EECE7352 - COMPUTER ARCHITECTURE: HOMEWORK 4

ANNA DEVRIES[1]

8 November 2019

CONTENTS

LIST OF FIGURES

[1] *Department of Electrical and Computer Engineering, Northeastern University, Boston, United States*

LIST OF TABLES

1 PART A

1.1 Cache Architecture Reversing

Description: Your job is to generate reference streams in Dinero's din format (see description provided above) and produce the following (where you are asked to generate an address stream, submit in your report enough of the trace so that we can understand the pattern you are using – do not submit the full trace since it will only waste paper): i) Assume that you have a direct-mapped 64KB instruction cache with a (64B block size). Generate an address stream that will touch every cache line once, but no more than once. ii) Assume the same instruction cache organization as in (1), but now the instruction cache is 2-way set associative, with LRU replacement. The total cache space is still 64KB with a 64B block size, but now you have 1/2 the number of indices. Generate an address stream that touches every cache index only 7 times, producing 3 misses and 4 hits, but only accesses 3 unique addresses per index. iii) Repeat part 2, but now produce 5 misses and 5 hits, again with only 3 unique address per index, but produce a interleaving pattern of Miss-Hit-Miss-Hit-Miss-Hit-... iv) Assume that you have a 4-way set associate 64KB data cache with a (32B block size). Generate an address stream that will generate 5 hits and 3 misses. Make sure that your stream includes both loads and stores. v) Given a partially specified cache organization, generate one or multiple instruction reference streams that can detect the set associativity and the total size of an instruction cache. Assume that you know that the block size is 16B and that the cache size will be no larger than 32KB. Generate the stream(s) and discuss how you figured out the total size and the associativity. vi) Repeat part 4, but now generate a stream that will determine the replacement algorithm. Again, discuss how you determined this.

1.2 Solution

1) For the first question, I utilized the below code to create an instruction stream that placed a single address on each cache block. This produced a 100% miss rate since each cache block was only touched once, see Figure 1 for the results and instruction stream pattern sequence. I ran the simulation with the following syntax: gcc stream_generator.c -o stream_gen ./stream_gen > stream_gen.txt dineroIV -l1-isize 64K -l1-ibsize 64 -l1-dsize 64K -l1-dbsize 64 -informat d < stream_gen.txt.

```
int partOne(){
    long addr = 0x000000;
    int i;
    for(i = 0; i < 1024; i++){
        printf("2 %x0 \n", addr);
        addr = addr + 64;
    }
    return 0;
}
```

Algorithm 1: Part A1

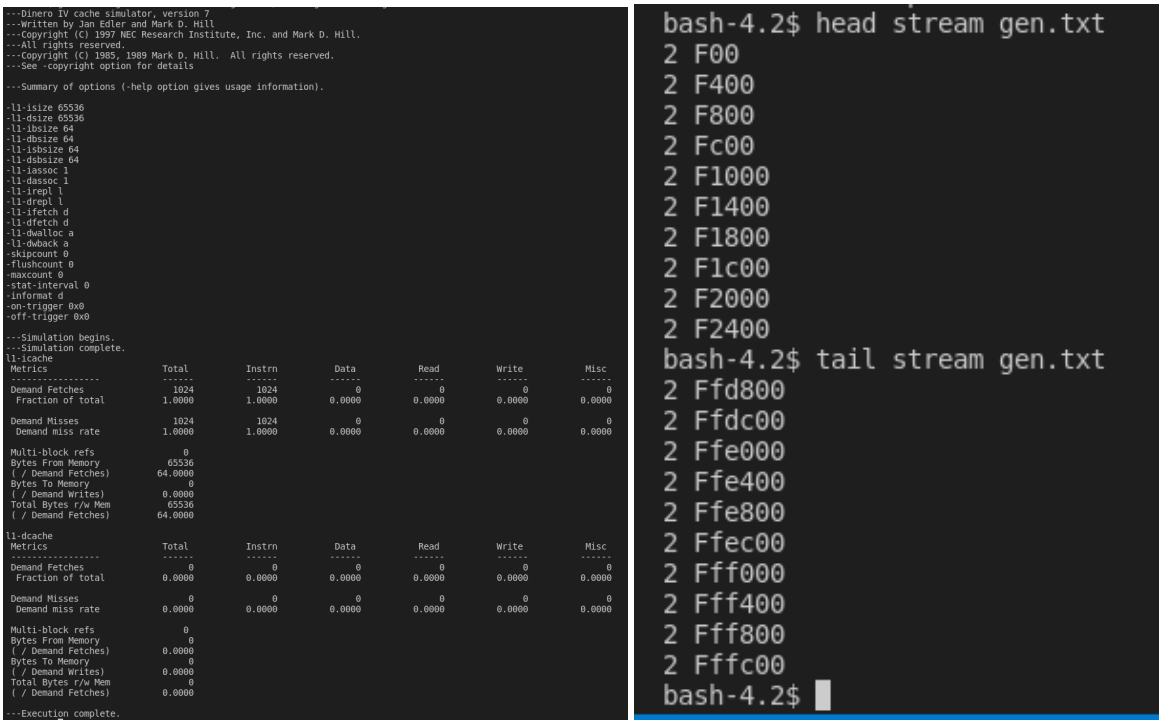

Figure 1: Part A1 results [A] and part A1 instruction pattern [B].

2) For the second question, I utilized the below code to create an instruction stream that touched each cache block 7 times - producing 3 misses and 4 hits. This produced a 43% miss rate, see Figure 2 for the results and instruction stream pattern sequence. I ran the simulation with the following syntax: gcc stream_generator.c -o stream_gen ./stream_gen > stream_gen.txt dineroIV -l1-isize 64K -l1-ibsize 64 -l1-dsize 64K -l1-dbsize 64 -l1-irepl l -l1-iassoc 2 -informat d < stream_gen.txt.

```
int partTwo(){
    long addr = 0x000000;
    int i;
    for(i = 0; i < 512; i++){
        printf("2 F%x0 \n", addr);
        printf("2 F%x0 \n", addr);
        printf("2 E%x0 \n", addr);
        printf("2 E%x0 \n", addr);
        printf("2 D%x0 \n", addr);
        printf("2 D%x0 \n", addr);
        printf("2 D%x0 \n", addr);
        addr = addr + 64;
    }
    return 0;
}
```

Algorithm 2: Part A2

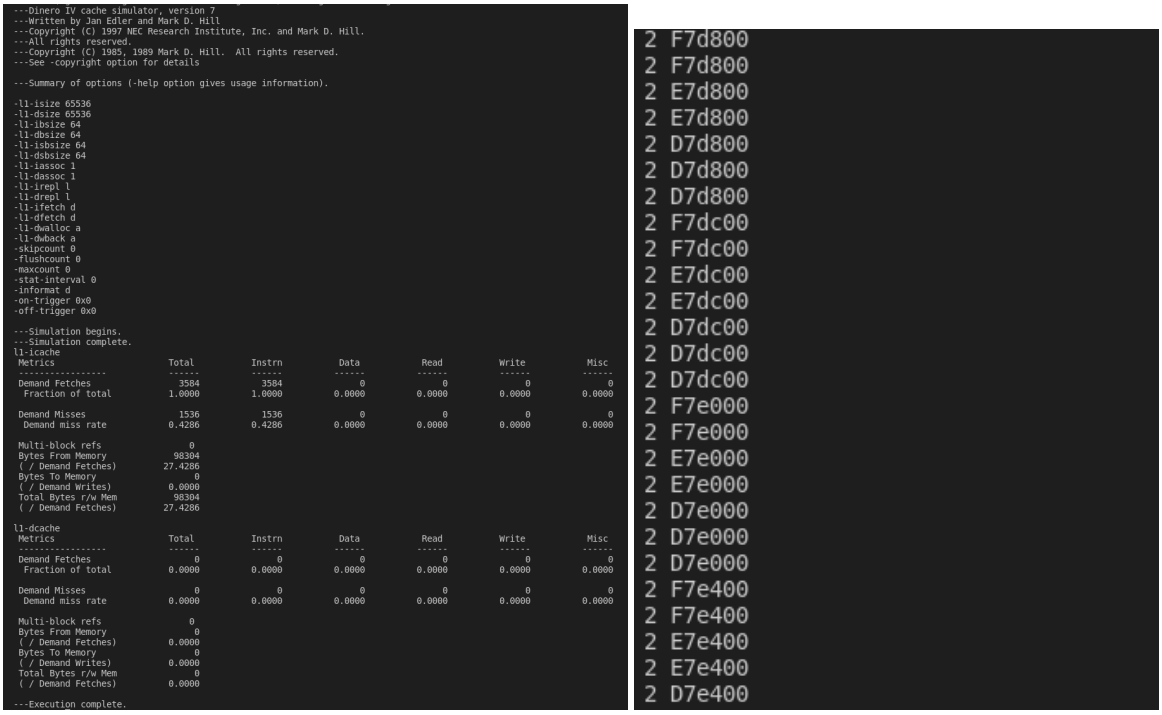

Figure 2: Part A2 results [A] and part A2 instruction pattern [B].

