EECE7352 Computer Architecture
Homework 4

Part A

Relevant python files to generate the trace and trace files are attached in the folder 'PartA'.

1. Direct Mapped 2KB instruction cache, 64B block size.

   $Cache\ size = 2KB$
   $Block\ size = 64B$
   $Number\ of\ lines/blocks\ in\ cache = \frac{2 * 2^{10}}{2^6} = 2^5 = 32$

   A snippet of the generated trace file is shown below:
   2 0
   2 40
   2 80
   2 c0
   2 100
   2 140
   2 180
   2 1c0
   2 200
   2 240
   2 280
   2 2c0
   …
   2 700
   2 740
   2 780
   2 7c0

   The dineroIV command used here is:
   ```
   dineroIV -l1-isize 2K -l1-ibsize 64 -informat d < trace1.txt
   ```

Dinero Output:

```
-l1-isize 2048
-l1-ibsize 64
-l1-isbsize 64
-l1-iassoc 1
-l1-irepl l
-l1-ifetch d
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                    Total          Instrn          Data           Read          Write           Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                 32             32              0              0              0              0
  Fraction of total          1.0000         1.0000         0.0000         0.0000         0.0000         0.0000

 Demand Misses                  32             32              0              0              0              0
  Demand miss rate          1.0000         1.0000         0.0000         0.0000         0.0000         0.0000

 Multi-block refs               0
 Bytes From Memory           2048
 ( / Demand Fetches)        64.0000
 Bytes To Memory                0
 ( / Demand Writes)         0.0000
 Total Bytes r/w Mem         2048
 ( / Demand Fetches)        64.0000
```

2. 8-way set associative, with LRU replacement 2KB instruction cache with 64B block size. Generate address stream that touches every cache index, producing 3 misses and 4 hits per index, and only includes 3 unique addresses per cache index

$Cache\ size = 2KB$
$Block\ size = 64B$

$$Number\ of\ lines/blocks\ in\ cache = \frac{2 * 2^{10}}{2^6} = 2^5 = 32$$

$$Number\ of\ sets = \frac{32}{8} = 4\ sets$$

A snippet of the generated trace file is shown below:
2 0
2 40
2 80
2 0
2 40
2 80
2 0
…
2 600
2 640

2 680
2 600
2 640
2 680
2 600

The dineroIV command used here is:

```
dineroIV -l1-isize 2K -l1-ibsize 64 -l1-iassoc 8 -informat d < trace2.txt
```

Dinero Output:

```
-l1-isize 2048
-l1-ibsize 64
-l1-isbsize 64
-l1-iassoc 8
-l1-irepl l
-l1-ifetch d
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                    Total          Instrn          Data           Read          Write           Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                 28             28              0              0              0              0
  Fraction of total          1.0000         1.0000         0.0000         0.0000         0.0000         0.0000

 Demand Misses                  12             12              0              0              0              0
  Demand miss rate           0.4286         0.4286         0.0000         0.0000         0.0000         0.0000

 Multi-block refs                0
 Bytes From Memory             768
 ( / Demand Fetches)         27.4286
 Bytes To Memory                 0
 ( / Demand Writes)         0.0000
 Total Bytes r/w Mem           768
 ( / Demand Fetches)         27.4286
```

3. 2-way set associative, with LRU replacement 2KB instruction cache with 64B block size. Generate address stream that touches every cache index, producing 5 misses and 5 hits per index, and only includes 3 unique addresses per cache index, but produce an interleaving pattern of Miss-Hit-Miss-Hit-Miss-Hit.

$Cache\ size = 2KB$
$Block\ size = 64B$
$Number\ of\ lines/blocks\ in\ cache = \frac{2 * 2^{10}}{2^6} = 2^5 = 32$
$Number\ of\ sets = \frac{32}{2} = 16\ sets$

A snippet of the generated trace file is shown below:

```
2 0
2 0
2 40
2 40
2 80
2 80
2 80
2 80
2 c0
2 c0
…
2 700
2 700
2 700
2 700
2 740
2 740
2 780
2 780
2 780
2 780
2 7c0
2 7c0
2 800
2 800
```

The dineroIV command used here is:

```
dineroIV -l1-isize 2K -l1-ibsize 64 -l1-iassoc 2 -informat d < trace3.txt
```

