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.

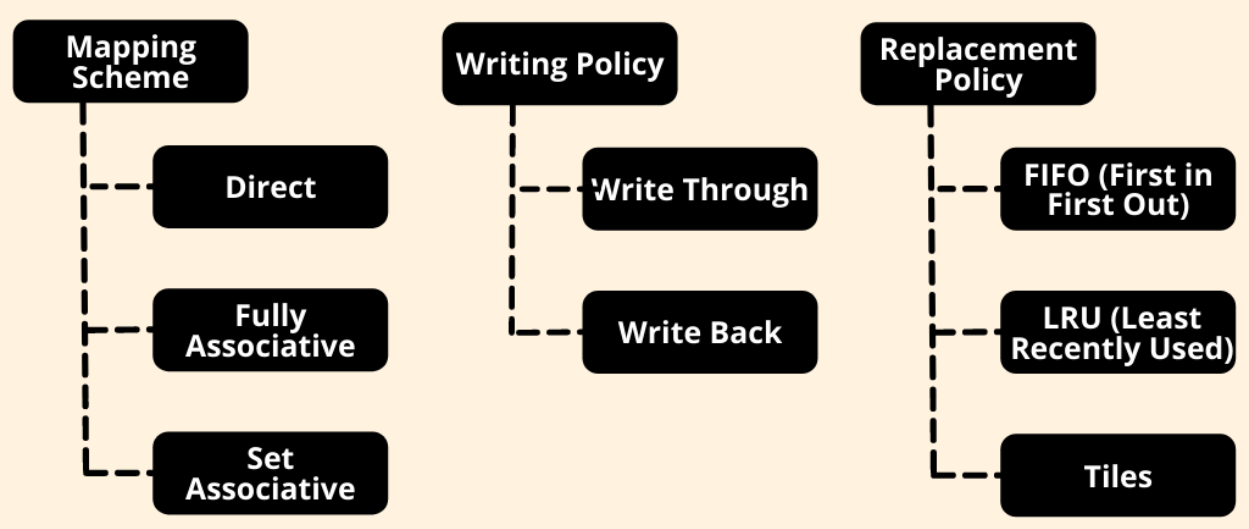


Figure 4.1: Cache Mapping and Cache Policies

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 and 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

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.