3) For the third question, I utilized the below code to create an instruction stream that touched each cache block 10 times - producing 5 misses and 5 hits. This produced a 50% miss rate, see Figure 3 for the results and instruction stream pattern sequence. I ran the simulation with the following syntax: gcc stream_generator.c -o stream_gen ./stream_gen > stream_gen.txt dineroIV -l1-isize 64K -l1-ibsize 64 -l1-dsize 64K -l1-dbsize 64 -l1-irepl l -l1-iassoc 2 -informat d < stream_gen.txt.

```c
int partThree(){
    long addr = 0x000000;
    int i;
    for(i = 0; i < 512; i++){
        printf("2 F%x0 \n", addr);
        printf("2 F%x0 \n", addr);
        printf("2 D%x0 \n", addr);
        printf("2 F%x0 \n", addr);
        printf("2 E%x0 \n", addr);
        printf("2 F%x0 \n", addr);
        printf("2 D%x0 \n", addr);
        printf("2 D%x0 \n", addr);
        printf("2 E%x0 \n", addr);
        printf("2 E%x0 \n", addr);
        addr = addr + 64;
    }
    return 0;
}
```

Algorithm 3: Part A3

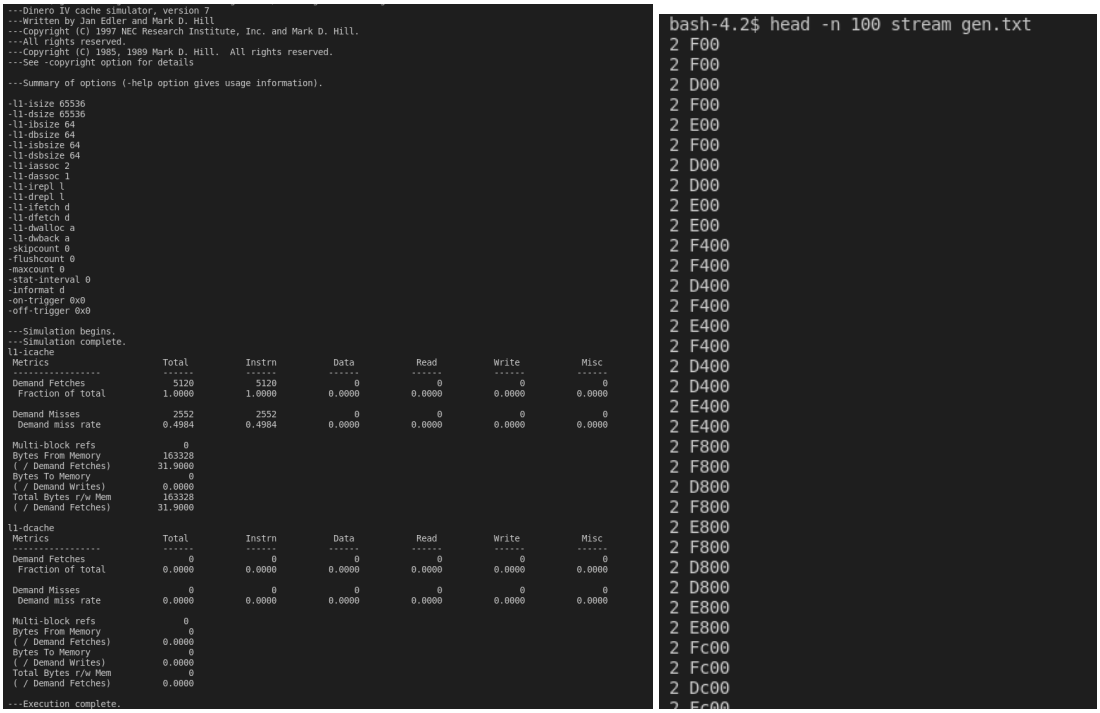

Figure 3: Part A3 results [A] and part A3 instruction pattern [B].

4) For the fourth question, I utilized the below code to create an instruction stream that touched each cache block 8 times - producing 3 misses and 5 hits with memory access instructions. This produced a 38% miss rate, see Figure 3 for the results and instruction stream pattern sequence. I ran the simulation with the following syntax: gcc stream_generator.c -o stream_gen ./stream_gen > stream_gen.txt dineroIV -l1-isize 64K -l1-ibsize 32 -l1-dsize 64K -l1-dbsize 32 -l1-irepl l -l1-iassoc 4 -informat d < stream_gen.txt.

```
int partFour(){
    long addr = 0x000000;
    int i;
    for(i = 0; i < 256; i++){
        printf("1 F%x0 \n", addr); // 0 = read data, 1 = write data
        printf("0 F%x0 \n", addr);
        printf("0 F%x0 \n", addr);
        printf("0 F%x0 \n", addr);
        printf("1 D%x0 \n", addr);
        printf("0 D%x0 \n", addr);
        printf("0 D%x0 \n", addr);
        printf("0 E%x0 \n", addr);
        addr = addr + 32;
    }
    return 0;
}
```

Algorithm 4: Part A4

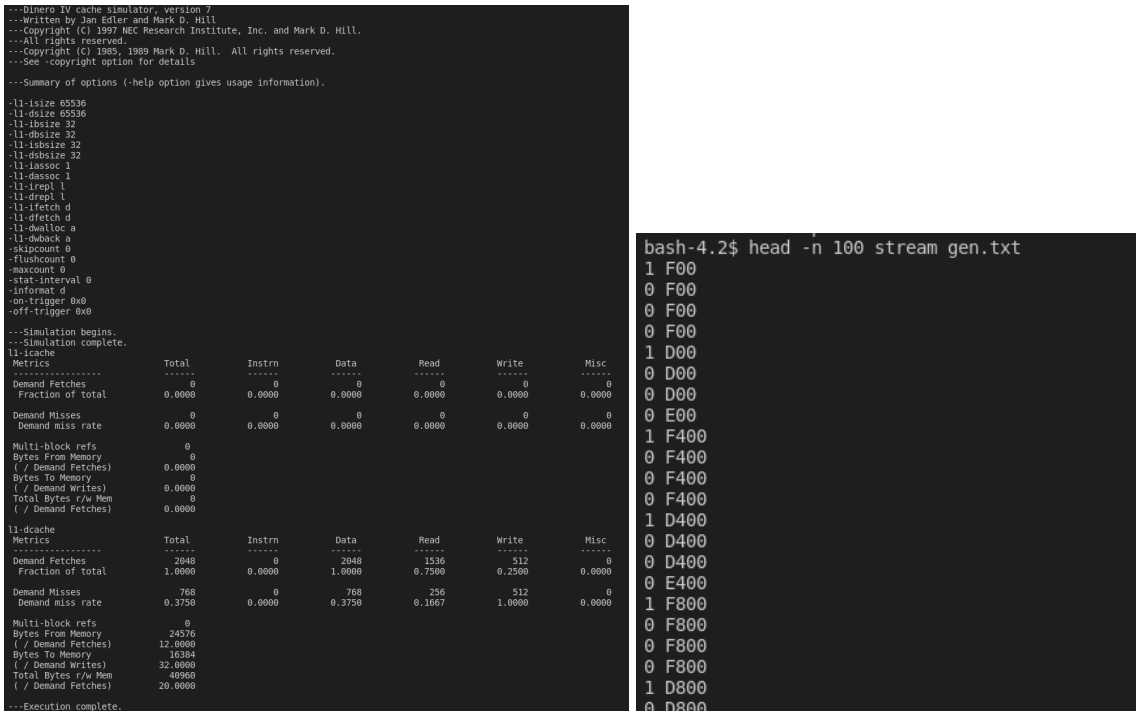

Figure 4: Part A4 results [A] and part A4 instruction pattern [B].

5) For the fifth question, I utilized two different instruction streams. The first stream uses a pattern to determine set-associativity, the results are place in Table 1. The set-associatitivy for the cache architecture may be inferred from miss rate. The higher miss rate refers to lower levels of associativity, whereas lower miss rate refers to higher levels of associativity. The pattern created to determine this is F F D F E D. For 4-way set associative, F, D, and E may be stored without overwriting each other; however, lower set-associativities will have some level of overlap.

Table 1: Set Associativity versus Miss Rate.

Associativity	Miss Rate
Direct-Mapped	100%
2-Way Set Associative	80%
4-Way Set Associative	60%

Next, I used the second instruction stream to determine the cache size, results are placed in Table 2. For the larger cache size, the values are less often overwritten. Addresses are increased up to 256 and then decremented down to 0 (repeated 8 times, totaling in 2048). 2048 is the largest amount of addresses available to be place in a 32KB cache size, 16B block size cache. A 32KB cache will have more hits because the cache is capable of holding more values, whereas the smaller cache sizes will overwrite information and the decrementing for-loop will not render "hits". The pattern of each instruction stream is provided in Figure 6.

```
int partFive_assoco(){
    long addr = 0x000000;
    int i;
    for(i = 0; i < 2048; i++){
        printf("0 %x0 \n", addr);
        printf("2 %x0 \n", addr);
```

Table 2: Cache Size versus Miss Rate.