Dinero Output:

```
-l1-isize 2048
-l1-ibsize 64
-l1-isbsize 64
-l1-iassoc 2
-l1-irepl l
-l1-ifetch d
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                    Total          Instrn          Data          Read          Write          Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                96             96              0             0              0              0
  Fraction of total          1.0000         1.0000         0.0000        0.0000         0.0000         0.0000

 Demand Misses                 33             33              0             0              0              0
  Demand miss rate           0.3438         0.3438         0.0000        0.0000         0.0000         0.0000

 Multi-block refs               0
 Bytes From Memory           2112
 ( / Demand Fetches)        22.0000
 Bytes To Memory                0
 ( / Demand Writes)          0.0000
 Total Bytes r/w Mem         2112
 ( / Demand Fetches)        22.0000
```

4. 2-way set associative 16KB data cache with 32B block size. Generate an address stream that will generate 5 hits and 5 misses

$Cache\ size = 16KB$
$Block\ size = 32B$

$$Number\ of\ lines/blocks\ in\ cache = \frac{2^4 * 2^{10}}{2^5} = 2^7 = 512$$

$$Number\ of\ sets = \frac{512}{2} = 256\ sets$$

A snippet of the generated trace file is shown below:
0 00000000
1 00000020
0 00000000
1 00000020
0 00000000
0 00000000
1 00000020
0 00000000
1 00000020
0 00000000

The dineroIV command used here is:

```
dineroIV -l1-dsize 16K -l1-dbsize 32 -l1-dassoc 2 -informat d < trace4.txt
```

Dinero Output:

```
-l1-dsize 16384
-l1-dbsize 32
-l1-dsbsize 32
-l1-dassoc 2
-l1-drepl l
-l1-dfetch d
-l1-dwalloc a
-l1-dwback a
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-dcache
 Metrics                    Total          Instrn          Data           Read          Write           Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                10              0             10              6              4              0
  Fraction of total          1.0000         0.0000         1.0000         0.6000         0.4000         0.0000

 Demand Misses                  2              0              2              1              1              0
  Demand miss rate          0.2000         0.0000         0.2000         0.1667         0.2500         0.0000

 Multi-block refs               0
 Bytes From Memory             64
 ( / Demand Fetches)        6.4000
 Bytes To Memory               32
 ( / Demand Writes)         8.0000
 Total Bytes r/w Mem           96
 ( / Demand Fetches)        9.6000
```

5. Generate instruction address reference streams that can detect the set associativity and the total size of an instruction cache. Given 32B block size, cache size not larger than 32KB.

   $Cache\ size < 32KB$
   $Block\ size = 32B$

   Set mapping can be used to create reference streams that can detect the set associativity and overall size of the instruction cache.

   The block offset field requires 5 bits, $(2^5 = 32)$. This means the lower 5 bits of the address will represent the offset within each cache block. For a maximum 32KB cache with 32B blocks, the total number of blocks would be $2^{10} = 1024\ blocks$. The index field hence requires 8 bits. The remaining bits are used for the tag field.

   To determine the total cache size, a sequence of addresses that map to different sets of the cache is generated. This means a sequence of addresses that have the same tag and index but different block offset values is generated as this causes each address to map to a different line in the same set. If the cache has fewer than 1024 blocks, addresses begin to miss due to eviction.

Sample sequence:

```
2 00000000    # set 0, tag 0
2 00000020    # set 1, tag 0
2 00000040    # set 2, tag 0
...
2 000007E0    # set 255, tag 0
2 00000800    # set 0, tag 1
2 00000820    # set 1, tag 1
2 00000840    # set 2, tag 1
...
2 00000FE0    # set 255, tag 1
```

DineroIV output:

```
dineroIV -l1-isize 32K -l1-ibsize 32 -informat d < trace5_1.txt
```

```
-l1-isize 32768
-l1-ibsize 32
-l1-isbsize 32
-l1-iassoc 1
-l1-irepl l
-l1-ifetch d
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                    Total          Instrn          Data           Read          Write           Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                512            512              0              0              0              0
  Fraction of total         1.0000         1.0000         0.0000         0.0000         0.0000         0.0000

 Demand Misses                 256            256              0              0              0              0
  Demand miss rate           0.5000         0.5000         0.0000         0.0000         0.0000         0.0000

 Multi-block refs                0
 Bytes From Memory            8192
 ( / Demand Fetches)         16.0000
 Bytes To Memory                 0
 ( / Demand Writes)          0.0000
 Total Bytes r/w Mem          8192
 ( / Demand Fetches)         16.0000

---Execution complete.
```

To detect the associativity of the cache, a sequence of addresses that map to the same set but have different tags is generated. If only one address per set can reside in the cache without eviction, the cache is direct-mapped. If all addresses fit in the set without eviction, it's fully associative. Otherwise, the observed number of addresses that fit in each set will correspond to the cache's associativity level.

