Lab 2: Cache Simulator Report

Noah Schoonover
noah.schoonover@student.nmt.edu
CSE 331: Computer Architecture

March 21, 2022 (Spring 2022)

1 Introduction

This lab report pertains to the CSE 331 Lab 2: Cache Simulator. The cache simulator was implemented in `C`. The cache configuration is read from a file, which specifies the line size, associativity, total cache size, replacement policy, miss penalty, and write miss policy. After configuring the cache, a trace file is read line by line for data memory accesses. For each trace, the program simulates the cache performance by maintaining a cache directory. Finally, the simulator logs the total hit percentage, load and store hit percentages, total run time in cycles, and average memory access latency to a file.

The usage is specified below:

```
./lab2 [config file] [trace file]
```

The output log file will be created at `[trace file].out`.

The *Methods* section will describe how the simulator was implemented, as well as testing methods for ensuring accuracy. The *Results* section will analyze the cache performance of each trace file against each configuration file. The *References* section cites sources which were used for gathering understanding on minor complexities for the simulator.

2 Methods

This section will discuss the implementation and testing of the cache simulator. Partial code samples and other snippets are provided for detailed explanation. The terms "line" and "block" will be used interchangeably.

2.a Implementation

First, the cache configuration is stored in a `struct` named `cfg`, and the cache is allocated with the parameters described below. To implement the cache simulator, it is necessary to calculate some parameters which are derived from the configuration file. For example, the number of lines/blocks in the cache can be calculated using the total cache size and the line/block size. First, the total cache size must be converted from `KB` to `Bytes`.

```
int data_size_bytes = cfg.data_size * (1 << 10);
num_lines = data_size_bytes / cfg.line_size;
```

A `struct` is defined for holding information about each line in the cache.

```
typedef struct {
    unsigned int tag;
    bool valid;
    bool dirty;
    unsigned int fifo;
} Line;
```

The `fifo` member is a counter used for tracking which cache block entered the set first and should be evicted according to the FIFO replacement algorithm. Now the cache can be allocated and initialized:

```
cache = malloc(num_lines * sizeof(Line));
// error checking omitted for brevity
memset(cache, 0, num_lines * sizeof(Line));
```

We also have to calculate the number of sets in the cache using the associativity parameter. In the case of a fully associative cache, we consider every block to exist within a single set.

```
if (cfg.associativity) {
    // direct mapping or n-way associativity
    lines_per_set = cfg.associativity;
    sets = num_lines / lines_per_set;
}
else {
    // full associativity
    lines_per_set = num_lines;
    sets = 1;
}
```

With these parameters calculated, it is possible to generate the bit offsets and bit masks required for breaking down each address in the trace file. Each 32-bit address can be broken down in the following manner:

- For direct mapping or set-associative caches:

```
|---- tag ----|---- set ----|---- offset ----|
```

- For fully associative caches:

```
|------------tag------------|---- offset ----|
```

The bit masks and offsets are calculated as follows:

```
set_mask = sets - 1;
    /* because sets is a power of 2, sets-1 will yield a mask
       for example, 8 == 0b1000 so 8 - 1 == 0b0111 */
set_bits = __builtin_popcount(set_mask);
    /* __builtin_popcount is a builtin gcc function for obtaining
       the Hamming Weight (number of set bits) */
block_bits = __builtin_popcount(cfg.line_size - 1);
    /* this is a quick way to get the floor of log2(line_size) */
set_offset = block_bits;
tag_offset = block_bits + set_bits;
```

In the case of a fully associative cache, `set_mask` and `set_bits` will equal zero. Thus, the `tag_offset` will be consistent with the address breakdown diagram above. Helper methods are then used for retrieving the set index and tag.

```
uint32_t get_set(uint32_t address) {
    uint32_t shifted_address = address >> set_offset;
    return shifted_address & set_mask;
}

uint32_t get_tag(uint32_t address) {
    return address >> tag_offset;
}
```

The `get_set` method will return 0 for fully associative caches because the `set_mask` is 0, which is important because all lines will belong to one set. When we calculate the first line in the set,

```
uint32_t start_line = set * lines_per_set;
```

the `start_line` index will always be zero.

