy for cache write miss.
- If there is cache hit, then it happens in 1 cycle.
- If there is cache miss, then the miss-penalty is 5 cycles.
- All the ALU instructions are processed by a pipelined processor and hence they execute in 1 cycle.
- The clock rate is 1 GHz.

Project Implementation

6.1 Programming Language and Libraries Used

The entire code was written in python on Google Collab for seamless collaboration.
The entire code was brainstormed and implemented from scratch.
Matplotlib was used to visually plot program output parameters for clarity.
Tkinter was used to design a basic graphical user interface for taking inputs from the program.

6.2 Meat Portions of the Code

6.2.1 Implementing Direct Mapped Cache with Write Back

```
for i in range(len(master_file)):
    total_alu_instructions+= int(master_file[i][13])
    if master_file[i][0]=='l':
        total_load_instructions+=1
        hex= (master_file[i][4:12])
        binary_string=HexToBin(hex)
        offset = binary_string[-int(offset_bits):]
        tag = binary_string[:int(tag_bits)]
        index = binary_string[int(tag_bits):int(tag_bits)+int(index_bits)]
        if tag_array[int(index,2)] == ' ' :
            tag_array[int(index,2)] = tag
            compulsary_read_miss+=1
            cycles+=miss_penalty
        elif (tag_array[int(index,2)] != tag):
            tag_array[int(index,2)] = tag
            conflict_read_miss+=1
            cycles+=miss_penalty
            if dirty_bit[int(index,2)]==1:
                Dirty_blocks+=1
        else:
            read_hit+=1
            cycles+=1
    else:
        total_store_instructions+=1
        hex= (master_file[i][4:12])
        binary_string=HexToBin(hex)
        offset = binary_string[-int(offset_bits):]
        tag = binary_string[:int(tag_bits)]
        index = binary_string[int(tag_bits):int(tag_bits)+int(index_bits)]
        if tag_array[int(index,2)] == ' ' :
            tag_array[int(index,2)] = tag
            dirty_bit[int(index,2)]=1
            compulsary_write_miss+=1
            cycles+=miss_penalty
        elif (tag_array[int(index,2)] != tag):
            tag_array[int(index,2)] = tag
            conflict_write_miss+=1
            cycles+=miss_penalty
            if dirty_bit[int(index,2)]==1:
                Dirty_blocks+=1
        else:
            write_hit+=1
            cycles+=1
```

This code has been implemented for a Direct Mapped Cache with Write Back Policy

- All the initial output parameters have been set to zero. They have been updated in the for loop as and when required.
- The tag comparisons were made with one cache block only as this is a direct mapped cache.
- For implementing the concept of write back policy, we updated only the cache and updated the main memory at the stage when the block was going to e evicted, thus reducing latency.

6.2.2 Implementing a Fully Associative Cache with Write Through and FIFO Replacement Policy

```
r=0

    for k in range(len(master_file)):
        alu_instructions+= int(master_file[k][13])
        if master_file[k][0]=='l':
            hex=(master_file[k][4:12])
            binary_string=HexToBin(hex)
            offset = binary_string[-int(offset_bits):]
            tag = binary_string[:int(tag_bits)]
            if tag in tag_array:
                read_hit+=1
                cycles+=1
            elif ' ' in tag_array:
                for i in range(len(tag_array)):
                    if tag_array[i]== ' ':
                        tag_array[i]=tag
                        compulsary_read_miss+=1
                        cycles+=miss_penalty
                        break
            else:
                tag_array[r%len(tag_array)]=tag
                r+=1
                capacity_read_misses+=1
                cycles+=miss_penalty
        else:
            hex=(master_file[k][4:12])
            binary_string=HexToBin(hex)
            offset = binary_string[-int(offset_bits):]
            tag = binary_string[:int(tag_bits)]
            if tag in tag_array:
                write_hit+=1
                cycles+=miss_penalty
            elif ' ' in tag_array:
                for i in range(len(tag_array)):
                    if tag_array[i]== ' ':
                        tag_array[i]=tag
                        compulsary_write_miss+=1
                        cycles+=2*miss_penalty
                        break

            else:
                tag_array[r%len(tag_array)]=tag
                r+=1
                capacity_write_misses+=1
                cycles+=2*miss_penalty
```

This code has been implemented for a Fully Associative Cache with Write Through and FIFO Replacement Policy

- All the initial output parameters have been set to zero. They have been updated in the for loop as and when required.
- For implementing the concept of FIFO, we used a dummy variable r to keep track of which cache block is first filled with memory. It is updated by one if a cache miss occurs
- The tag comparisons were made corresponding to each cache block as this is a fully associative cache.
- For implementing the concept of write through policy, we updated the main memory

as soon as we write to a cache and this introduces extra latency, thus reducing the total program execution time.