DineroIV sample command:

```
dineroIV -l1-isize 32K -l1-ibsize 32 -l1-iassoc 2 -informat d < trace5_2.txt
```

DineroIV output for 2 way associativity:

```
-l1-isize 32768
-l1-ibsize 32
-l1-isbsize 32
-l1-iassoc 2
-l1-irepl l
-l1-ifetch d
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                    Total          Instrn          Data           Read          Write           Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                 16             16              0              0              0              0
  Fraction of total          1.0000         1.0000         0.0000         0.0000         0.0000         0.0000

 Demand Misses                  16             16              0              0              0              0
  Demand miss rate          1.0000         1.0000         0.0000         0.0000         0.0000         0.0000

 Multi-block refs                0
 Bytes From Memory             512
 ( / Demand Fetches)         32.0000
 Bytes To Memory                 0
 ( / Demand Writes)          0.0000
 Total Bytes r/w Mem           512
 ( / Demand Fetches)         32.0000

---Execution complete.
```

6. Repeat 5, generate an instruction stream that will determine replacement algorithm.

   To determine the replacement policy of a cache i.e., LRU, FIFO, random, a sequence of addresses is generated that fills up a specific set within the cache and then accesses additional addresses that map to the same set but differ in ways that will cause evictions. On analyzing which addresses are evicted first, the replacement policy can be inferred.

   Generated sequence:
   2 00000000
   2 00001000
   2 00002000
   2 00003000
   2 00004000
   2 00005000
   2 00000000
   2 00001000
   2 00002000
   2 00003000

   DineroIV command:

`dineroIV -l1-isize 4K -l1-ibsize 32 -l1-iassoc 4 -l1-irepl r -informat d < trace6.txt`

The switch -l1-irepl is varied and the output (number of demand misses) is observed.

```
-l1-isize 4096
-l1-ibsize 32
-l1-isbsize 32
-l1-iassoc 4
-l1-irepl r
-l1-ifetch d
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                    Total          Instrn          Data          Read          Write          Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                 10             10             0             0             0             0
  Fraction of total          1.0000         1.0000        0.0000        0.0000        0.0000        0.0000

 Demand Misses                   8              8             0             0             0             0
  Demand miss rate           0.8000         0.8000        0.0000        0.0000        0.0000        0.0000

 Multi-block refs                0
 Bytes From Memory             256
 ( / Demand Fetches)         25.6000
 Bytes To Memory                 0
 ( / Demand Writes)          0.0000
 Total Bytes r/w Mem           256
 ( / Demand Fetches)         25.6000

---Execution complete.
```

```
-l1-isize 4096
-l1-ibsize 32
-l1-isbsize 32
-l1-iassoc 4
-l1-irepl l
-l1-ifetch d
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                    Total          Instrn          Data          Read          Write          Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                 10             10             0             0             0             0
  Fraction of total          1.0000         1.0000        0.0000        0.0000        0.0000        0.0000

 Demand Misses                  10             10             0             0             0             0
  Demand miss rate           1.0000         1.0000        0.0000        0.0000        0.0000        0.0000

 Multi-block refs                0
 Bytes From Memory             320
 ( / Demand Fetches)         32.0000
 Bytes To Memory                 0
 ( / Demand Writes)          0.0000
 Total Bytes r/w Mem           320
 ( / Demand Fetches)         32.0000

---Execution complete.
```

Part B

Model instruction cache and data cache with the provided trace. Total cache space is 16KB. The block size should be varied (8B, 16B and 32B) and the associativity should be varied (direct-mapped, 2-way and 4-way). Model both split (separate instruction and data caches) and shared (all accesses go to a single cache that holds both instructions and data) caches. There is a total of 18 simulations. No other parameters should be varied. Graph the results you get from these experiments and discuss in detail why you see different trends in the graphs.