Cache Size	Miss Rate
32 KB	78%
16 KB	89%
8 KB	95%
4 KB	97%
2 KB	99%

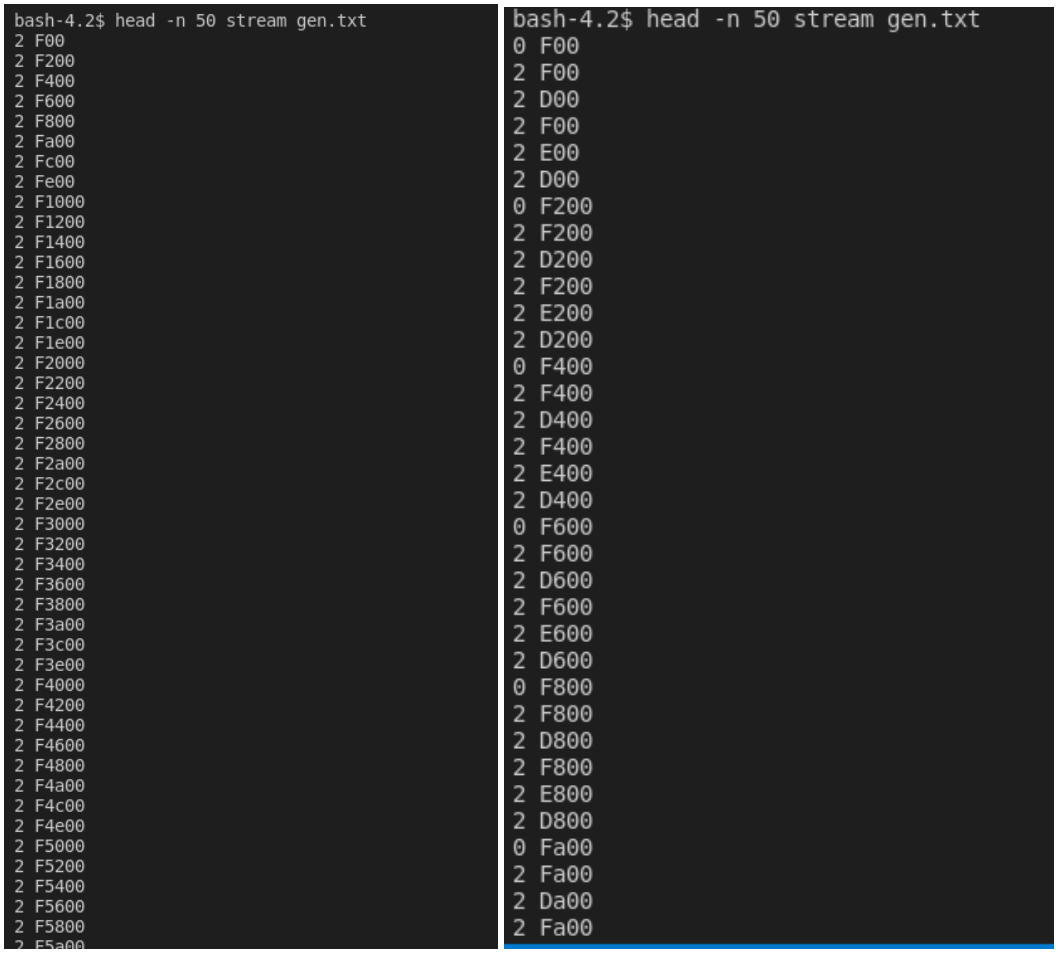

Figure 5: Part A5 cache size determination pattern [A] and part A5 association determination pattern [B].

```c
        printf("2 D%x0 \n", addr);
        printf("2 F%x0 \n", addr);
        printf("2 E%x0 \n", addr);
        printf("2 D%x0 \n", addr);
        addr = addr + 32;
    }
    return 0;
}

int partFive_size(){
    long addr = 0x0000000;
    int i;
    for(i = 0; i < 256; i++){
        printf("2 F%x0 \n", addr);
        addr = addr + 32;
    }
    for(i = 256; i > 0; i--){
        printf("2 F%x0 \n", addr);
        addr = addr - 32;
    }
    for(i = 0; i < 256; i++){
        printf("2 F%x0 \n", addr);
        addr = addr + 32;
    }
    for(i = 256; i > 0; i--){
        printf("2 F%x0 \n", addr);
        addr = addr - 32;
    }
    for(i = 0; i < 256; i++){
        printf("2 F%x0 \n", addr);
        addr = addr + 32;
    }
    for(i = 256; i > 0; i--){
        printf("2 F%x0 \n", addr);
        addr = addr - 32;
    }
    for(i = 0; i < 256; i++){
        printf("2 F%x0 \n", addr);
        addr = addr + 32;
    }
    for(i = 256; i > 0; i--){
        printf("2 F%x0 \n", addr);
        addr = addr - 32;
    }
    return 0;
}
```

Algorithm 5: Part A5

6) Replacement algorithms determine which values are removed from the cache. The two types studied for this homework were LRU and FIFO. LRU means least recently used. This refers to removing the value with the least amount of accesses from the cache. An example is cleaning my garage: when I decide which items to throw out to make room for more items, I do not throw away the oldest thing necessarily. Rather, I throw away the least used item. However, FIFO stands for first in first out. This refers to removing the oldest value in the cache. This is similar to a tunnel: the first car to enter the tunnel will be the first car to exit the tunnel.
With these ideas, I assumed a 32KB cache size with a 16B block size and two-way set associative for simulating the cache. Following the concepts above, I created an instruction stream pattern that would render the two different replacement algorithms distinct. On the same line of cache, I added the following distinct elements: A B A C C B. For a LRU replacement algorithm, this results in miss, miss, hit, miss, hit and miss (67% miss rate). For a FIFO replacement algorithm, this results in miss, miss, hit, miss hit, and hit (50% miss rate).

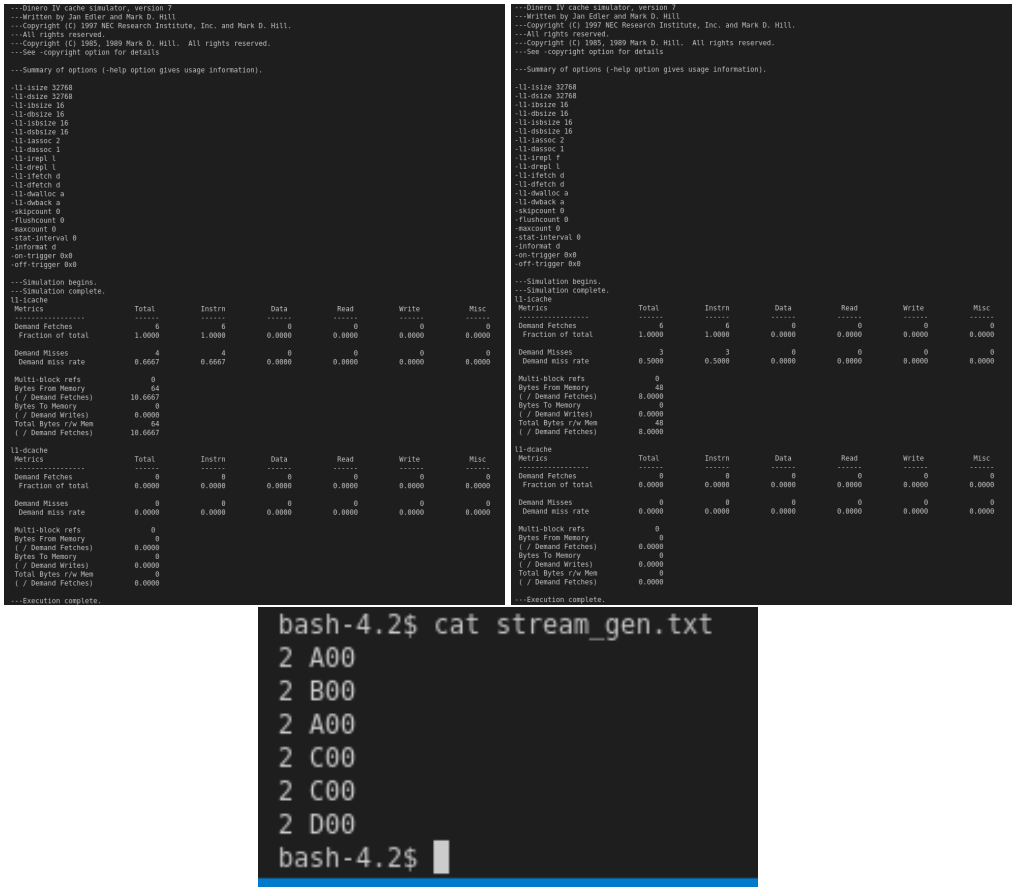

Figure 6: Part A6 LRU results [A] and part A6 FIFO results [B] and part A6 instruction stream pattern [C].

```c
int partSix(){
    long addr = 0x000000;
    int i;

    printf("2 A%x0 \n", addr);      // LRU: M, M, H, M, H, M
    printf("2 B%x0 \n", addr);      // FIFO: M, M, H, M, H, H
    printf("2 A%x0 \n", addr);
    printf("2 C%x0 \n", addr);
    printf("2 C%x0 \n", addr);
```

```
10      printf("2 B%xo \n", addr);
11
12      return 0;
13  }
```

Algorithm 6: Part A6

2 PART B

2.1 Cache Architecture Simulation

Description: For the next part of this assignment, using the trace provided as input to DineroIV, model an instruction cache and a data cache with a combined total cache space of 16KB (for a split cache, assume a 8KB instruction cache and a 8KB data cache. The block size should be varied (16B, 64B and 256B) and the associativity should be varied (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 are a total of 12 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. Copy the trace file provided to the COE Linux system to run your study, but please gzip or compress the file when you are not using it.