6.2.3 Implementing a Set Associative Cache with Write Back and LRU Replacement Policy

```
for k in range(len(master_file)):
    alu_instructions+= int(master_file[k][13])
    if master_file[k][0]=='l':
        hex=(master_file[k][4:12])
        binary_string=HexToBin(hex)
        offset = binary_string[-int(offset_bits):]
        tag = binary_string[:int(tag_bits)]
        index = binary_string[int(tag_bits):int(tag_bits)+int(index_bits)]
        array_new=tag_array[int(index,2)]
        if tag in array_new:
            m=array_new.index(tag)
            index_array_all[int(index,2)].remove(m)
            index_array_all[int(index,2)].insert(0,m)
            read_hit+=1
            cycles+=1
        elif ' ' in array_new:
            for i in range(len(array_new)):
                if array_new[i]== ' ':
                    array_new[i]=tag
                    index_array_all[int(index,2)].remove(index_array_all[int(index,2)][-1])
                    index_array_all[int(index,2)].insert(0,i)
                    cycles+=miss_penalty
                    compulsary_read_miss+=1
                    break
        else:
            array_new[index_array_all[int(index,2)][-1]]=tag
            m=index_array_all[int(index,2)][-1]
            index_array_all[int(index,2)].remove(index_array_all[int(index,2)][-1])
            index_array_all[int(index,2)].insert(0,m)
            capacity_read_misses+=1
            if dirty_bit[int(index,2)][i]==1:
                Dirty_blocks+=1
            cycles+=miss_penalty
    else:
        hex=(master_file[k][4:12])
        binary_string=HexToBin(hex)
        offset = binary_string[-int(offset_bits):]
        tag = binary_string[:int(tag_bits)]
        index = binary_string[int(tag_bits):int(tag_bits)+int(index_bits)]
        array_new=tag_array[int(index,2)]
```

```
if tag in array_new:
    m=array_new.index(tag)
    dirty_bit[int(index,2)][array_new.index(tag)]=1
    index_array_all[int(index,2)].remove(m)
    index_array_all[int(index,2)].insert(0,m)
    write_hit+=1
    cycles+=1
elif ' ' in array_new:
    for i in range(len(array_new)):
        if array_new[i]== ' ':
            array_new[i]=tag
            index_array_all[int(index,2)].remove(index_array_all[int(index,2)][-1])
            index_array_all[int(index,2)].insert(0,i)
            cycles+=miss_penalty
            compulsary_write_miss+=1
            dirty_bit[int(index,2)][i]=1
            break
else:
    array_new[index_array_all[int(index,2)][-1]]=tag
    m=index_array_all[int(index,2)][-1]
    index_array_all[int(index,2)].remove(index_array_all[int(index,2)][-1])
    index_array_all[int(index,2)].insert(0,m)
    capacity_write_misses+=1
    if dirty_bit[int(index,2)][i]==1:
        Dirty_blocks+=1
    cycles+=miss_penalty
```

This code has been implemented for a Set Associative Cache with Write Back and Cache Replacement Policy

- All the initial output parameters have been set to zero. They have been updated in the for loop as and when required.
- For implementing the concept of LRU, we used a queue to keep track of the accesses in which the cache blocks were accessed.
- The tag comparisons were made corresponding to each set as this is a set associative cache.
- For implementing the concept of write back policy, we assigned each cache blocks an initial dirty bit of zero and later updated it to 1 when the data was updated in the cache but not in the main memory.

6.3 Challenges

- One of the major challenges we faced while implementing the project was the lack of knowledge regarding the various aspects of Cache. Since, the topic was yet to be covered in the class, we familiarized ourselves with the various aspects of a cache on our own.