There are 18 simulations shown in Table 1 that are to be modeled. The dineroIV outputs are generated initially using the attached bash script and all outputs are stored as text files, which are then processed using a python script. The sample dinero IV command is shown below.

Split cache:

```
dineroIV -l1-isize 8K -l1-ibsize ${block_size} -l1-iassoc ${assoc} -l1-dsize 8K -
l1-dbsize ${block_size} -l1-dassoc ${assoc} -informat d < trace
```

Shared cache:

```
dineroIV -l1-usize 16K -l1-ubsize ${block_size} -l1-uassoc ${assoc} -informat d <
trace
```

The eighteen possible simulations are shown in Table 1.

S. No	Block Size	Associativity	Split/Shared Cache
1	8	Direct-mapped	Shared
2	8	Direct-mapped	Split
3	8	2-way set associative	Shared
4	8	2-way set associative	Split
5	8	4-way set associative	Shared
6	8	4-way set associative	Split
7	16	Direct-mapped	Shared
8	16	Direct-mapped	Split
9	16	2-way set associative	Shared
10	16	2-way set associative	Split
11	16	4-way set associative	Shared
12	16	4-way set associative	Split
13	32	Direct-mapped	Shared
14	32	Direct-mapped	Split
15	32	2-way set associative	Shared
16	32	2-way set associative	Split
17	32	4-way set associative	Shared
18	32	4-way set associative	Split

Table 1: Simulations to Model Instruction and Data Caches

The results of the dineroIV simulation in the form of various plots is shown in the next section.

Figures 1 and 2 illustrate the relationship between cache block size and demand misses for instruction and data caches. As block size increases, total cache misses generally decrease, but at a diminishing rate. Larger blocks reduce the need for memory fetches but can increase miss penalty and hinder associativity. Across both caches, higher associativity, i.e. 4-way consistently outperforms lower associativity in reducing misses.

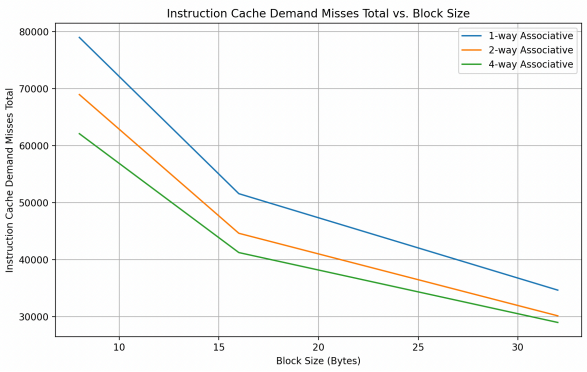

Figure 1: ICache Demand Misses Total

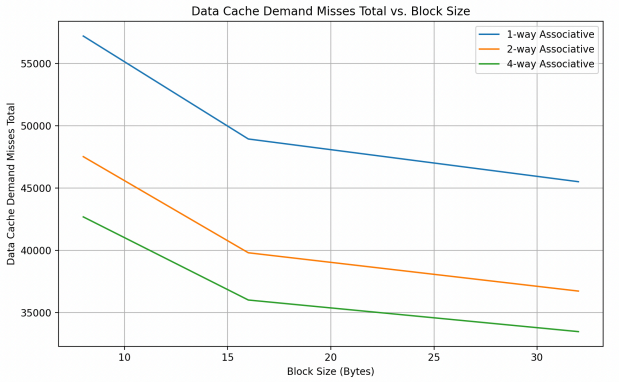

Figure 2: DCache Demand Misses Total

Figures 3 and 4 illustrate the relationship between cache block size and the amount of data transferred from main memory to the cache, for each cache. The graphs show that increasing cache block size reduces the amount of data transferred from memory to both the instruction and data caches. However, this reduction diminishes with larger block sizes as larger block sizes can also increase the miss rate, leading to more frequent fetches from memory, even though each fetch brings more data. Higher associativity consistently outperforms lower associativity in minimizing data transfers.