2.2 Solution

Tables 3, 4 and 5 contain the results from each of the 12 simulations. Additionally, Figure 7 shows the two graphs developed for the simulations. In regards to the split data and instruction cache architecture, the data cache decreased with accuracy between the 64B and 256B block size but increased with accuracy between the 16B and 64B block size. Furthermore, the instruction cache increased with accuracy as block size increased throughout the simulation. In regards to the combined cache, miss rate decreased with increasing block size. In both of these cache architectures, 4-way set associativity outperformed 2-way.

Table 3: Split cache (8KB each) architecture simulations.

Simulation	Instr Cache Fetches	Instr Cache Misses (Miss Rate%)
16B block size, 2-way associativity	559159	44658 (8%)
64B block size, 2-way associativity	559159	20883 (3.7%)
256B block size, 2-way associativity	559159	13487 (2.4%)
16B block size, 4-way associativity	559159	41276 (7.4%)
64B block size, 4-way associativity	559159	20496 (3.7%)
256B block size, 4-way associativity	559159	12864 (2.3%)

Table 4: Split cache (8KB each) architecture simulations.

Simulation	Data Cache Fetches	Data Cache Miss (Miss Rate %)
16B block size, 2-way associativity	467428	39810 (8.5%)
64B block size, 2-way associativity	467428	36140 (7.7%)
256B block size, 2-way associativity	467428	40879 (8.8%)
16B block size, 4-way associativity	467428	36015 (7.7%)
64B block size, 4-way associativity	467428	32670 (7%)
256B block size, 4-way associativity	467428	37820 (8.1%)

First, to discuss the set associativity. Higher set associativities are more complex but render a lower miss rate. This improvement in performance is due to the number of available blocks per cache line. In 2-way set associativity, there are only 2 blocks to avoid conflict; however, in 4-way set associativity, there are 4 open spaces per cache line. This change in architecture reduces memory stall cycles and average memory access time.

Table 5: Unified cache (16KB) architecture simulations.

Simulation	Cache Fetches	Cache Miss (Miss Rate %)
16B block size, 2-way associativity	1026587	79713 (7.8%)
64B block size, 2-way associativity	1026587	52744 (5.1%)
256B block size, 2-way associativity	1026587	47960 (4.6%)
16B block size, 4-way associativity	1026587	72590 (7.1%)
64B block size, 4-way associativity	1026587	46919 (4.6%)
256B block size, 4-way associativity	1026587	41449 (4%)

Second, to discuss the cache performance in regards to increased block size. Larger blocks take advantage of spatial locality; however, too large of blocks limits the number of available blocks in the cache which increases conflict misses. The instruction cache and combined cache benefited from the increased block size by increasing the spatial locality; however, the data cache in the split cache architecture maxed its benefits from block size at 64B. The increase in miss rate from 64B to 256B shows that the blocks became too large for the cache size and decreased the number of available blocks.
Third, to discuss the split versus combined cache architecture designs. The performance of the combined cache falls between the performances of the data cache and instruction cache in the split architecture. It is difficult to compare the results specifically to miss rate; however, more importantly, the performance may be compared based upon thrashing. Split cache design generally outperforms a combined cache design because it reduces the number of cache lines replacements. By separating instructions and data, the cache will not be frequently replacing cache lines to remove instructions for data or vice versa.

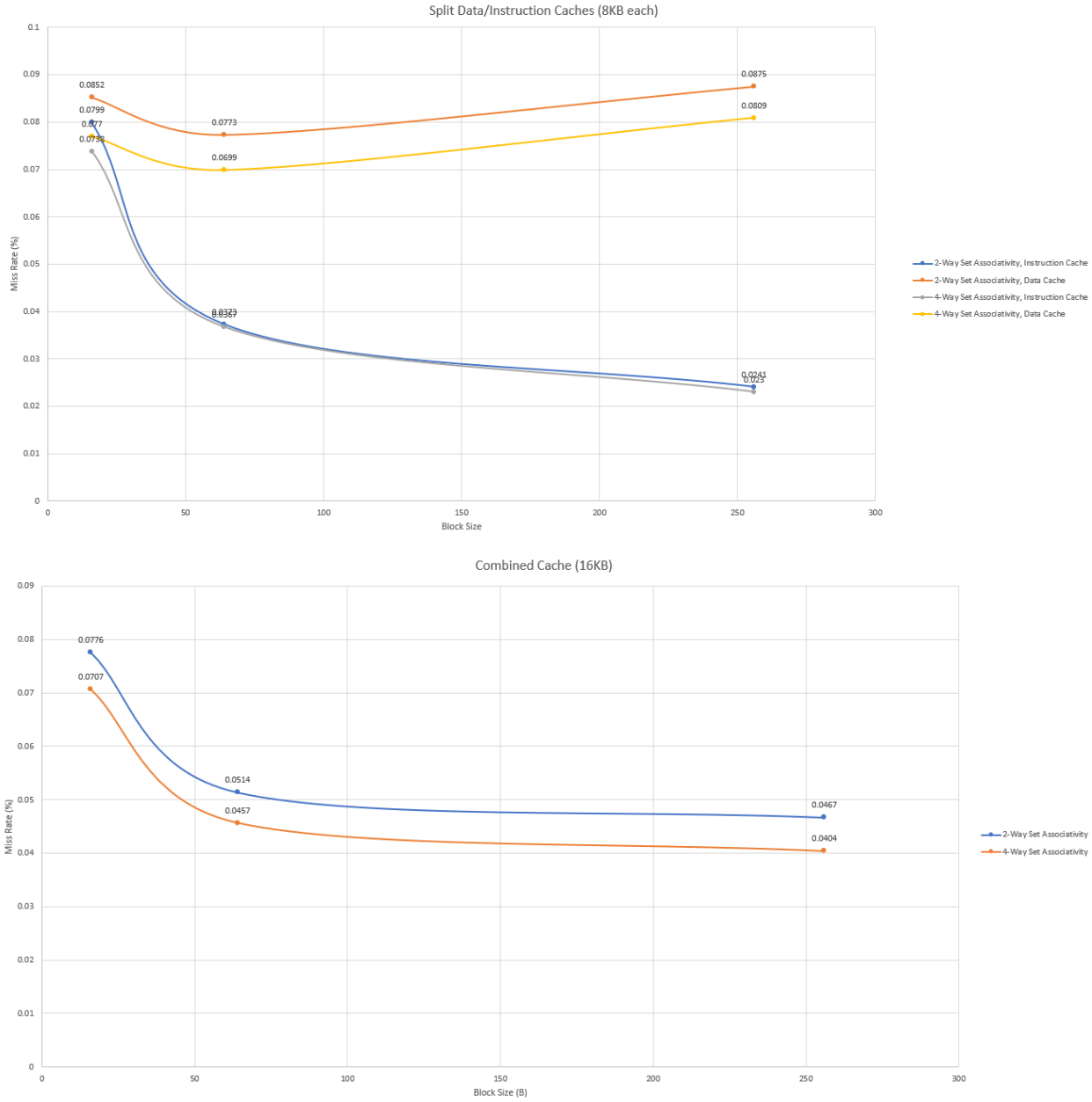

Figure 7: Part B split cache simulations [A] and part B combined cache simulations [B].

3 PART C

3.1 Fetch Policy Comparison

Description: For the next part of this assignment, use the prefetching capabilities provided in dineroIV (the -Tfetch, -pfdist and –pfabort switches). Using the same trace used in part B, study the effects of different 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.

3.2 Solution

For part C, I assumed a split cache architecture design (8KB for the data cache and 8KB for the instruction cache) with 64B block size and 4-way set associativity. I automated this process by creating a bash script and saving the results to a csv file. I tested every option available for each switch. First, I evaluated the miss rate for the cache with each fetch policy individually. Next, I evaluated the miss rate for the cache with each prefetch distance option. For the prefetch distance, fetch policy could not be d, l or s so I tested the prefetch distance option from 0 to 100 with the a, m and t fetch policies. Finally, I evaluated the miss rate for the cache with each prefetch percentage option (0-100%). This switch required the fetch policy to be either a, m, t, l or s so I concurrently tested every prefetch percentage option with those different fetch policies.