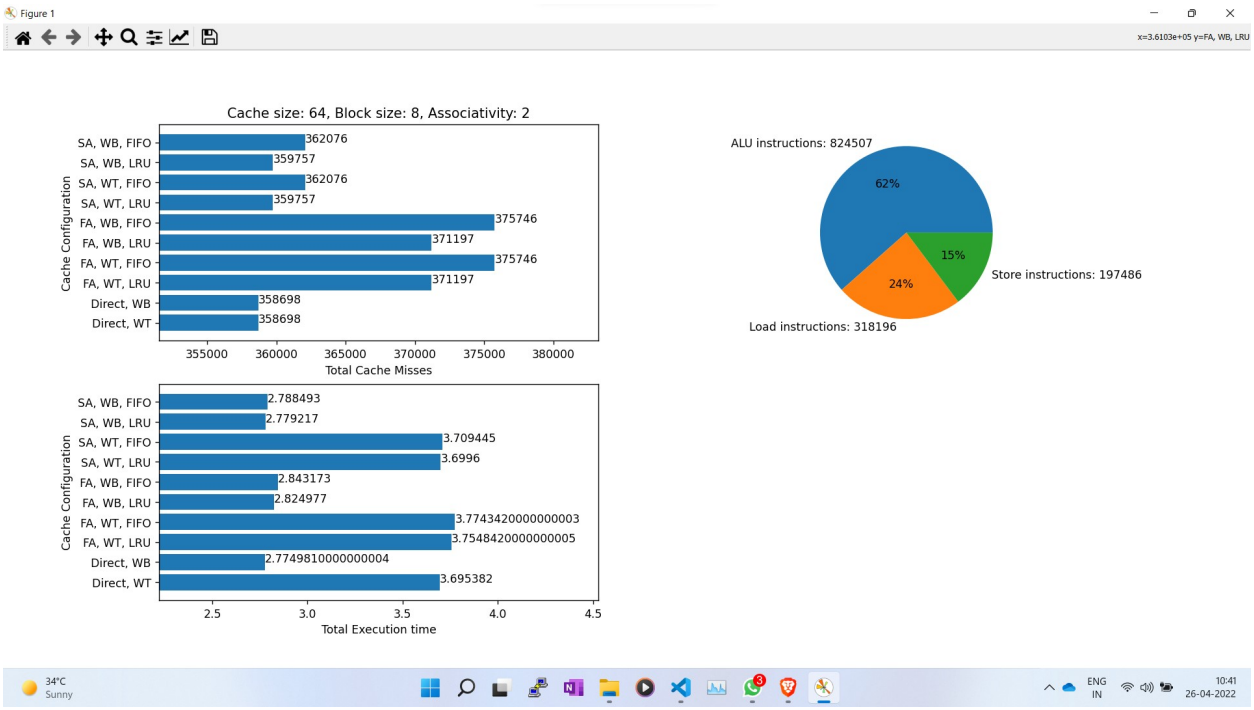
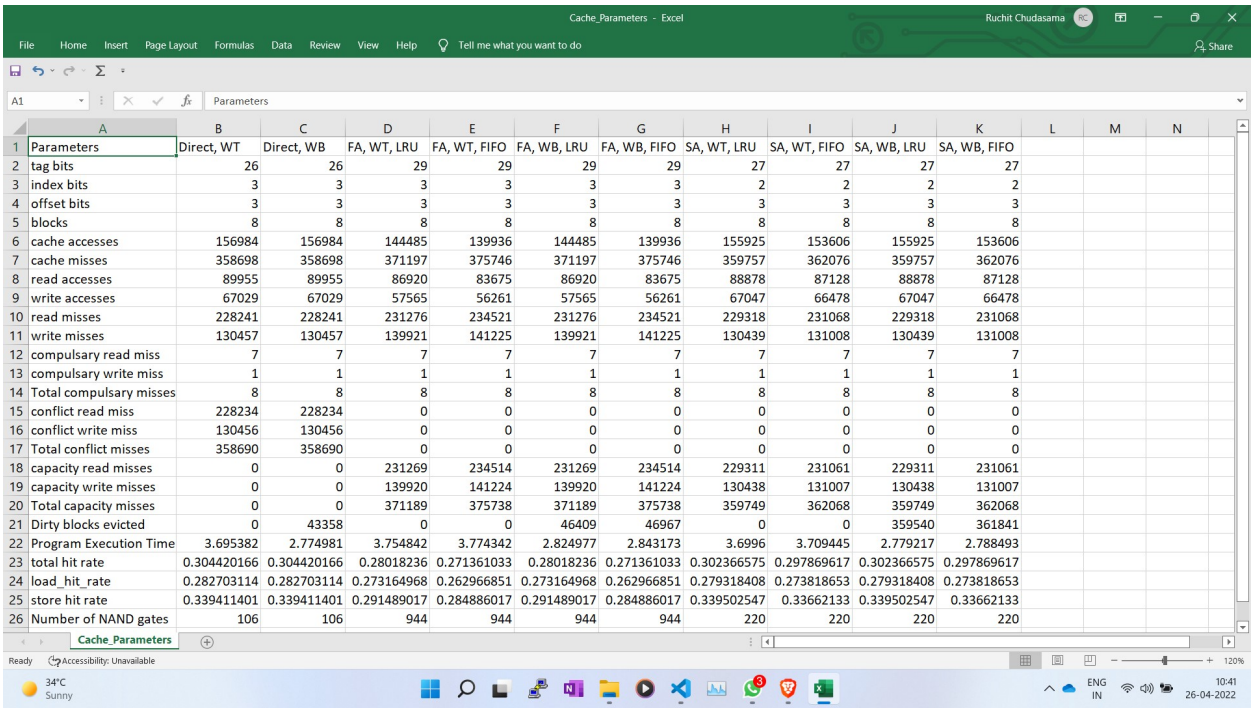