- Another challenge we faced was that since we were a group of only three members, everyone had to did a little more work as compared to other group's members in the project.
- We also faced a challenge while implementing the LRU cache mapping. After a lot of brainstorming and web search, we decided to use the concept of queue to implement LRU.
- We tried to verify the results of our cache simulator using the online cache simulator "Paracache", but it was crashing for mere 50,000 memory references. Just to compare the coverage of the simulators, our code of cache simulator is running for at least 3 million memory references.

Testing and Experiments

- Open the 'cache.py' file. On running this file a pop up window will appear which will ask you to select a trace file.
- After selecting the trace file dialogue boxes will appear where we will have to enter cache size,block size and associativity.
- Now wait for half a minute for simulator to run .
- After the program has been run , the plots showing execution time , cache misses and the distribution of instructions type will appear in a separate window .
- Also a csv file named `"cache_parameters.csv"` will be automatically generated in the D drive.
- On the console, a conclusive statement will appear telling the best execution time and to what cache organization it corresponds to.
- The file will have 25 parameters for 10 different types of cache organization .

The written code was run several times and Matplotlib, a python library was used to visually plot the data. The data trends were observed and were validated and they were in accordance with what we have been taught in the course.

Data Analysis

An extensive and comprehensive analysis was performed on data which was visually stored in the form of bar graphs and pie charts and additionally all the cache parameters were stored in a csv file. The following observations were made:

- Higher associativity improves the hit-rate but negatively affects the complexity of the architecture in terms of number of NAND gates used.
- Higher associativity increases the miss-penalty, thus adversely affecting average memory access time of the system.
- From this analysis we can predict that in computers that we use, higher associativity is implemented in L1 Cache, which will have relatively small capacity.
- Direct map is implemented in L2 cache.
- An approximation overview of cost versus performance need to be simulated in order to give deeper understanding on the trade-offs of cache types.

From this analysis we can predict that in computers that we use nowadays:

- Higher associativity is implemented in L1 Cache, which will have relatively small capacity.
- Direct map is implemented in L2 cache.
- An approximation overview of cost versus performance need to be simulated in order to give deeper understanding on the trade-offs of cache types.