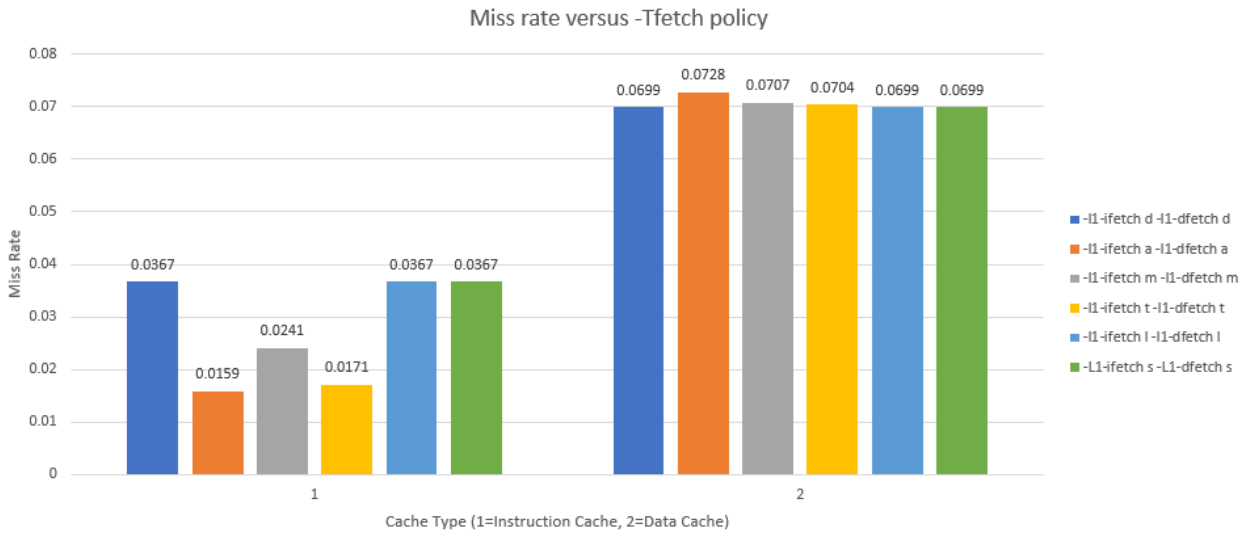

Figure 8: Miss rate versus -Tfetch policy.

Figure 8 provides a visual table comparing the miss rates for each -Tfetch policy. The lowest miss rates for the instruction cache were seen in the always fetch and tagged fetch policies with 1.6% and 1.7% miss rates respectively, rendering a 0.1% difference. For data cache, the always fetch policy had a slightly higher miss rate then the tagged fetch policy, 7.3% and 7% miss rate respectively with a 0.3% difference. Therefore, the tagged policy outperformed all other fetch policies overall.
Figure 9 provides graphs representing the miss rate versus prefetch distance. For all fetch policies, increasing prefetch distance increased the miss rate for both the instruction and data caches. Prefetching refers to fetching the data before it is needed in a program in order to reduce or remove latency. Prefetching distance specifically refers to the distance away the program with prefetch the data. For this specific implementation, prefetching decreases programming accuracy from the stand-alone fetch option.

Figure 10 provides graphs representing the miss rate versus prefetch percentage. The -pfabort policy refers to the prefetch abort percentage. The miss rate increased in the instruction cache but decreased in the data cache for all fetch policies as the -pfabort value increased. While the data cache miss rate improved, it only improved by 0.1% in the tagged fetch policy from 0% to 100% prefetch, whereas the instruction cache miss rate increased by 2%. Due to the overall decline in accuracy, the fetch policy with 0% prefetch and 0 prefetch distance still outperforms other prefetch options. Factors that could cause this are the program may not consist of regular access pattern instructions or the prefetched blocks are not needed. These optimizations were performed on a cache with a LRU replacement algorithm - this specific replacement algorithm handles prefetched and demand blocks separately, eliminating the optimizations from prefetching.

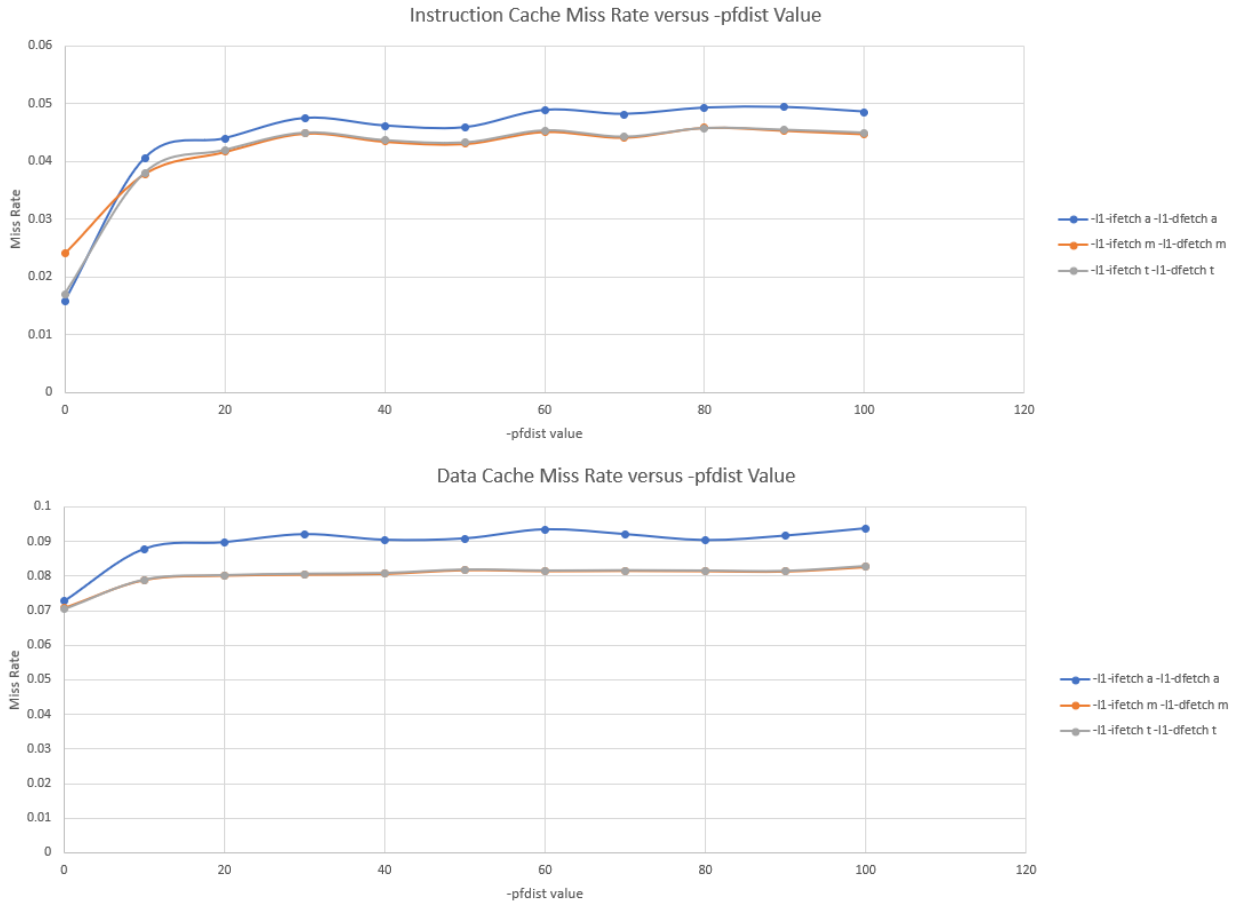

Figure 9: Instruction cache miss rate versus -pfdist value [A] and data cache miss rate versus -pfdist value [B].