Figure 8.1: Observations for small cache size



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
tag bits	26	26	29	29	29	29	27	27	27	27
index bits	3	3	3	3	3	3	2	2	2	2
offset bits	3	3	3	3	3	3	3	3	3	3
blocks	8	8	8	8	8	8	8	8	8	8
cache accesses	156984	156984	144485	139936	144485	139936	155925	153606	155925	153606
cache misses	358698	358698	371197	375746	371197	375746	359757	362076	359757	362076
read accesses	89955	89955	86920	83675	86920	83675	88878	87128	88878	87128
write accesses	67029	67029	57565	56261	57565	56261	67047	66478	67047	66478
read misses	228241	228241	231276	234521	231276	234521	229318	231068	229318	231068
write misses	130457	130457	139921	141225	139921	141225	130439	131008	130439	131008
compulsary read miss	7	7	7	7	7	7	7	7	7	7
compulsary write miss	1	1	1	1	1	1	1	1	1	1
Total compulsary misses	8	8	8	8	8	8	8	8	8	8
conflict read miss	228234	228234	0	0	0	0	0	0	0	0
conflict write miss	130456	130456	0	0	0	0	0	0	0	0
Total conflict misses	358690	358690	0	0	0	0	0	0	0	0
capacity read misses	0	0	231269	234514	231269	234514	229311	231061	229311	231061
capacity write misses	0	0	139920	141224	139920	141224	130438	131007	130438	131007
Total capacity misses	0	0	371189	375738	371189	375738	359749	362068	359749	362068
Dirty blocks evicted	0	43358	0	0	46409	46967	0	0	359540	361841
Program Execution Time	3.695382	2.774981	3.754842	3.774342	2.824977	2.843173	3.6996	3.709445	2.779217	2.788493
total hit rate	0.304420166	0.304420166	0.28018236	0.271361033	0.28018236	0.271361033	0.302366575	0.297869617	0.302366575	0.297869617
load_hit_rate	0.282703114	0.282703114	0.273164968	0.262966851	0.273164968	0.262966851	0.279318408	0.273818653	0.279318408	0.273818653
store hit rate	0.339411401	0.339411401	0.291489017	0.284886017	0.291489017	0.284886017	0.339502547	0.33662133	0.339502547	0.33662133
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

