Lab 2

Nicholas Smith

1. Introduction

In this lab, a multi-configuration cache simulator will be programmed in C. This simulator will then be used to compare cache configurations by comparing results of running a suite of programs through each configuration and comparing generated memory statistics.

2. Method

2.1 Cache Construction

To implement the cache, a variety of configuration options must be considered: Line size, Associativity, Data Size, Replacement policy, and Write miss policy. Our cache must be capable of maintaining these parameters and their behavior as it operates. A structure cache_t was created to maintain this data as well as the address scheme and sets within the cache.

For each line in a trace file, the line is parsed into an instruction, address, and a counter for instructions since the last memory access is updated. The address in converted to an address_t struct which splits the address string into block, set, and tag by cache specifications.

2.2 Testing

To test the cache, a print_cache() function was written to verify that the cache was initialized as intended. This function also displayed the determined number of bits for each field of an address as well as the total number of lines, sets, and the lines per set. With this information, the cache initialization was verified.

By modifying the sample.conf file, each possible combination of associativity, replacement policy, and write miss policy were tested via the gcc.trace file as a standard.

With a direct mapped cache, the lowest hit rate was expected. A fully associative cache was expected to have the highest hit rate. N-way associative caches were expected to be somewhere in between direct mapped and fully associative. This proved to be true in testing.

The following comparisons of associativity were made with a cache configuration of 8 byte line size, 16KB data size, FIFO replacement, and no write allocate. The trace file was gcc.trace.

Associativity	Hit Rate
0 (FA)	0.91021
1 (DM)	0.83178
4 (N-Way)	0.884807

FIFO versus random replacement isn't easily tested due the luck of the draw nature of random replacement, instead it was simply verified that either scheme produced a satisfactory result.

With a write allocate policy, we expect to see a higher hit rate than no write allocate as data is brought into the cache for future use.

The following comparison of write policies was made with a cache configuration of 8 byte line size, 4 way set associativity, 16KB data size, and FIFO replacement. The trace file was gcc.trace.

Write Policy	Hit Rate
0 (no write allocate)	0.885
1 (write allocate)	0.931

As we can see, the cache is behaving according to expectations.

3. Results and Discussion

Due to the fact that 40 simulations were conducted in this lab, the following notation will be used to improve readability in graphics to denote trace files and configurations for each each simulation:

"trace-n" where trace is the name of the trace file and n is a reference to a configuration.

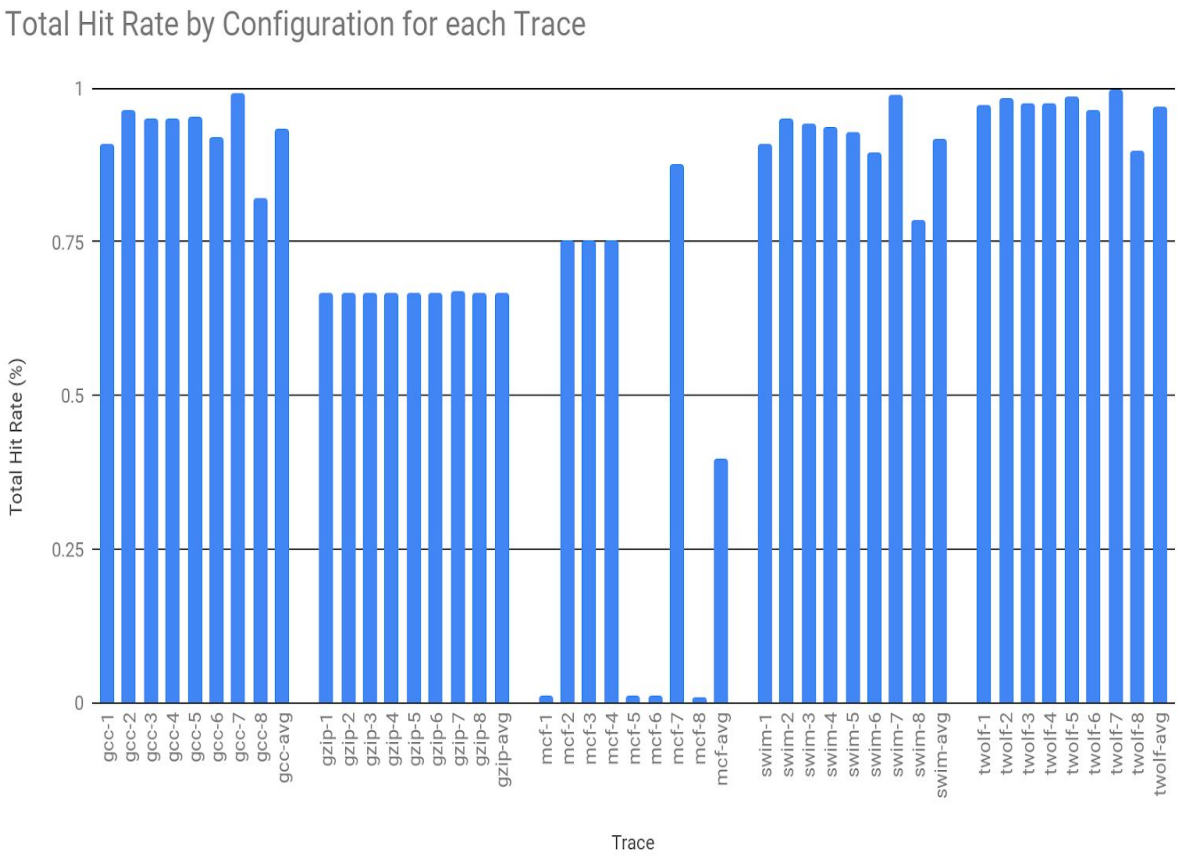
For example, gcc-1 denotes the gcc trace ran with the 2way-nwa(1) configuration.