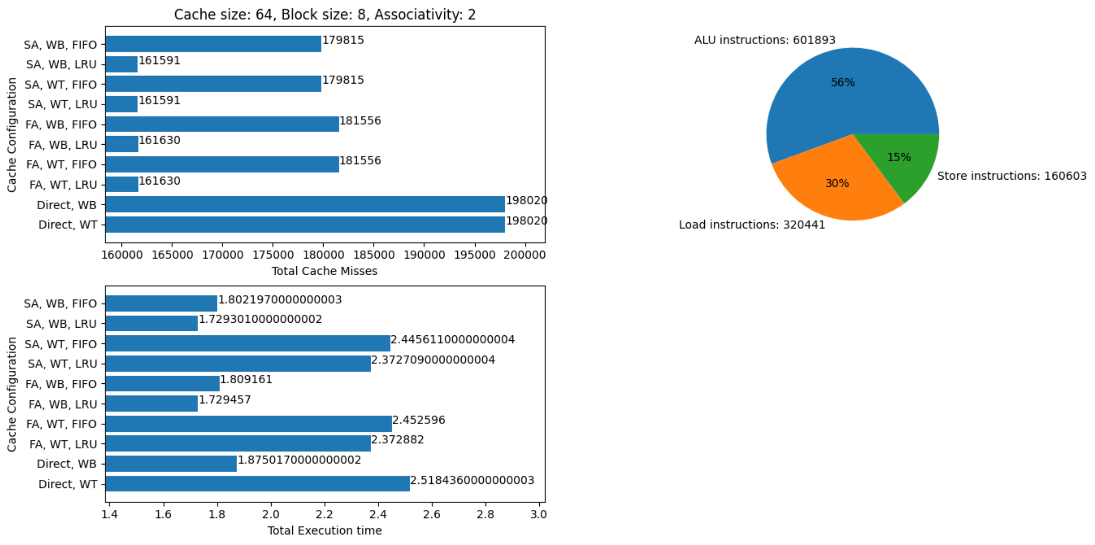
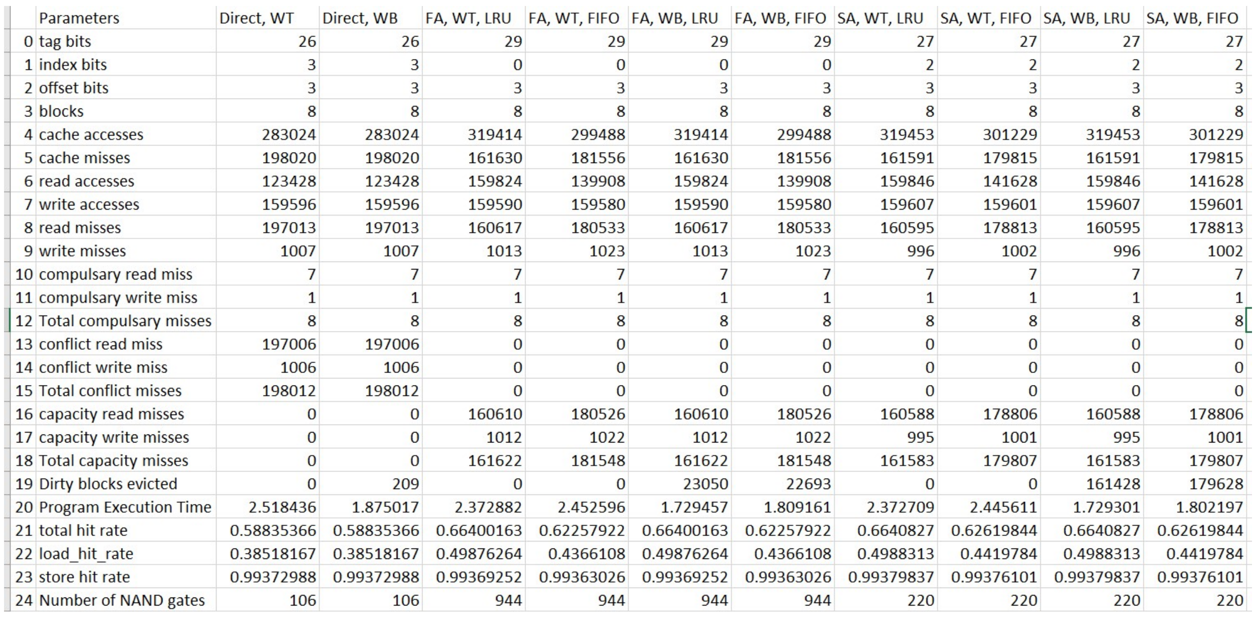


Figure 8.1: Observations for small cache size



	Parameters	Direct, WT	Direct, WB	FA, WT, LRU	FA, WT, FIFO	FA, WB, LRU	FA, WB, FIFO	SA, WT, LRU	SA, WT, FIFO	SA, WB, LRU	SA, WB, FIFO
0	tag bits	26	26	29	29	29	29	27	27	27	27
1	index bits	3	3	0	0	0	0	2	2	2	2
2	offset bits	3	3	3	3	3	3	3	3	3	3
3	blocks	8	8	8	8	8	8	8	8	8	8
4	cache accesses	283024	283024	319414	299488	319414	299488	319453	301229	319453	301229
5	cache misses	198020	198020	161630	181556	161630	181556	161591	179815	161591	179815
6	read accesses	123428	123428	159824	139908	159824	139908	159846	141628	159846	141628
7	write accesses	159596	159596	159590	159580	159590	159580	159607	159601	159607	159601
8	read misses	197013	197013	160617	180533	160617	180533	160595	178813	160595	178813
9	write misses	1007	1007	1013	1023	1013	1023	996	1002	996	1002
10	compulsary read miss	7	7	7	7	7	7	7	7	7	7
11	compulsary write miss	1	1	1	1	1	1	1	1	1	1
12	Total compulsary misses	8	8	8	8	8	8	8	8	8	8
13	conflict read miss	197006	197006	0	0	0	0	0	0	0	0
14	conflict write miss	1006	1006	0	0	0	0	0	0	0	0
15	Total conflict misses	198012	198012	0	0	0	0	0	0	0	0
16	capacity read misses	0	0	160610	180526	160610	180526	160588	178806	160588	178806
17	capacity write misses	0	0	1012	1022	1012	1022	995	1001	995	1001
18	Total capacity misses	0	0	161622	181548	161622	181548	161583	179807	161583	179807
19	Dirty blocks evicted	0	209	0	0	23050	22693	0	0	161428	179628
20	Program Execution Time	2.518436	1.875017	2.372882	2.452596	1.729457	1.809161	2.372709	2.445611	1.729301	1.802197
21	total hit rate	0.58835366	0.58835366	0.66400163	0.62257922	0.66400163	0.62257922	0.6640827	0.62619844	0.6640827	0.62619844
22	load_hit_rate	0.38518167	0.38518167	0.49876264	0.4366108	0.49876264	0.4366108	0.4988313	0.4419784	0.4988313	0.4419784
23	store hit rate	0.99372988	0.99372988	0.99369252	0.99363026	0.99369252	0.99363026	0.99379837	0.99376101	0.99379837	0.99376101
24	Number of NAND gates	106	106	944	944	944	944	220	220	220	220