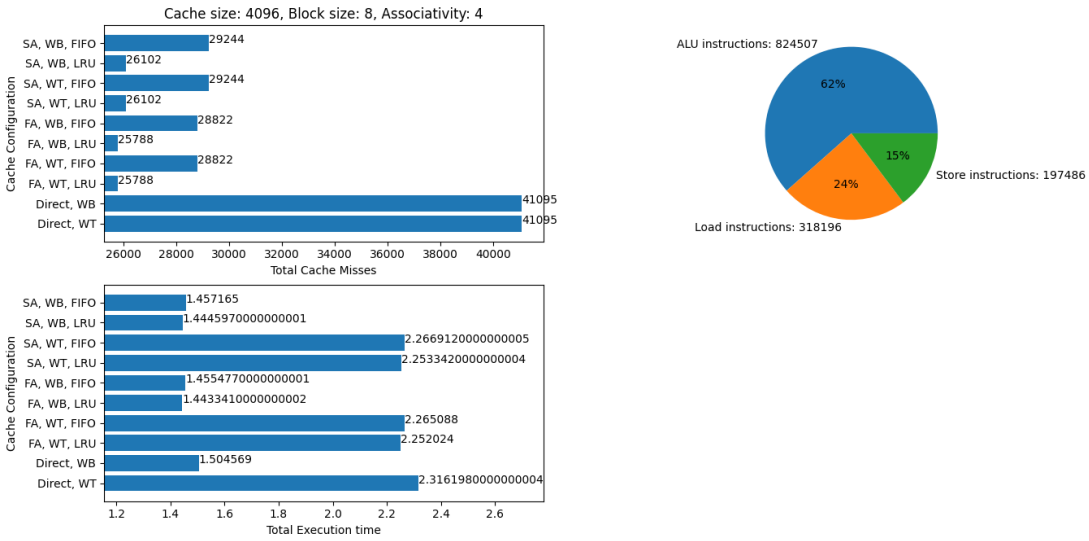
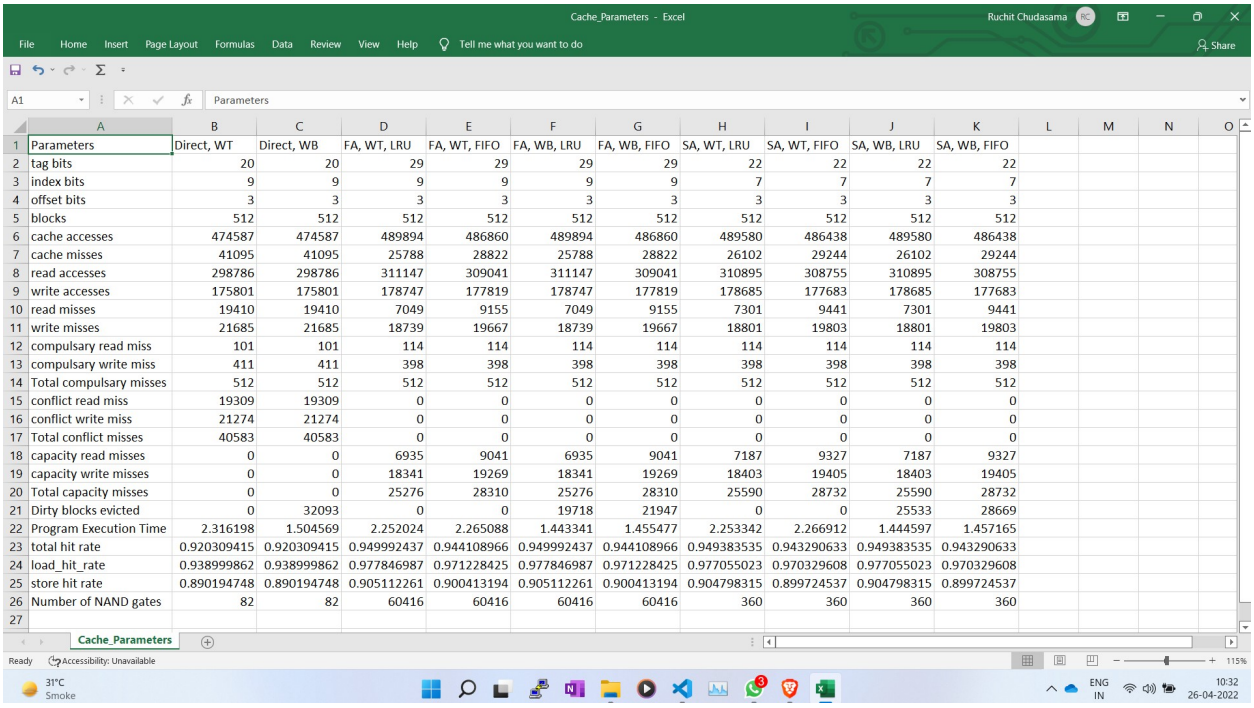


Figure 8.3: Observations for moderate cache size



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
tag bits	20	20	29	29	29	29	22	22	22	22
index bits	9	9	9	9	9	9	7	7	7	7
offset bits	3	3	3	3	3	3	3	3	3	3
blocks	512	512	512	512	512	512	512	512	512	512
cache accesses	474587	474587	489894	486860	489894	486860	489580	486438	489580	486438
cache misses	41095	41095	25788	28822	25788	28822	26102	29244	26102	29244
read accesses	298786	298786	311147	309041	311147	309041	310895	308755	310895	308755
write accesses	175801	175801	178747	177819	178747	177819	178685	177683	178685	177683
read misses	19410	19410	7049	9155	7049	9155	7301	9441	7301	9441
write misses	21685	21685	18739	19667	18739	19667	18801	19803	18801	19803
compulsary read miss	101	101	114	114	114	114	114	114	114	114
compulsary write miss	411	411	398	398	398	398	398	398	398	398
Total compulsary misses	512	512	512	512	512	512	512	512	512	512
conflict read miss	19309	19309	0	0	0	0	0	0	0	0
conflict write miss	21274	21274	0	0	0	0	0	0	0	0
Total conflict misses	40583	40583	0	0	0	0	0	0	0	0
capacity read misses	0	0	6935	9041	6935	9041	7187	9327	7187	9327
capacity write misses	0	0	18341	19269	18341	19269	18403	19405	18403	19405
Total capacity misses	0	0	25276	28310	25276	28310	25590	28732	25590	28732
Dirty blocks evicted	0	32093	0	0	19718	21947	0	0	25533	28669
Program Execution Time	2.316198	1.504569	2.252024	2.265088	1.443341	1.455477	2.253342	2.266912	1.444597	1.457165
total hit rate	0.920309415	0.920309415	0.949992437	0.944108966	0.949992437	0.944108966	0.949383535	0.943290633	0.949383535	0.943290633
load_hit_rate	0.938999862	0.938999862	0.977846987	0.971228425	0.977846987	0.971228425	0.977055023	0.970329608	0.977055023	0.970329608
store hit rate	0.890194748	0.890194748	0.905112261	0.900413194	0.905112261	0.900413194	0.904798315	0.899724537	0.904798315	0.899724537
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