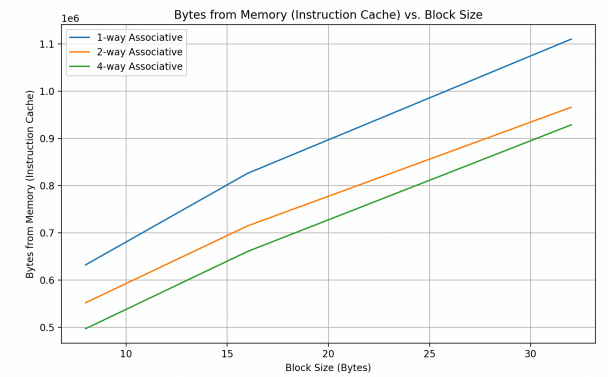

Figure 3: ICache Bytes from Memory

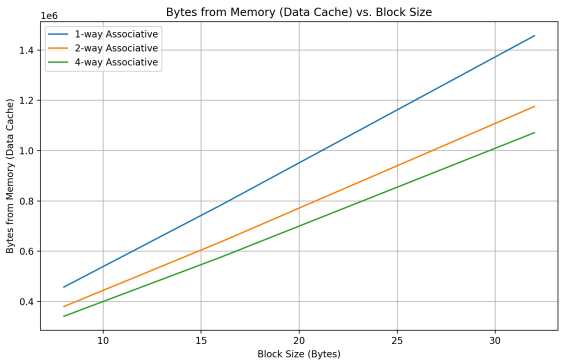

Figure 4: DCache Bytes from Memory

Figures 5 and 6 illustrate the total number of bytes read and written to the each cache as the block size increases. Both graphs show that as the block size increases, the total number of bytes read and written to the cache also increases. Higher associativity consistently outperforms lower associativity in minimizing data transfers as higher associativity allows the cache to store more data without evicting useful data, which reduces the need to read and write data from and to main memory.

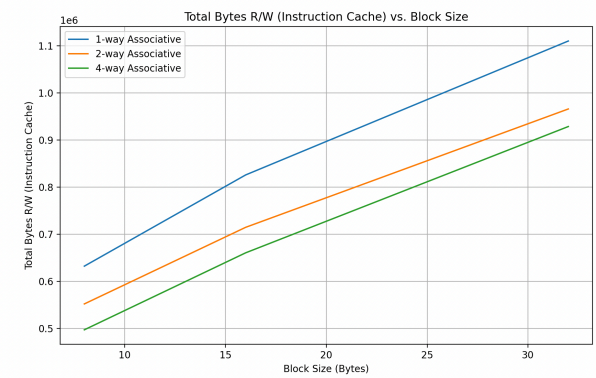

Figure 5: ICache Total Bytes R/W

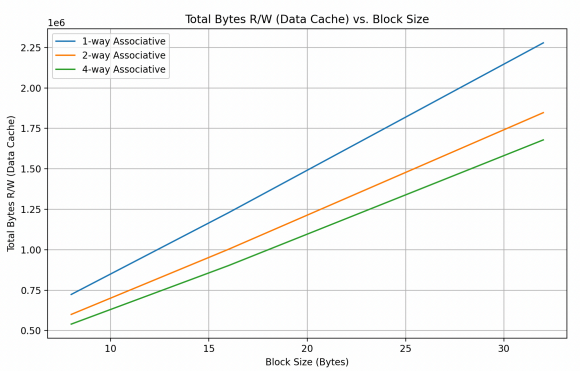

Figure 6: DCache: Total Bytes R/W

Figures 7, 8 and 9 are plotted using the output from the shared cache. Figure 7 shows the number of demand misses for different cache configurations. It is observed that as the cache size or associativity is increased, the number of demand misses generally decreases. Figure 8 shows the demand miss rates for various cache configurations. The miss rate is the percentage of memory accesses that result in a cache miss. Here, it is seen that the miss rate decreases as the cache size or associativity increases, which is consistent with the observation from Figure 7. Figure 9 displays the total number of memory bytes accessed for each cache configuration. As the cache size or associativity increases, the total memory bytes accessed tend to decrease.