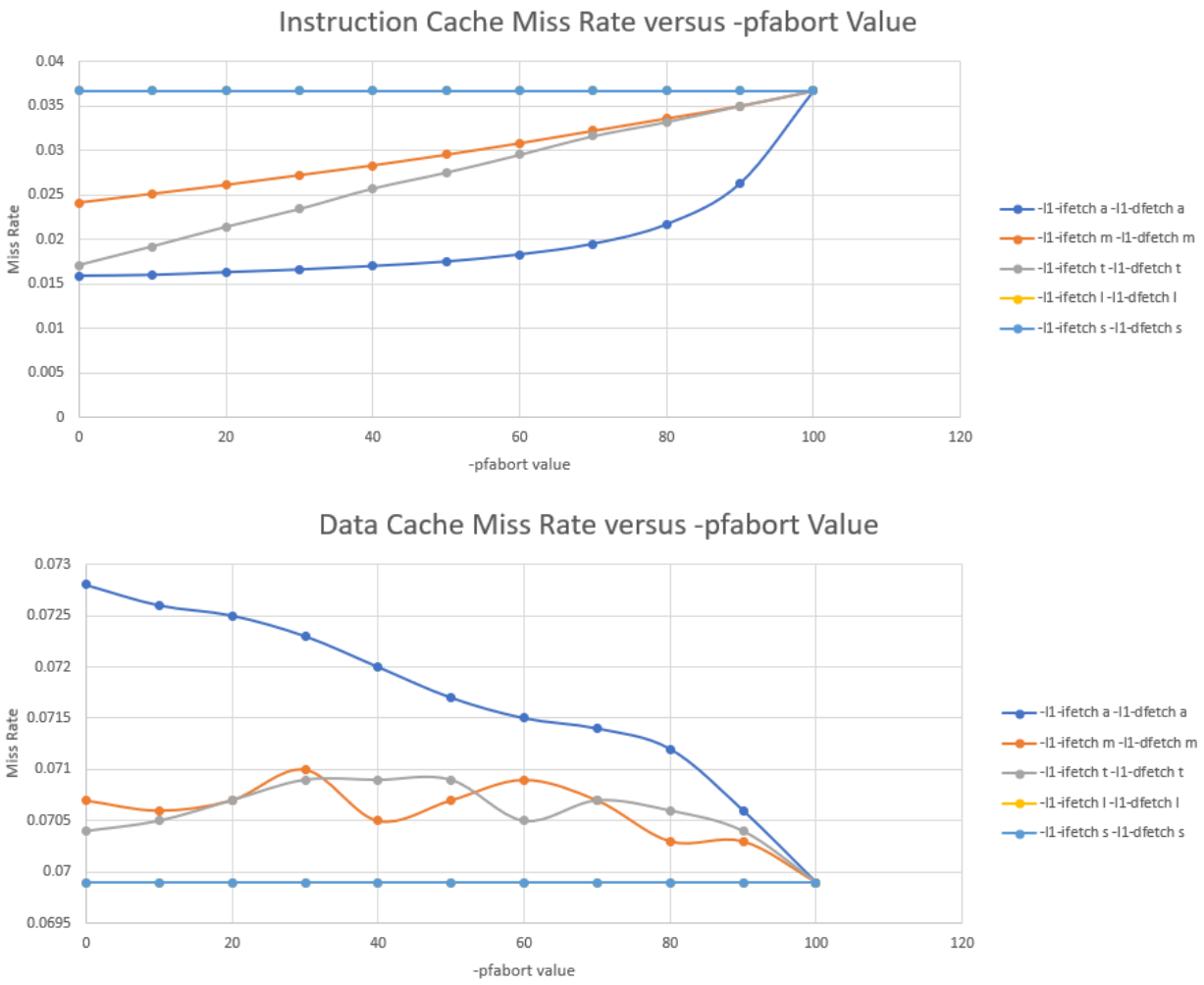

Figure 10: Instruction cache miss rate versus -pfabort value [A] and data cache miss rate versus -pfabort value [B]..

```
#!/bin/bash

# Architecture Variables
split_cache_size='8K'
block_size='64'
cache_level='1'
assoco='4'

## Architecture general setup
## dineroIV -l${cache_level}-isize ${split_cache_size} -l${cache_level}-ibsize ${block_size} -l${cache_level}-dsize ${split_cache_size} -l${cache_level}-dbsize ${block_size} -l${cache_level}-iassoc ${assoco} -l${cache_level}-dassoc ${assoco} -l${cache_level}-ifetch $switch -l${cache_level}-dfetch $switch -informat d < trace | grep 'Demand miss rate' | awk '{print $4}'

## Option Key:
## C single character
## U unsigned decimal integer
## N cache level (1 <= N <= 5)
## T cache type (u=unified, i=instruction, d=data)

# Add variable switches
# -lN-Tfetch C      Fetch policy (d=demand, a=always, m=miss, t=tagged, l=load forward, s=subblock) (default d)
switch_array=( d a m t l s )
for switch in ${switch_array[@]}
do
    echo -l${cache_level}-ifetch $switch -l${cache_level}-dfetch $switch
    dineroIV -l${cache_level}-isize ${split_cache_size} -l${cache_level}-ibsize ${block_size} -l${cache_level}-dsize ${split_cache_size} -l${cache_level}-dbsize ${block_size} -l${cache_level}-iassoc ${assoco} -l${cache_level}-dassoc ${assoco} -l${cache_level}-ifetch $switch -l${cache_level}-dfetch $switch -informat d < trace | grep 'Demand miss rate' | awk '{print $4}'
done

# -lN-Tpfdist U     Prefetch distance (in sub-blocks) (default 1)
# Note: switch will NOT work for "d" fetch policy, so check with all other fetch policies
for ((switch=0;switch<=100;switch+=10));
do
    switch_array2=( a m t )
    for switch2 in ${switch_array2[@]}
    do
        echo -l${cache_level}-ifetch $switch2 -l${cache_level}-dfetch $switch2 -l${cache_level}-ipfdist $switch -l${cache_level}-dpfdist $switch
        dineroIV -l${cache_level}-isize ${split_cache_size} -l${cache_level}-ibsize ${block_size} -l${cache_level}-dsize ${split_cache_size} -l${cache_level}-dbsize ${block_size} -l${cache_level}-iassoc ${assoco} -l${cache_level}-dassoc ${assoco} -l${cache_level}-ifetch $switch2 -l${cache_level}-dfetch $switch2 -l${cache_level}-ipfdist $switch -l${cache_level}-dpfdist $switch -informat d < trace | grep 'Demand miss rate' | awk '{print $4}'
    done
done

# -lN-Tpfabort U    Prefetch abort percentage (0-100) (default 0)
# Note: switch will NOT work for "d" fetch policy, so check with all other fetch policies
for ((i=0;i<=100;i+=10));
do
    switch_array2=( a m t l s )
    for switch2 in ${switch_array2[@]}
    do
        echo -l${cache_level}-ifetch $switch2 -l${cache_level}-dfetch $switch2 -l${cache_level}-ipfabort $i -l${cache_level}-dpfabort $i
        dineroIV -l${cache_level}-isize ${split_cache_size} -l${cache_level}-ibsize ${block_size} -l${cache_level}-dsize ${split_cache_size} -l${cache_level}-dbsize ${block_size} -l${cache_level}-iassoc ${assoco} -l${cache_level}-dassoc ${assoco} -l${cache_level}-ifetch
```

```
    $switch2 -l${cache_level}-dfetch $switch2 -l${cache_level}-ipfabort $i -l${cache_level}-
    dpfabort $i -informat d < trace | grep 'Demand miss rate' | awk '{print $4}'
    done
done
```

Algorithm 7: Part C

4 PART D

4.1 Article Discussion

Description: Read that the attached HPCA paper on Last-Level Cache design. Then answer the following five questions: i.) Discuss the key points in this paper. ii.) Describe two of the experiments presented, and discuss the results. iii.) What did you learn about the workloads studied? vi.) How did cache size benefit or hurt applications with a high degree of data sharing? v.) What is OpenMP, and how do you use this infrastructure to write parallel programs?

4.2 Solution

1) The HPCA paper provides a detailed study of data-sharing and chip-multiprocessor cache analysis. The researchers experimented on different multi-threaded data-mining bioinformatics workloads to determine amount of data cache sharing. This study utilizes a workload that exhibits the bioinformatics community needs, as it focuses on developing algorithms to mine large amounts of DNA, RNA and protein data [2]. The study exhibits the amount of data-sharing by different threads of the workload and the amount of accesses to the last-level cache to shared cache lines [2]. Additionally, the study shows a direct correlation between performance of shared caches and the amount of data sharing, investigating cache performance on workload with the number of threads [2].
2) One experiment was an application instruction profile. This experiment performed an dynamic instruction profile on each workload with four threads. Instructions were separated into categories, either memory or ALU instructions. Specifically, any instruction operating in memory was categorized as a memory instruction whereas instructions operating purely in a register file were categorized as a ALU instruction. This experiment determined that workloads consisted of about 43-65% memory instructions [2].
Another experiment was the workload L1/L2 cache behavior. The research gathered statistics on the L1 and L2 caches regarding accesses and misses for each workload. This experiment found that 80-95% of L1 cache misses also missed in the L2 cache. Furthermore, this implies that the workloads contain two data sets: a frequently used small set and a large set that doesn't fit in the L2 data cache [2].
3) I learned that these workloads were based on the needs of the bioinformatics community - focusing in workloads that represent complex algorithms that mine data. Memory accesses dominated the workload instructions (43-65% of instructions) [2]. Additionally, the data sets were either frequently used small sets or large sets that extended passed the L1 and L2 cache [2]. The workloads varied in amount of data-sharing, some exhibiting high levels of data-sharing or very little such as SNP [2]. Furthermore, GeneNet, SEMPHY and SBM share data among two to four threads but PLSA only shares data among two or three threads [2].
4) The experiment found that data-sharing by workloads is a function of available cache size. This increases the importance of sharing a last-level cache. This cache improved performance most as a 32MB cache. This helped workloads with high level of shared-data by giving a shared cache to store and access data, lowering amount of costly jumps to main memory [2].
5) OpenMP is a programming interface that allows multi-platform shared memory programming. OpenMP can be utilized to create parallel programming workloads. Programming on OpenMP shares memory and data, and allows the developer to execute sections in parallel by creating threads for specified sections.

5 PART E

5.1 Trends: Memory Stream versus Optimization

Description: In this part you will use the allcache.so Pin tool to study the relationship between a program, compiler optimizations, and memory behavior. Select any C program of your choice. You will need source code that you can compile and run on the COE system and can compile with gcc. You will find the allcache.so Pin tool in the /COEnet/Linux/pin-3.11/source/tools/Memory/objintel64 directory. Copy this tool to a directory that you have read/write access to. Then run the tool. 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.