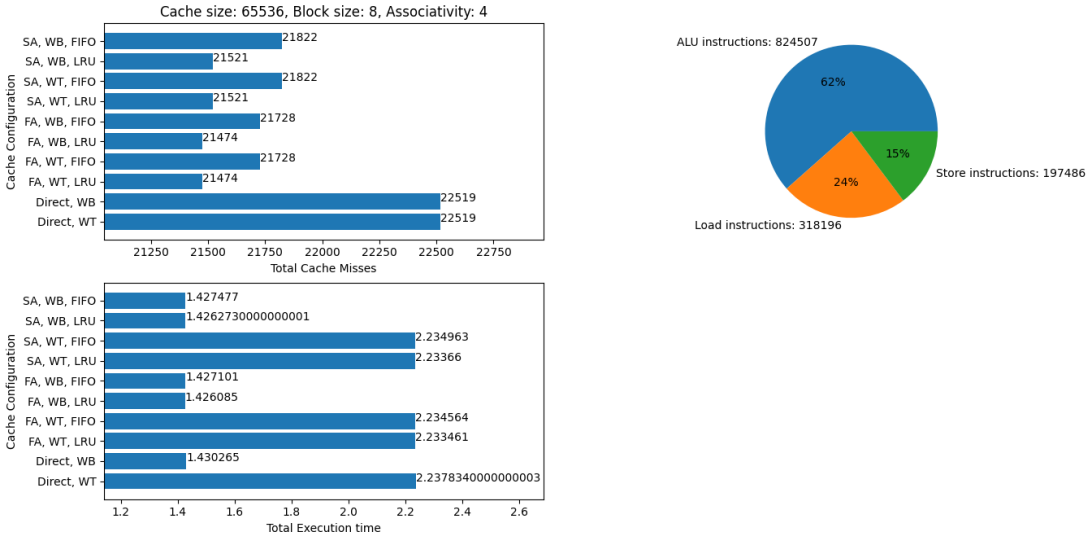
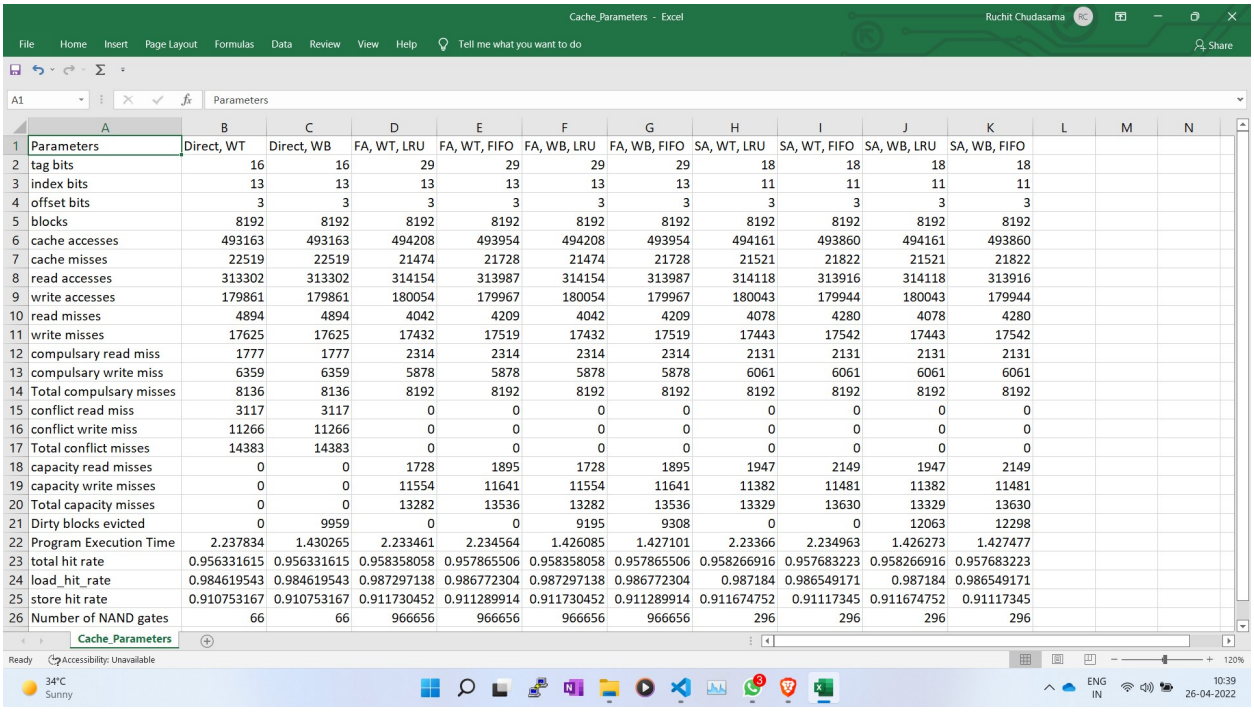