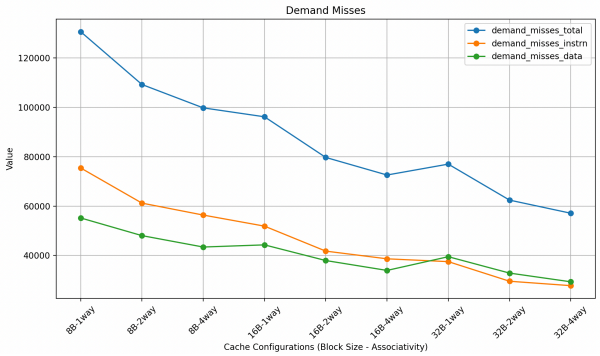

Figure 7: Demand Misses Plot

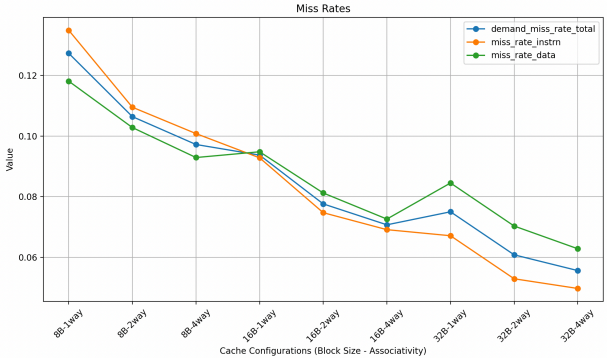

Figure 8: Demand Miss Rates Plot

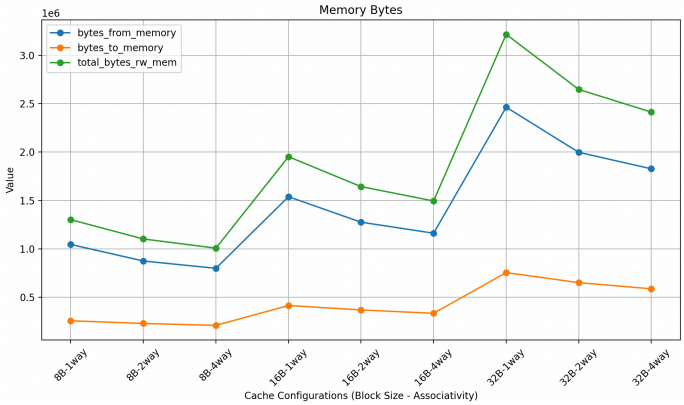

Figure 9: Memory Bytes Plot

Part C

Use the prefetching capabilities provided in dineroIV (the -Tfetch and -Tpfdist switches). Study the effects of different data cache prefetching policies and report your results. Attempt to find the best prefetching policy and discuss why you feel this would be the best policy for the given workload.

The -Tfetch switch explores several prefetching modes. Demand-only (d) fetches data only on cache misses, serving as the baseline. Always prefetch (a) continuously prefetches data, which can be useful for predictable access patterns. Prefetch on miss (m) triggers prefetching only when a miss occurs, reducing unnecessary prefetching. Tagged prefetch (t) uses tags to predict future data access, suitable for complex access patterns. Load forward (l) prefetches data based on upcoming load operations, benefiting workloads with load dependencies. Finally, subblock prefetch (s) fetches smaller portions of data, optimizing fine-grained access patterns by reducing unnecessary block fetches. These modes are explored at various distances using the -Tpfdist switch to determine the best prefetching approach. Table 2 shows the impact of different prefetching modes and distances on cache performance metrics such as demand miss rate, prefetch misses, and prefetch miss rate, helping identify the most effective policy for a 2KB, 64-block data cache.

The following dineroIV command was used to generate the outputs:

```
dineroIV -l1-dsize 2K -l1-dbsize 64 -l1-dfetch ${fetch_switch} -l1-dpfdist ${dist_switch} -informat d < trace
```

-Tfetch switch	-Tpfdist switch	Prefetch Fetches	Demand Misses	Demand Miss Rate	Prefetch Misses	Prefetch Misses Rate
a	1	288238	85837	0.1836	72113	0.2502
a	2	288238	95245	0.2038	80569	0.2795
a	4	288238	97548	0.2087	81672	0.2833
m	1	62278	83045	0.1777	50315	0.8079
m	2	68280	88856	0.1901	58603	0.8583
m	4	70144	91842	0.1965	61568	0.8777
t	1	69329	82711	0.1769	55464	0.8
t	2	71326	88933	0.1903	60743	0.8516
t	4	71839	91890	0.1966	63030	0.8774
l	1	0	80885	0.173	0	0
s	1	288238	80885	0.173	0	0