When querying the cache, we loop through each line in the set using the loop shown below. In the case of a fully associative cache, the loop will visit every line. In the case of a direct mapping or n-way set associative cache, the loop will visit 1 or n lines, respectively.

```
for (int i = start_line; i < start_line + lines_per_set; i++) {
    if (cache[i].tag == tag && cache[i].valid) {
        // cache hit
    }
}
```

On a cache hit, the simulator will call `log_load_hit()` or `log_store_hit()` which simply increment hit counters.

On a cache miss, the address is inserted into the cache. First, the replacement algorithm will always search for an invalid block to overwrite before evicting a valid block. If all blocks in the set are valid, the replacement algorithm will use the `fifo` counter or a random integer to determine which block to evict, depending upon the cache configuration.

Note: if no-write allocate is specified in the configuration, the address will not be inserted into the cache for store operations.

2.b Testing

In order to test the cache simulator, I relied largely on `GDB` and curated trace files.

The source files contain many pre-processor directives for printing verbose debugging data to the console window. For example, setting `#define CACHE_VERBOSE 1` will enable a lot of print statements in the cache. I set breakpoints throughout the simulator and verified the behavior of each cache query or insert.

In order to test the replacement algorithms, I curated a trace file for testing which contained addresses mapping to the same set. This was achieved by modifying only the first portion of the address (the tag), while keeping the ending portions (the set and offset) identical.

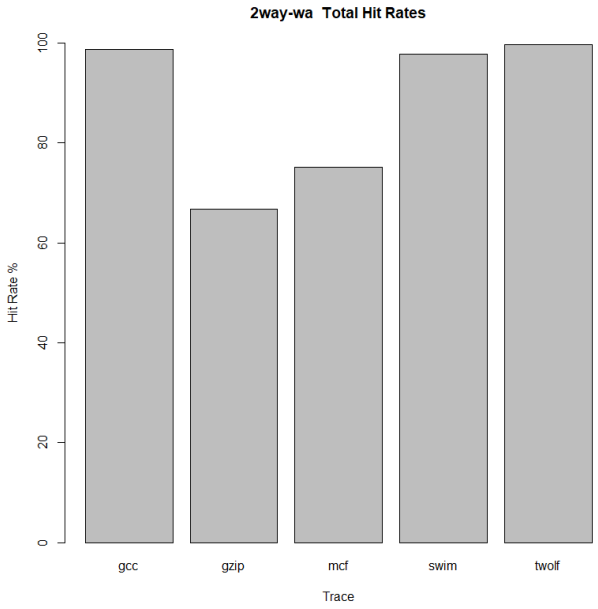
```
s 0xABCD001F 1
l 0xABCE001F 1
l 0xABCF001F 6
l 0xABD0001F 2
s 0xABD1001F 2
s 0xABD2001F 3
l 0xABD3001F 2
l 0xABD4001F 2
l 0xABD5001F 2
s 0xABD6001F 3
l 0xABD7001F 4
l 0xABD8001F 7
l 0xABD9001F 3
```

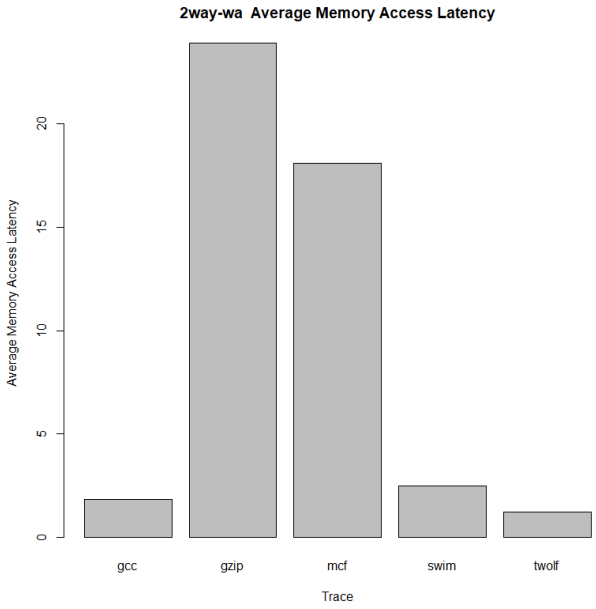
The simulator can then be executed within `GDB` using breakpoints to efficiently monitor the replacement algorithm within a single set.