The following table is a legend of “n” values for each configuration as well as the details behind them:

n	Configuration	Block Size	Associativity	Data Size	Replace	Miss Penalty	Write Allocate
1	2way-nwa	32	2	128	1	70	0
2	2way-wa	32	2	128	1	70	1
3	4way-fifo	32	4	64	1	50	1
4	4way-rand	32	4	64	0	50	1
5	large-dm	8	1	256	1	50	1
6	medium-dm	8	1	32	1	50	1
7	mega	64	8	4096	1	100	1
8	small-dm	8	1	8	1	50	1

3.1 Total Hit Rate

The following graph represents total hit rate by configuration for each trace, an average is provided on the right of each group for comparison:



For each trace, mega(7) consistently had the highest hit rate of any configuration, posting a maximum hit rate of 99.8% in the twolf trace. With mega(7) being the largest, highest associativity cache tested, this makes sense.

Similarly small-dm(8) consistently had the lowest hit rate of any configuration, posting a 0.89% hit rate in the mcf trace. This also is reasonable as it is the smallest cache. It also features direct mapping to further cause collisions.

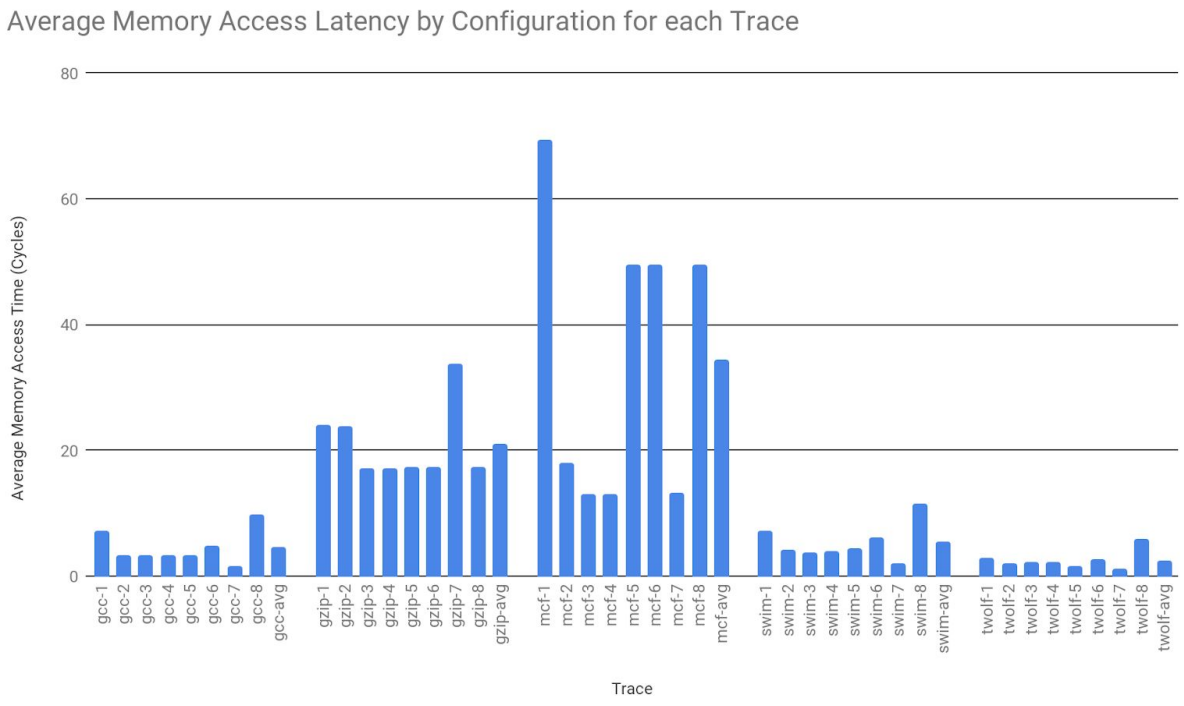
The effects of write allocate, depicted by 2way-nwa(1) and 2way-wa(2), always reduce the miss rate of the cache when write allocating. In mcf this effect is especially apparent.