Figure 8.2: Values of cache parameters for small cache size

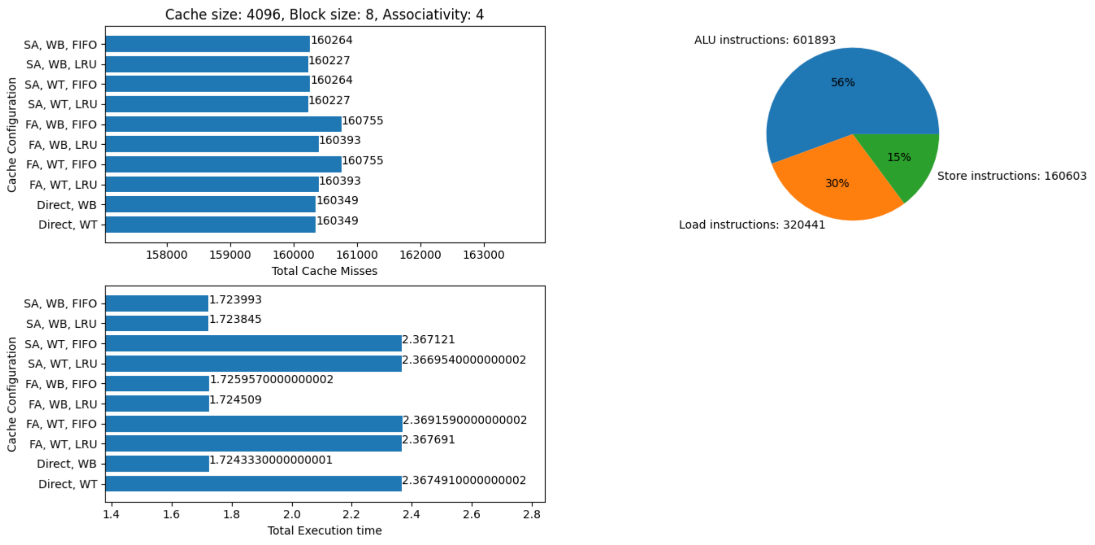
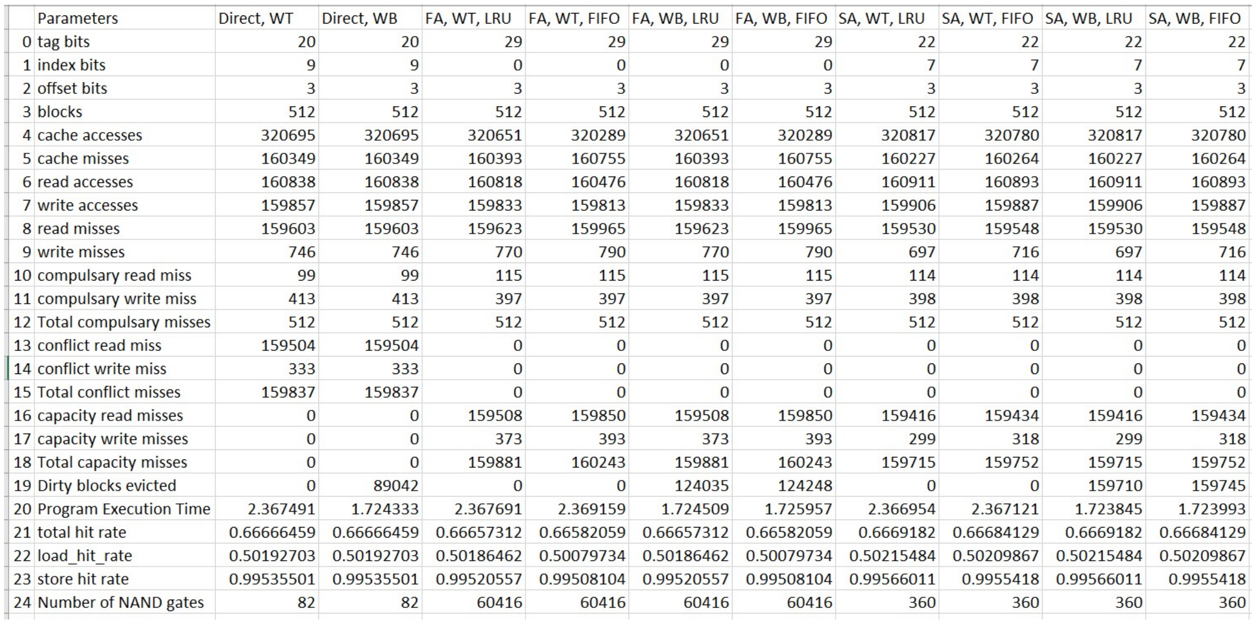