3 Results

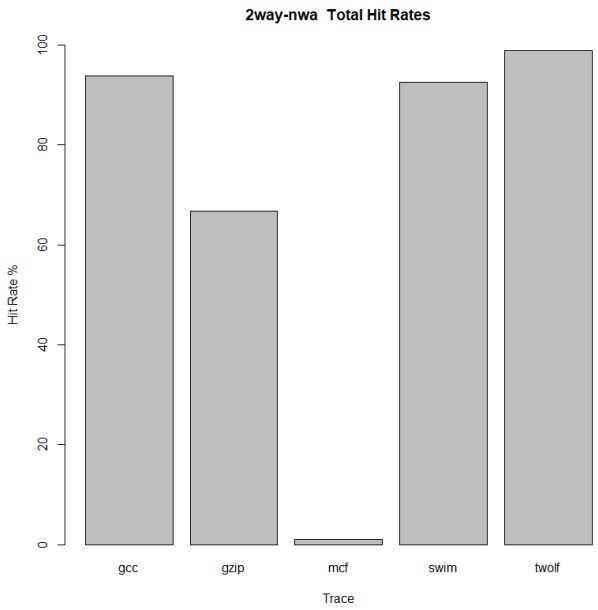
The following graphs show the results of each trace file against each configuration. For each configuration, the total hit rate and average memory access latency is shown. See the *Appendix* for a complete list of each output from the simulator.