Figure 8.5: Observations for large cache size



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
tag bits	16	16	29	29	29	29	18	18	18	18
index bits	13	13	13	13	13	13	11	11	11	11
offset bits	3	3	3	3	3	3	3	3	3	3
blocks	8192	8192	8192	8192	8192	8192	8192	8192	8192	8192
cache accesses	493163	493163	494208	493954	494208	493954	494161	493860	494161	493860
cache misses	22519	22519	21474	21728	21474	21728	21521	21822	21521	21822
read accesses	313302	313302	314154	313987	314154	313987	314118	313916	314118	313916
write accesses	179861	179861	180054	179967	180054	179967	180043	179944	180043	179944
read misses	4894	4894	4042	4209	4042	4209	4078	4280	4078	4280
write misses	17625	17625	17432	17519	17432	17519	17443	17542	17443	17542
compulsary read miss	1777	1777	2314	2314	2314	2314	2131	2131	2131	2131
compulsary write miss	6359	6359	5878	5878	5878	5878	6061	6061	6061	6061
Total compulsary misses	8136	8136	8192	8192	8192	8192	8192	8192	8192	8192
conflict read miss	3117	3117	0	0	0	0	0	0	0	0
conflict write miss	11266	11266	0	0	0	0	0	0	0	0
Total conflict misses	14383	14383	0	0	0	0	0	0	0	0
capacity read misses	0	0	1728	1895	1728	1895	1947	2149	1947	2149
capacity write misses	0	0	11554	11641	11554	11641	11382	11481	11382	11481
Total capacity misses	0	0	13282	13536	13282	13536	13329	13630	13329	13630
Dirty blocks evicted	0	9959	0	0	9195	9308	0	0	12063	12298
Program Execution Time	2.237834	1.430265	2.233461	2.234564	1.426085	1.427101	2.23366	2.234963	1.426273	1.427477
total hit rate	0.956331615	0.956331615	0.958358058	0.957865506	0.958358058	0.957865506	0.958266916	0.957683223	0.958266916	0.957683223
load_hit_rate	0.984619543	0.984619543	0.987297138	0.986772304	0.987297138	0.986772304	0.987184	0.986549171	0.987184	0.986549171
store hit rate	0.910753167	0.910753167	0.911730452	0.911289914	0.911730452	0.911289914	0.911674752	0.91117345	0.911674752	0.91117345
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)