Figure 8.3: Observations for moderate cache size



	Parameters	Direct, WT	Direct, WB	FA, WT, LRU	FA, WT, FIFO	FA, WB, LRU	FA, WB, FIFO	SA, WT, LRU	SA, WT, FIFO	SA, WB, LRU	SA, WB, FIFO
0	tag bits	20	20	29	29	29	29	22	22	22	22
1	index bits	9	9	0	0	0	0	7	7	7	7
2	offset bits	3	3	3	3	3	3	3	3	3	3
3	blocks	512	512	512	512	512	512	512	512	512	512
4	cache accesses	320695	320695	320651	320289	320651	320289	320817	320780	320817	320780
5	cache misses	160349	160349	160393	160755	160393	160755	160227	160264	160227	160264
6	read accesses	160838	160838	160818	160476	160818	160476	160911	160893	160911	160893
7	write accesses	159857	159857	159833	159813	159833	159813	159906	159887	159906	159887
8	read misses	159603	159603	159623	159965	159623	159965	159530	159548	159530	159548
9	write misses	746	746	770	790	770	790	697	716	697	716
10	compulsary read miss	99	99	115	115	115	115	114	114	114	114
11	compulsary write miss	413	413	397	397	397	397	398	398	398	398
12	Total compulsary misses	512	512	512	512	512	512	512	512	512	512
13	conflict read miss	159504	159504	0	0	0	0	0	0	0	0
14	conflict write miss	333	333	0	0	0	0	0	0	0	0
15	Total conflict misses	159837	159837	0	0	0	0	0	0	0	0
16	capacity read misses	0	0	159508	159850	159508	159850	159416	159434	159416	159434
17	capacity write misses	0	0	373	393	373	393	299	318	299	318
18	Total capacity misses	0	0	159881	160243	159881	160243	159715	159752	159715	159752
19	Dirty blocks evicted	0	89042	0	0	124035	124248	0	0	159710	159745
20	Program Execution Time	2.367491	1.724333	2.367691	2.369159	1.724509	1.725957	2.366954	2.367121	1.723845	1.723993
21	total hit rate	0.66666459	0.66666459	0.66657312	0.66582059	0.66657312	0.66582059	0.6669182	0.66684129	0.6669182	0.66684129
22	load_hit_rate	0.50192703	0.50192703	0.50186462	0.50079734	0.50186462	0.50079734	0.50215484	0.50209867	0.50215484	0.50209867
23	store hit rate	0.99535501	0.99535501	0.99520557	0.99508104	0.99520557	0.99508104	0.99566011	0.9955418	0.99566011	0.9955418
24	Number of NAND gates	82	82	60416	60416	60416	60416	360	360	360	360