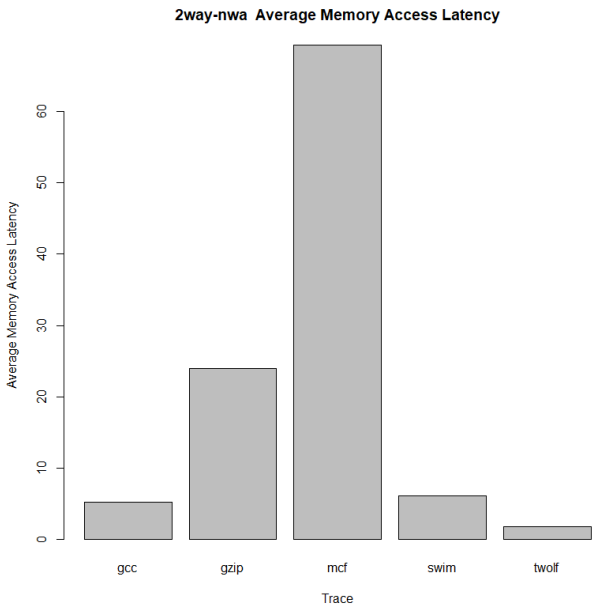
- `2way-wa.conf`



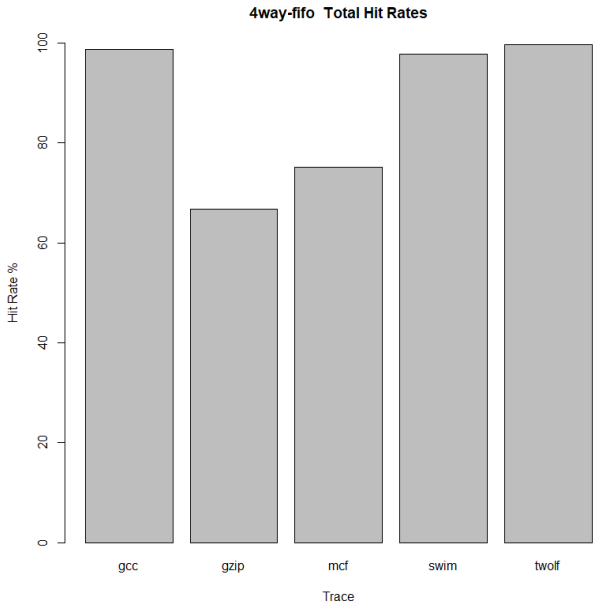


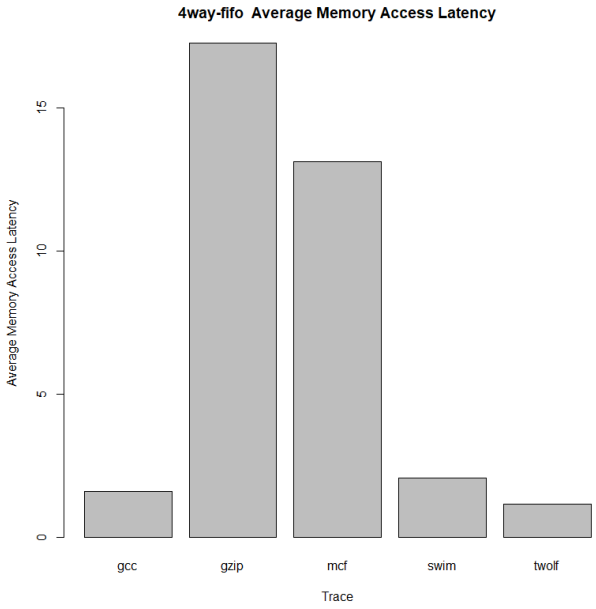
- `2way-nwa.conf`



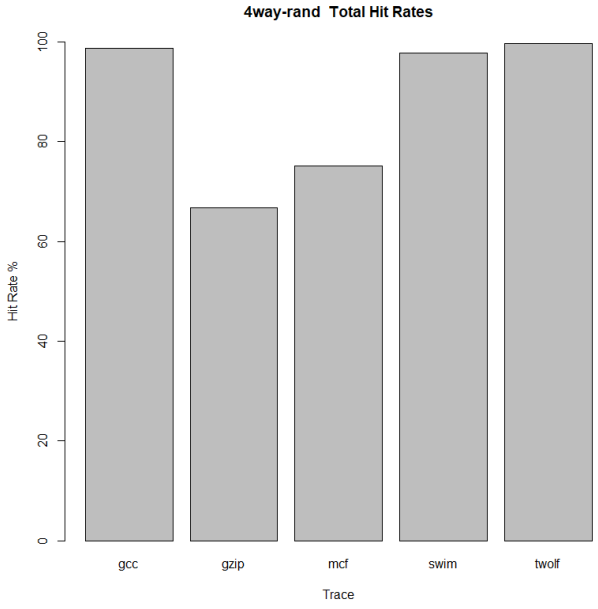


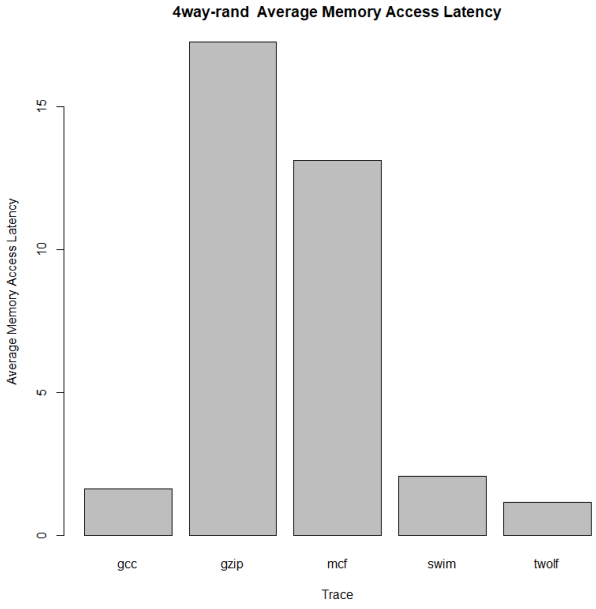
- `4way-fifo.conf`



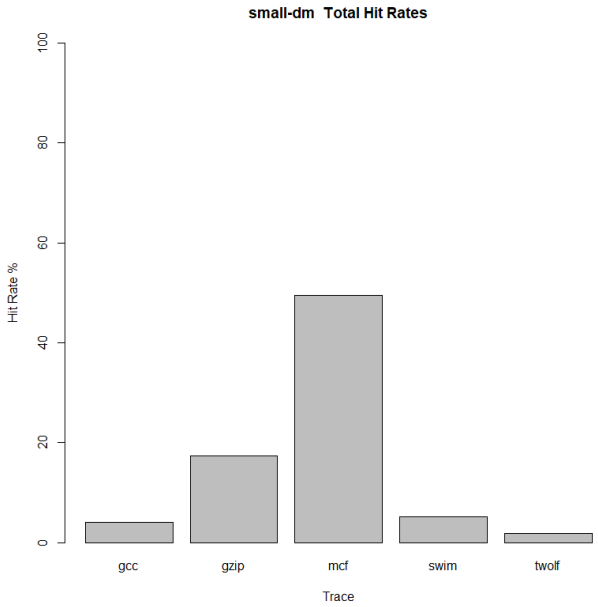


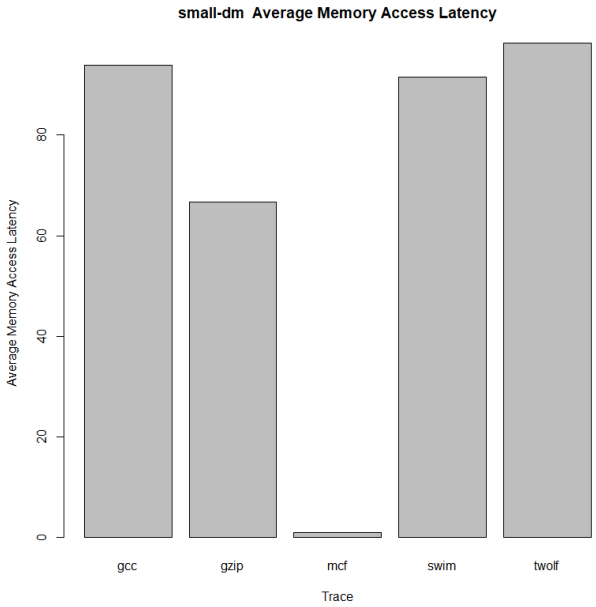
- `4way-rand.conf`



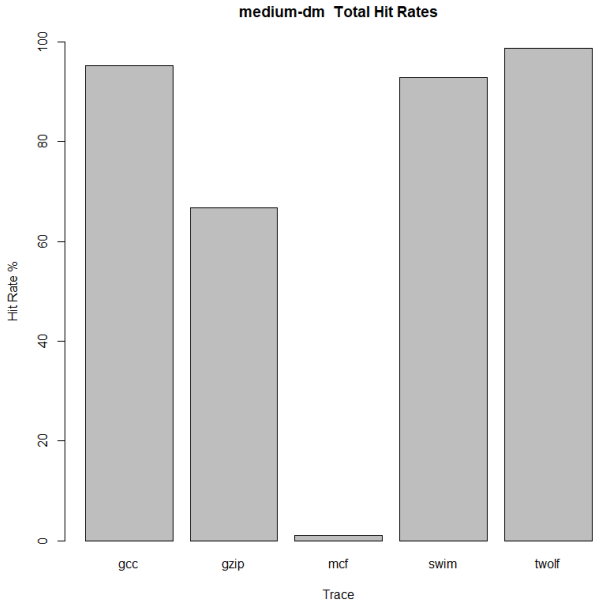


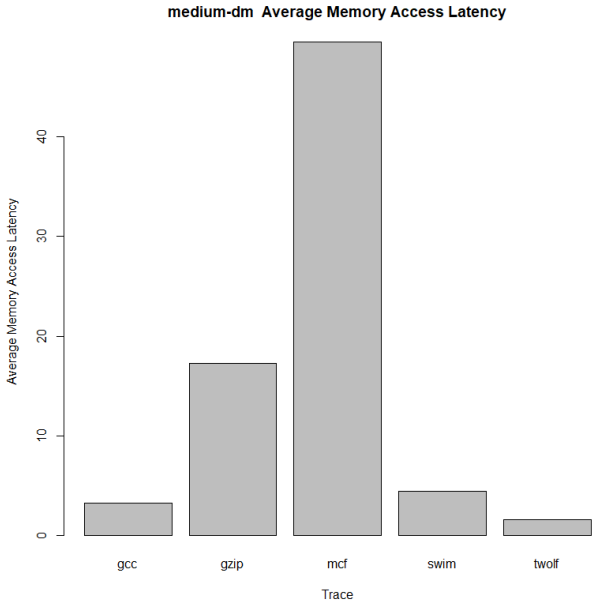
- `small-dm.conf`



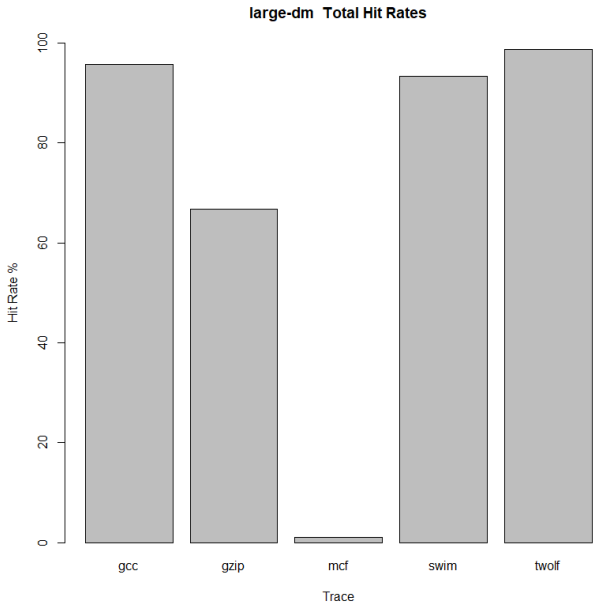


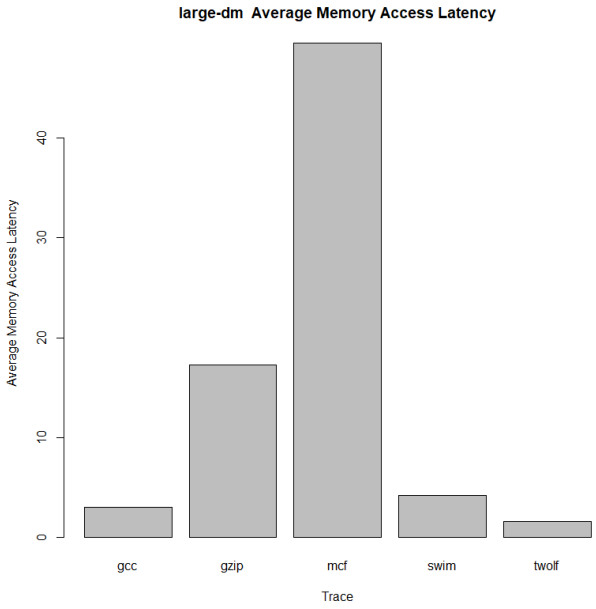
- `medium-dm.conf`



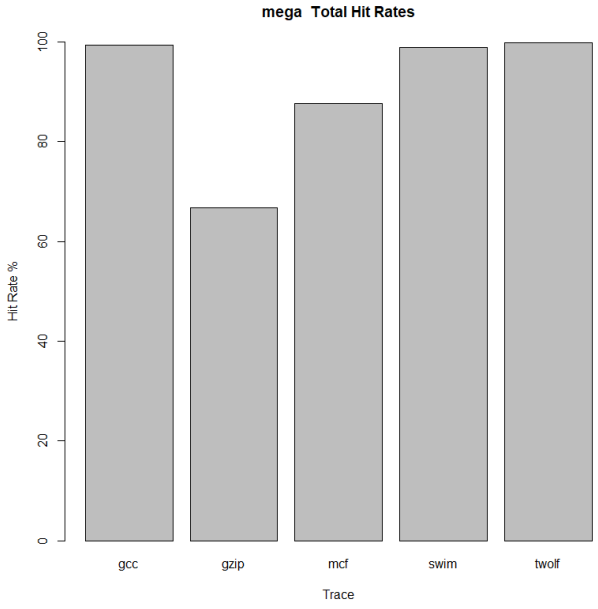


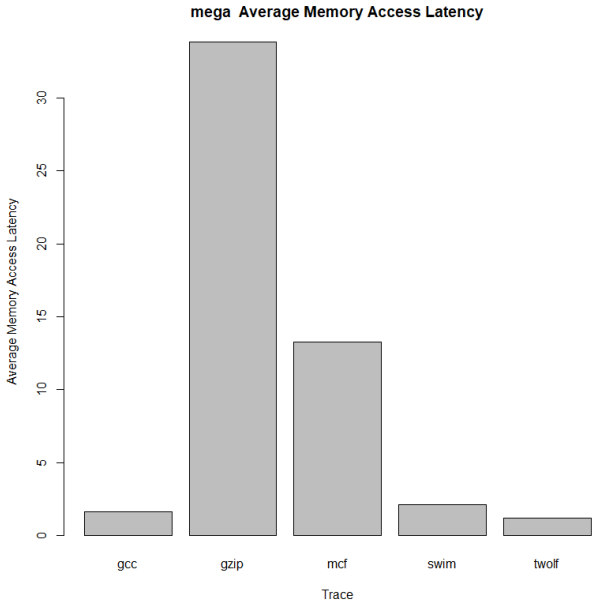
- `large-dm.conf`





- `mega.conf`





References

[1] *Cache Simulator*. URL: https://coffeebeforearch.github.io/2020/12/16/cache-simulator.html.

[2] *printbin utility function*. URL: https://www.geeksforgeeks.org/binary-representation-of-a-given-number/.

4 Appendix

- `2way-wa.conf`

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 98.76