The replacement policy generally does not matter in terms of hit rate, usually affecting the overall value by less than 1%. However, during execution the overhead of FIFO was definitely noticeable with the program sometimes taking several seconds to complete a FIFO trace whereas random was almost instantaneous.

The cache configuration seems to have the least effect on the gzip trace with each configuration posting a 66.x% hit rate. Meanwhile, mcf was the most influenced by the configuration, posting a hit rate anywhere between 0.89% and 87.6% for the small-dm(8) and mega(7) configurations respectively. Mcf tended to post high hit rates with set associative configurations. Meanwhile, direct mapped caches had abysmally low hit rates. Interestingly, 2way-nwa(1), a set associative cache, also had a low hit rate. This points to the addresses in mcf being close together and causing many conflicts, especially in direct mapped caches. The low hit rate of 2way-nwa(1) also indicates that this was a write heavy program.

3.2 Average Memory Access Latency

The following graph represents Average Memory Access Latency (AMAL) by configuration for each trace, an average is provided on the right of each group for comparison:



In every case except for gzip and mcf, small-dm(8) had the worst AMAL of any configuration. This follows logically from it having the lowest hit rate of any configuration. However, small-dm(8) has a miss penalty of 50. In gzip, where the hit rate was 66.x% across the board, configs with larger penalties stuck out. In mcf, small-dm(8)

had a comparable hit rate to that of 2way-nwa(1) yet the higher miss penalty of 70 caused 2way-nwa(1) to be the worst performer.

The mega(7) configuration typically had the lowest AMAL with the exceptions of gzip and mcf. Mega(7) has a miss penalty of 100 cycles, the largest of any configuration by a margin of 30 cycles. Thus in gzip, where the hit rate was 66.x% across all configurations, mega(7) logically had the highest AMAL. In the mcf trace, 4way-fifo(3) posted the lowest AMAL of 13.1545 cycles. Mega(7) posted 13.266 cycles. This inconsistency is once again explained by the high miss penalty of mega(7).

Write allocate, as depicted by the results of 2way-nwa(1) and 2way-wa(2), aids in reducing AMAL.

Replacement policy, as depicted by the results of 4way-fifo(3) and 4way-rand(4), does not appear to have a significant effect on reducing AMAL.

N-way set associativity tends to reduce AMAL when compared to direct mapping, this is best shown in the mcf results where, with the exception of 2way-nwa(1), the direct mapped caches had a higher AMAL than any set associative cache.

4. Conclusion

In this lab, a cache simulator was designed in C to test various cache designs with a number of programs. After verifying the simulator's correctness, 5 trace files were used to generate run time data with a variety of configurations. It was found that in general, set associative caches had better performance. It was also found that allocation on write misses improves performance. However, random versus first in first out replacement policies did not have any significant effects on performance.