Figure 8.4: Values of cache parameters for moderate cache size

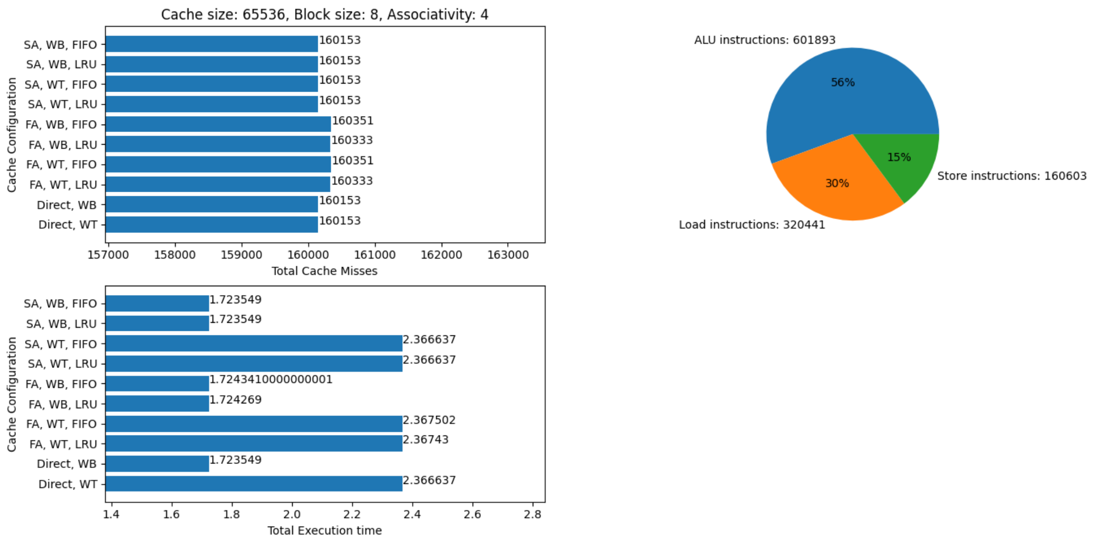
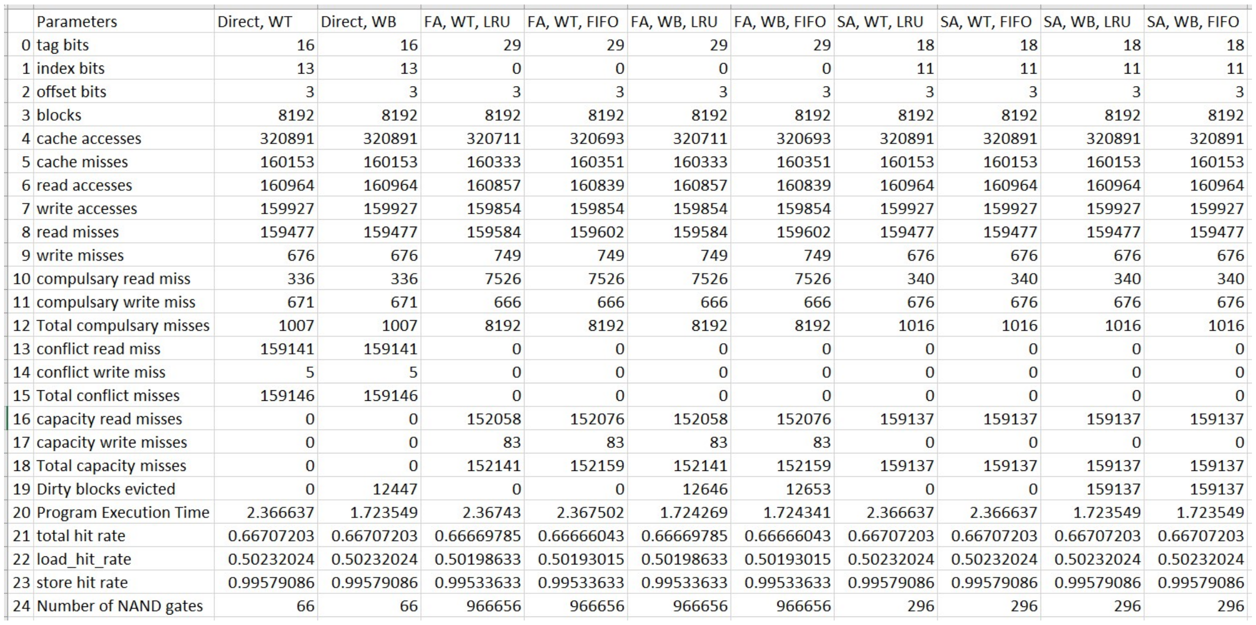