Load Hit Percentage: 99.44
Store Hit Percentage: 97.68
Total Run Time: 1979625 cycles
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.83
Load Hit Percentage: 50.26
Store Hit Percentage: 99.88
Total Run Time: 12094719 cycles
==> traces/mcf.trace.out <==
Total Hit Percentage: 75.24
Load Hit Percentage: 96.15
Store Hit Percentage: 75.06
Total Run Time: 13439159 cycles
==> traces/swim.trace.out <==
Total Hit Percentage: 97.85
Load Hit Percentage: 99.70
Store Hit Percentage: 92.92
Total Run Time: 1625116 cycles
==> traces/twolf.trace.out <==
Total Hit Percentage: 99.66
Load Hit Percentage: 99.87
Store Hit Percentage: 99.10
Total Run Time: 1565636 cycles
Average Memory Access Latency: 1.24
```

- **2way-nwa.conf**

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 93.83
Load Hit Percentage: 98.76
Store Hit Percentage: 85.90
Total Run Time: 3734778 cycles
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.71
Load Hit Percentage: 50.24
Store Hit Percentage: 99.58
Total Run Time: 12131910 cycles
==> traces/mcf.trace.out <==
Total Hit Percentage: 1.05
Load Hit Percentage: 95.61
Store Hit Percentage: 0.27
Total Run Time: 50664176 cycles
==> traces/swim.trace.out <==
Total Hit Percentage: 92.61
Load Hit Percentage: 99.08
Store Hit Percentage: 75.31
Total Run Time: 2722768 cycles
==> traces/twolf.trace.out <==
Total Hit Percentage: 98.86
Load Hit Percentage: 99.80
Store Hit Percentage: 96.35
Total Run Time: 1830734 cycles
Average Memory Access Latency: 1.79
```

- **4way-fifo.conf**

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 98.72
Load Hit Percentage: 99.38
Store Hit Percentage: 97.66
Total Run Time: 1862437 cycles
```

```
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.83
Load Hit Percentage: 50.26
Store Hit Percentage: 99.88
Total Run Time: 8903019 cycles
```

```
==> traces/mcf.trace.out <==
Total Hit Percentage: 75.24
Load Hit Percentage: 96.17
Store Hit Percentage: 75.06
Total Run Time: 9837530 cycles
```

```
==> traces/swim.trace.out <==
Total Hit Percentage: 97.82
Load Hit Percentage: 99.66
Store Hit Percentage: 92.91
Total Run Time: 1499406 cycles
```

```
==> traces/twolf.trace.out <==
Total Hit Percentage: 99.65
Load Hit Percentage: 99.86
Store Hit Percentage: 99.09
Total Run Time: 1532710 cycles
Average Memory Access Latency: 1.17
```

- **4way-rand.conf**

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 98.71
Load Hit Percentage: 99.37
Store Hit Percentage: 97.64
Total Run Time: 1866700 cycles
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.83
Load Hit Percentage: 50.26
Store Hit Percentage: 99.88
Total Run Time: 8903019 cycles
==> traces/mcf.trace.out <==
Total Hit Percentage: 75.24
Load Hit Percentage: 96.17
Store Hit Percentage: 75.06
Total Run Time: 9837579 cycles
==> traces/swim.trace.out <==
Total Hit Percentage: 97.79
Load Hit Percentage: 99.61
Store Hit Percentage: 92.91
Total Run Time: 1504698 cycles
==> traces/twolf.trace.out <==
Total Hit Percentage: 99.65
Load Hit Percentage: 99.86
Store Hit Percentage: 99.09
Total Run Time: 1532955 cycles
Average Memory Access Latency: 1.17
```