Table 2: Possible Prefetching Switch Configurations

The always prefetch switch initiates prefetching on every access, leading to unnecessary data fetching and increased prefetch misses, especially at greater distances. For example, at a distance

of 4, it raises the demand miss rate to 0.2087 and the prefetch miss rate to 0.2833, indicating diminishing returns for this workload. The prefetch on miss, m switch prefetches only after a miss, minimizing unnecessary prefetches but resulting in a relatively high demand miss rate. At a distance of 4, it reaches a demand miss rate of 0.1965 and a prefetch miss rate of 0.8777, showing limited effectiveness. The tagged prefetch, t switch uses tags to anticipate accesses, achieving slightly better results than prefetch on miss but showing similar demand and prefetch miss rates across distances. At a distance of 4, it has a demand miss rate of 0.1966 and a prefetch miss rate of 0.8774. The load forward, l switch prefetches based on upcoming load operations, achieving the lowest demand miss rate (0.173) without any prefetch misses, suggesting it efficiently captures load dependencies and avoids unnecessary prefetches. The subblock prefetch, s switch fetches smaller data portions, achieving the same low demand miss rate (0.173) and zero prefetch misses as load-forward mode, making it suitable for workloads with fine-grained data needs.

To summarize, the l and s modes perform best, both achieving a demand miss rate of 0.173 with zero prefetch misses. However, the load forward is optimal as it prefetches only relevant data based on load operations, minimizing cache pollution and unnecessary traffic. This approach aligns well with the workload's predictable access patterns, making it the recommended policy for this workload.

Part D

Read that the attached paper on using deep learning to guide cache replacement. Describe how an LSTM model can be used to guide replacement. Why is this type of replacement scheme more effective than LRU? Explain the fundamental differences between heuristic-driven versus learning-based schemes. Even though this paper was published in 2019, there has been more recent work that cites the ideas from this paper. Describe the work that has cited this paper and discuss how it differs from Glider.

The 2019 paper “Applying Deep Learning to the Cache Replacement Problem” introduces an innovative approach using an LSTM model to guide cache replacement. By framing cache management as a sequence labeling task, each cache access is classified as either "cache-friendly" or "cache-averse" based on previous access patterns. An LSTM with an attention mechanism analyzes sequences of memory accesses, learning dependencies within the program's control flow history to inform cache decisions. This learning-based approach enables Glider—a cache replacement policy derived from the LSTM model—to achieve superior cache efficiency compared to heuristic approaches like Least Recently Used (LRU).

In contrast, LRU employs a straightforward heuristic, replacing the cache line that has been unused for the longest period. This can be ineffective in workloads with complex access patterns, as LRU does not adapt to the varying reuse intervals commonly seen in real-world applications. Glider, on the other hand, uses an offline-trained LSTM to learn intricate access patterns over longer histories, resulting in significantly lower miss rates. By aligning cache replacement decisions more closely with optimal strategies, Glider outperforms traditional heuristics.

The main difference between heuristic-based and learning-based cache replacement lies in adaptability. While heuristic methods like LRU rely on fixed rules, which may struggle with diverse workloads, learning-based methods dynamically adjust to workload-specific access patterns, enabling more flexible and accurate predictions.

Since the publication of “Applying Deep Learning to the Cache Replacement Problem,” research has increasingly focused on using machine learning in cache management and other resource allocation tasks. Antoniadis et al.'s 2023 paper, “Online Metric Algorithms with Untrusted Predictions,” examines how algorithms can leverage machine learning predictions for online decisions, even when predictions are unreliable. Their work introduces a framework that balances prediction use with fallback strategies, adapting to both prediction accuracy and real-world uncertainty. This approach reflects a broader trend of integrating traditional algorithms with prediction-driven methods, enhancing strategies like Glider to improve performance in complex systems like multicore architectures.

Part E

In this part you will use the allcache.so Pin tool to study the relationship between a program, compiler optimizations, and memory behavior. You will see a large number of memory address values for both the instruction stream and the data stream. Now recompile your program applying different optimization switches, rerun the new binary with Pin, and compare the results. Try out 3 different compiler switch settings that generate a significant change in the memory stream. Plot these results and attempt to explain what trends you are seeing.

The program studied in this part is the C code for gaussian elimination. The code is present in the folder attached to this document. The pin output was obtained using the attached bash script, the outputs were converted to the pandas dataframe format, and the following plots were obtained using a python script.