5.2 Solution

For part E, I chose to study the relationship between compiler optimization and memory behavior on the Dhrystone Benchmark C program. I tested Dhrystone with three different optimization flags: -O0, -O1, -O2, and -O3. The first optimization (-O0) is the default optimization for GCC. This flag optimizes compilation time of a program, effectively decreasing compile time and memory usage but increasing code size and execution time. The second optimization (-O1) optimizes code size and execution time but increases compile time and memory usage. The third optimization (-O2) further improves execution time but also increases compile time. The fourth optimization (-O3) decreases execution time from -O2 but also increases compile time from -O2.

Figure 11 illustrates the differences in L1 hits/misses among four different optimizations. General trends among all optimizations are that the L1 instruction cache has a higher number of hits and the L1 data cache has a higher number of misses. Additionally, the number of hits in both caches decreases dramatically between the -O0 and the -O1 optimization, but only decreases slightly between the -O1, -O2 and -O3 optimizations. On the opposite side, the L1 instruction data cache miss number remains fairly stable between the -O0, -O1, and -O2 optimizations but increases dramatically between the -O2 and -O3 optimization. The L1 instruction and data cache hit number decreases but the number of misses remains stable between the -O0 to -O2 optimization flags. This shows that the program accesses the L1 cache less often for those specific optimizations. Optimization flags 1 and 2 improve program execution time, this includes rearranging program instructions to decrease the necessary number of accesses. While these flags do not improve the number of misses, they do require fewer memory accesses.

Figure 12 illustrates the differences in L2 and L3 hits/misses among the four different optimizations. While the higher optimization flags decrease the number of necessary memory accesses in the L1 cache, they also increase number of hits and decrease number of misses in the L2 cache. The number of hits remained stable but the number of misses decreased between -O0 and -O2. This further showed the decrease in number of accesses to the L2 cache (but an increase in accuracy in the L2 cache). However, there is a large increase in number of hits between -O2 and -O3. Between the -O2 and -O3 optimization flags, nine specific optimizations are turned on by gcc: -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-loop-vectorize, -ftree-slp-vectorize, -fvect-cost-model, -ftree-partial-pre and -fipa-cp-clone. Of this list, the two most important options that affect the memory accesses are -funswitch-loops and -fpredictive-commoning [1]. The -funswitch-loops moves loop invariant conditions out of loops, and -fpredictive-commoning reuses computations performed in previous loop iterations (especially memory loads and stores) [1]. This allows the compiler to rearrange and reuse instructions, particularly memory access instructions. Furthermore, this decreases the amount of cache misses. By rearranging instructions, the compiler avoids capacity and conflict misses. It avoids these misses by decreasing the number of

memory accesses (less information is re-writing to the cache). Finally, Level 3 cache has very little changes between the optimizations because most optimization affects are seen in L1 and L2 caches.

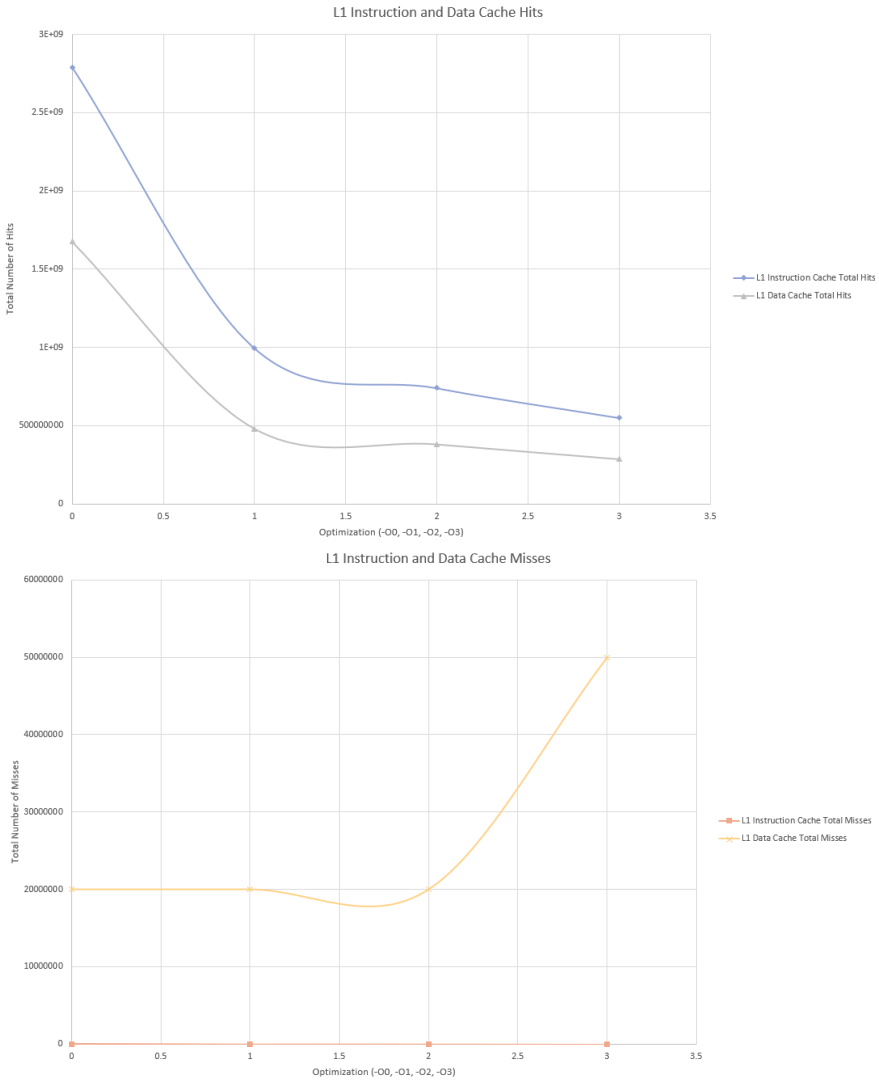

Figure 11: L1 Instruction versus data cache hits [A], L1 Instruction versus data cache misses [B].

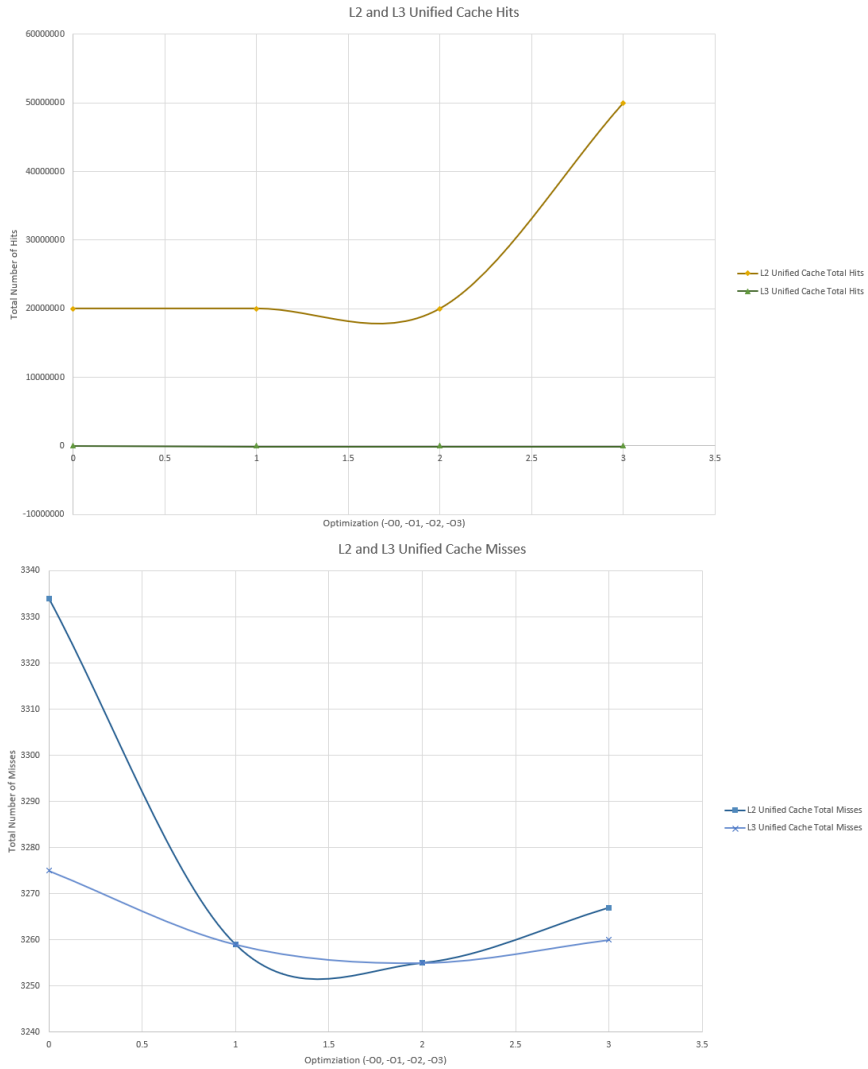

Figure 12: L2 versus L3 unified cache hits [A], L2 versus V3 unified cache misses [B].

6 EXTRA CREDIT 1

6.1 Cache Simulation

Description: Write your own simulator (you can modify the dineroIV source as a starting point) to implement a column associative cache, as described in the attached paper. Provide a working model that compiles and runs on the COE Linux system and report on the miss rate obtained when modeling a 8KB instruction cache with a 32B block size, that uses a column associative structure. Submit a copy of your code on Turnitin.