- **small-dm.conf**

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 93.85
Load Hit Percentage: 96.40
Store Hit Percentage: 89.74
Total Run Time: 3093758 cycles
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.70
Load Hit Percentage: 50.22
Store Hit Percentage: 99.58
Total Run Time: 8933154 cycles
==> traces/mcf.trace.out <==
Total Hit Percentage: 1.02
Load Hit Percentage: 91.64
Store Hit Percentage: 0.27
Total Run Time: 36284447 cycles
==> traces/swim.trace.out <==
Total Hit Percentage: 91.61
Load Hit Percentage: 97.36
Store Hit Percentage: 76.24
Total Run Time: 2422027 cycles
==> traces/twolf.trace.out <==
Total Hit Percentage: 98.27
Load Hit Percentage: 99.05
Store Hit Percentage: 96.17
Total Run Time: 1860667 cycles
Average Memory Access Latency: 1.85
```

- **medium-dm.conf**

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 95.36
Load Hit Percentage: 98.17
Store Hit Percentage: 90.83
Total Run Time: 2712685 cycles
```

```
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.70
Load Hit Percentage: 50.23
Store Hit Percentage: 99.58
Total Run Time: 8931537 cycles
```

```
==> traces/mcf.trace.out <==
Total Hit Percentage: 1.04
Load Hit Percentage: 93.18
Store Hit Percentage: 0.27
Total Run Time: 36278469 cycles
```

```
==> traces/swim.trace.out <==
Total Hit Percentage: 92.92
Load Hit Percentage: 98.83
Store Hit Percentage: 77.14
Total Run Time: 2227497 cycles
```

```
==> traces/twolf.trace.out <==
Total Hit Percentage: 98.72
Load Hit Percentage: 99.55
Store Hit Percentage: 96.52
Total Run Time: 1753014 cycles
Average Memory Access Latency: 1.63
```

- **large-dm.conf**

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 95.83
Load Hit Percentage: 98.72
Store Hit Percentage: 91.18
Total Run Time: 2592684 cycles
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.71
Load Hit Percentage: 50.23
Store Hit Percentage: 99.58
Total Run Time: 8930851 cycles
==> traces/mcf.trace.out <==
Total Hit Percentage: 1.04
Load Hit Percentage: 93.24
Store Hit Percentage: 0.27
Total Run Time: 36278126 cycles
==> traces/swim.trace.out <==
Total Hit Percentage: 93.43
Load Hit Percentage: 99.37
Store Hit Percentage: 77.57
Total Run Time: 2151988 cycles
==> traces/twolf.trace.out <==
Total Hit Percentage: 98.84
Load Hit Percentage: 99.69
Store Hit Percentage: 96.59
Total Run Time: 1724447 cycles
Average Memory Access Latency: 1.57
```

- **mega.conf**

```
==> traces/gcc.trace.out <==
Total Hit Percentage: 99.35
Load Hit Percentage: 99.69
Store Hit Percentage: 98.80
Total Run Time: 1873992 cycles
==> traces/gzip.trace.out <==
Total Hit Percentage: 66.85
Load Hit Percentage: 50.26
Store Hit Percentage: 99.93
Total Run Time: 16872369 cycles
==> traces/mcf.trace.out <==
Total Hit Percentage: 87.61
Load Hit Percentage: 97.59
Store Hit Percentage: 87.53
Total Run Time: 9931529 cycles
==> traces/swim.trace.out <==
Total Hit Percentage: 98.86
Load Hit Percentage: 99.82
Store Hit Percentage: 96.29
Total Run Time: 1518049 cycles
==> traces/twolf.trace.out <==
Total Hit Percentage: 99.80
Load Hit Percentage: 99.91
Store Hit Percentage: 99.53
Total Run Time: 1545374 cycles
Average Memory Access Latency: 1.20
```