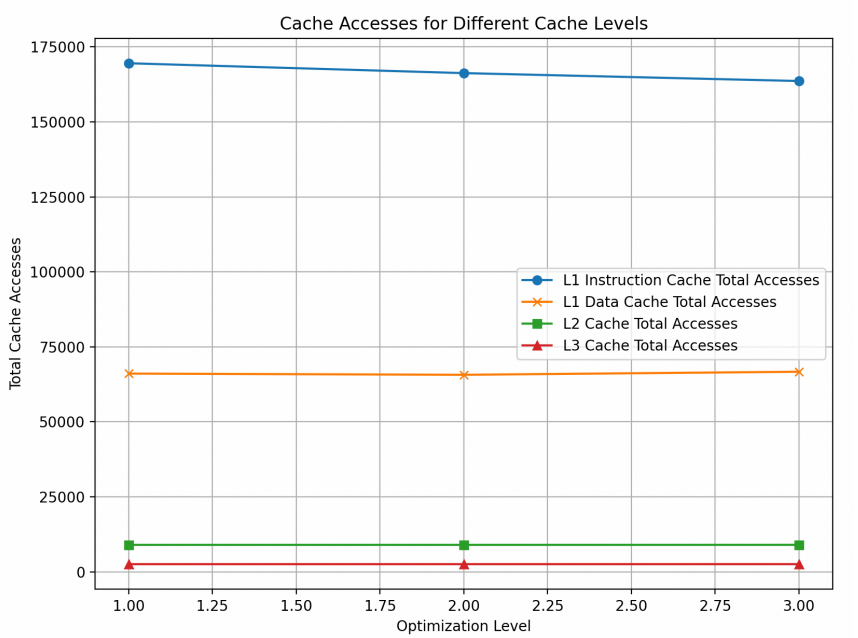

Figure 10: Cache Access Plot

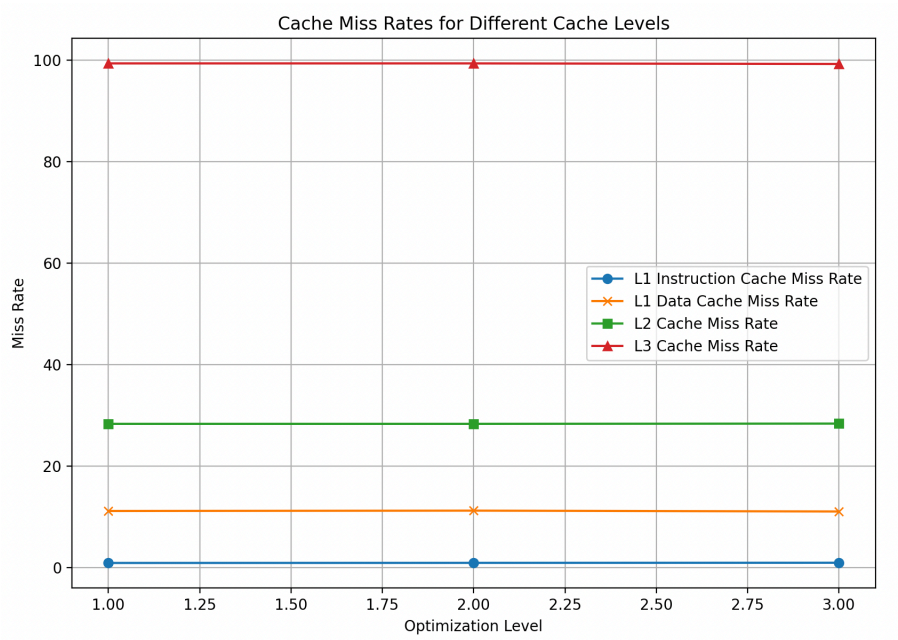

Figure 11: Cache Miss Rates Plot

Figure 10 shows a steady increase in cache accesses across all cache levels as the optimization level increases. This is likely due to more aggressive code optimizations, which can lead to increased instruction fetching and data access. Figure 11 demonstrates a decrease in cache miss rates for all cache levels as the optimization level rises. This suggests that the compiler is able to generate more efficient code, resulting in better cache locality and fewer cache misses. Overall, these figures indicate that higher optimization levels can lead to improved cache performance, potentially resulting in faster execution times.