Figure 8.5: Observations for large cache size



	Parameters	Direct, WT	Direct, WB	FA, WT, LRU	FA, WT, FIFO	FA, WB, LRU	FA, WB, FIFO	SA, WT, LRU	SA, WT, FIFO	SA, WB, LRU	SA, WB, FIFO
0	tag bits	16	16	29	29	29	29	18	18	18	18
1	index bits	13	13	0	0	0	0	11	11	11	11
2	offset bits	3	3	3	3	3	3	3	3	3	3
3	blocks	8192	8192	8192	8192	8192	8192	8192	8192	8192	8192
4	cache accesses	320891	320891	320711	320693	320711	320693	320891	320891	320891	320891
5	cache misses	160153	160153	160333	160351	160333	160351	160153	160153	160153	160153
6	read accesses	160964	160964	160857	160839	160857	160839	160964	160964	160964	160964
7	write accesses	159927	159927	159854	159854	159854	159854	159927	159927	159927	159927
8	read misses	159477	159477	159584	159602	159584	159602	159477	159477	159477	159477
9	write misses	676	676	749	749	749	749	676	676	676	676
10	compulsary read miss	336	336	7526	7526	7526	7526	340	340	340	340
11	compulsary write miss	671	671	666	666	666	666	676	676	676	676
12	Total compulsary misses	1007	1007	8192	8192	8192	8192	1016	1016	1016	1016
13	conflict read miss	159141	159141	0	0	0	0	0	0	0	0
14	conflict write miss	5	5	0	0	0	0	0	0	0	0
15	Total conflict misses	159146	159146	0	0	0	0	0	0	0	0
16	capacity read misses	0	0	152058	152076	152058	152076	159137	159137	159137	159137
17	capacity write misses	0	0	83	83	83	83	0	0	0	0
18	Total capacity misses	0	0	152141	152159	152141	152159	159137	159137	159137	159137
19	Dirty blocks evicted	0	12447	0	0	12646	12653	0	0	159137	159137
20	Program Execution Time	2.366637	1.723549	2.36743	2.367502	1.724269	1.724341	2.366637	2.366637	1.723549	1.723549
21	total hit rate	0.66707203	0.66707203	0.66669785	0.66666043	0.66669785	0.66666043	0.66707203	0.66707203	0.66707203	0.66707203
22	load_hit_rate	0.50232024	0.50232024	0.50198633	0.50193015	0.50198633	0.50193015	0.50232024	0.50232024	0.50232024	0.50232024
23	store hit rate	0.99579086	0.99579086	0.99533633	0.99533633	0.99533633	0.99533633	0.99579086	0.99579086	0.99579086	0.99579086
24	Number of NAND gates	66	66	966656	966656	966656	966656	296	296	296	296

Figure 8.6: Values of cache parameters for large cache size

Future Scope of this Project

- We had thought to implement LFU (Least Frequently Used) replacement policy for all cache mappings. Due to time constraint, we couldn't implement it. If given a chance, we will surely implement it so that this policy can be compared with other policies.
- We had also thought to compute the hardware latency that comes at an additional cost while increasing the associativity of the cache. We will surely devise a way to do so if given a chance.

Work Distribution

- The entire cache study as well as brainstorming for the project was done by Aryan and Ruchit.
- The entire code was written by both Aryan and Ruchit by collaboritng on Google Collab.
- All the timely reporting of the project to Professor Sameer Kulkarni was done by Aryan and Ruchit.
- The numerical analysis as well the writing the report was done by Aryan and Ruchit.
- Rahul designed the graphical user interface for the program as well as made the presentation slides.

Bibliography

- A. Paramita and K. G. Smitha, "PARACACHE: Educational Simulator for Cache and Virtual Memory," 2017 International Symposium on Educational Technology (ISET), 2017, pp. 234-238, doi: 10.1109/ISET.2017.60.
- https://www.cs.utah.edu/ rajeev/cs3810/
- https://cseweb.ucsd.edu/classes/fa07/cse240a/project1.html (Trace files)