6.2 Solution

First, I implemented a direct-mapped cache as a proof of concept to compare results against DineroIV. The model simulates a 8KB instruction cache with a 32B block size and 32-bit addresses. Utilizing the provided trace file, my model produced a 7.2% miss rate which had a 1% difference compared to DineroIV, see Figure 13. This difference was small and could be contributed to rounding errors in the programs.

```
Results:
-----------------------------------------------------------
Instruction Cache
Description: 8192B cache size, 32B block size, 32-bit addresses, direct mapped

Cache Hits:     518793 (92.78 %)
Cache Misses:    40366 (7.22 %)
Total Fetches:    559159

bash-4.2$
```

```
---Simulation begins.
---Simulation complete.
l1-icache
 Metrics                 Total         Instrn         Data          Read          Write         Misc
 ----------------        ------        ------        ------        ------        ------        ------
 Demand Fetches          559159        559159             0             0             0             0
  Fraction of total      1.0000        1.0000        0.0000        0.0000        0.0000        0.0000

 Demand Misses            34685         34685             0             0             0             0
  Demand miss rate       0.0620        0.0620        0.0000        0.0000        0.0000        0.0000

 Multi-block refs             0
 Bytes From Memory      1109920
 ( / Demand Fetches)     1.9850
 Bytes To Memory              0
 ( / Demand Writes)      0.0000
 Total Bytes r/w Mem    1109920
 ( / Demand Fetches)     1.9850

---Execution complete.
```

Figure 13: Results of my model with direct-mapped configuration [A], DineroIV results of a direct-mapped cache [B].

After I verified this proof of concept, I improved my direct-mapped cache to be a column associative cache. The results are in Figure 14. A column associative cache is built from a direct-mapped cache. During cache hits, the cache reacts the same, finding the cache block by index and comparing tag values. However, during a cache miss, a column associative cache will perform a second hashing function to check elsewhere for the value. If this is also a miss, the data will be brought to the secondary hash function location. If this data receives another cache hit, it will be brought to the primary direct-mapped cache block location. For my program, I simulated a column associative cache. The specific hashing function I utilized was bit switching. If the direct-mapped cache block resulted in a miss, the index's binary was flipped (i.e. 110 => 001). This hashing type improved the cache by approximately 1.1%.

```
bash-4.2$ gcc -std=c99 cache.c -o cache && ./cache trace

Welcome to La Cache - a Column Associative Cache Simulation
by Anna DeVries
5 November 2019


Results:
-----------------------------------------------------
Instruction Cache
Description: 8192B cache size, 32B block size, 32-bit addresses, column associative

Cache Hits:     525130 (93.91 %)
Cache Misses:    34029 (6.09 %)
Total Fetches:    559159

bash-4.2$
```

Figure 14: Column-associative cache results.

```c
/*      Cache Simulation
        EECE 7352 - Computer Architecture

        by Anna DeVries
        5 November 2019

        Usage:
        ./cache <trace file>

        Trace File Format:
        LABEL   = 0       read data
                = 1       write data
                = 2       instruction fetch
                = 3       escape record (treated as unknown access type)
                = 4       escape record (causes cache flush)
        0 <= ADDRESS <= ffffffff where the hexadecimal addresses are NOT preceded by "0x"

        Example Trace File Formats:
        2 0     This is an instruction fetch at hex address 0.
        0 1000   This is a data read at hex address 1000.
        1 70f60888  This is a data write at hex address 70f60888.

        Description:
        8KB instruction cache
        32B block size
        32-bit address
        Column associative structure
        Reports miss rate                */

/*      Libraries to include             */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <assert.h>

/*      Global defines
  ---------------------------------------------------
  | Tag: 19 bits | Index: 8 bits | Byte Offset: 5 bits |
  ---------------------------------------------------
```

```
  31           13 12           5 4                   0
*/
#define SIZE 8192                       /* cache size in bytes */
#define BSIZE 32                        /* block size in bytes */
#define ASIZE 32                        /* address size in bits */
#define OFFSET 5                        /* log10(BSIZE)/log10(2) */
#define INDEX 8                         /* log10((SIZE/BSIZE))/log10(2) - 0 */
#define TAG 19                          /* BSIZE - calc2 - calc1 */


/*      Typedefs                        */
typedef struct Block_* Block;
typedef struct Cache_* Cache;
typedef struct Instruction_* Instruction;

/*      Struct objects                  */
// Block object
struct Block_{
    int tag;
};

// Cache object
struct Cache_{
    int hits, misses, size, bsize, lines;
    Block* blocks;
};

// Instruction object
struct Instruction_{
    int command;
    int addr;
    int tag, index, offset;
};

/*      Utility functions               */
Instruction create_instr(){
    // Variables
    Instruction instruction;

    // Allocate memory for instructions
    instruction = (Instruction) malloc(sizeof(struct Instruction_));
    if(instruction == NULL){
        printf("No memory allocated for cache.\n");

        return 0;
    }

    // Initalize instruction variables
    (*instruction).command = 0;
    (*instruction).addr = 0;
    (*instruction).tag = 0;
    (*instruction).offset = 0;
    (*instruction).index = 0;

    return instruction;
}

char *hex_to_binary(int address){
    // Variables
    Instruction instruction;
    char hex_buffer[9];
    char *bin_buffer = (char *)malloc(sizeof(char)*ASIZE);

    // Convert int address into a string
    sprintf(hex_buffer, "%x", address);
```

```
106
107     strncpy(bin_buffer,"", sizeof(bin_buffer) - 1);
108
109     // Convert string hex address to string binary
110     for(int i = 0; i < sizeof(hex_buffer); i++){
111         if(hex_buffer[i] == '0'){
112             strcat(bin_buffer, "0000");
113         }
114         else if(hex_buffer[i] == '1'){
115             strcat(bin_buffer, "0001");
116         }
117         else if(hex_buffer[i] == '2'){
118             strcat(bin_buffer, "0010");
119         }
120         else if(hex_buffer[i] == '3'){
121             strcat(bin_buffer, "0011");
122         }
123         else if(hex_buffer[i] == '4'){
124             strcat(bin_buffer, "0100");
125         }
126         else if(hex_buffer[i] == '5'){
127             strcat(bin_buffer, "0101");
128         }
129         else if(hex_buffer[i] == '6'){
130             strcat(bin_buffer, "0110");
131         }
132         else if(hex_buffer[i] == '7'){
133             strcat(bin_buffer, "0111");
134         }
135         else if(hex_buffer[i] == '8'){
136             strcat(bin_buffer, "1000");
137         }
138         else if(hex_buffer[i] == '9'){
139             strcat(bin_buffer, "1001");
140         }
141         else if(hex_buffer[i] == 'a'){
142             strcat(bin_buffer, "1010");
143         }
144         else if(hex_buffer[i] == 'b'){
145             strcat(bin_buffer, "1011");
146         }
147         else if(hex_buffer[i] == 'c'){
148             strcat(bin_buffer, "1100");
149         }
150         else if(hex_buffer[i] == 'd'){
151             strcat(bin_buffer, "1101");
152         }
153         else if(hex_buffer[i] == 'e'){
154             strcat(bin_buffer, "1110");
155         }
156         else if(hex_buffer[i] == 'f'){
157             strcat(bin_buffer, "1111");
158         }
159     }
160
161     return bin_buffer;
162 }
163
164 int binary_to_int(char *bin_buffer, int size){
165     // Variables
166     int sum = 0;
167     int base = 1;
168
169     // Convert binary to integer
170     for(int i = size - 1; i >= 0; i--){
```

```
171         if (bin_buffer[i] == '1')
172             sum += base;
173         base = base * 2;
174     }
175 
176     return sum;
177 }
178 
179 /*      Cache functions                 */
180 Cache create(){
181     /* Variables */
182     Cache cache;
183 
184     /* Allocate memory for cache */
185     cache = (Cache) malloc(sizeof(struct Cache_));
186     if(cache == NULL){
187         printf("No memory allocated for cache.\n");
188 
189         return 0;
190     }
191 
192     /* Initalize cache variables*/
193     (*cache).hits = 0;
194     (*cache).misses = 0;
195     (*cache).size = SIZE;
196     (*cache).bsize = BSIZE;
197     (*cache).lines = SIZE / BSIZE;
198     (*cache).blocks = (Block*) malloc(sizeof(Block) * (*cache).lines);
199     assert((*cache).blocks != NULL);
200 
201     /* Initalize blocks */
202     for(int i = 0; i < (*cache).lines; i++ ){
203         (*cache).blocks[i] = (Block) malloc(sizeof(struct Block_ ));
204         assert((*cache).blocks[i] != NULL);
205         (*(*cache).blocks[i]).tag = 0;
206     }
207 
208     return cache;
209 }
210 
211 Cache destroy(Cache cache){
212     free((*cache).blocks);
213     free(cache);
214     cache = NULL;
215 
216     return cache;
217 }
218 
219 int play_with_cache(Cache cache, int offset, int index, int tag){
220     // Variables
221     Block block;
222     Block new_block;
223     int c, k;
224     char new_index[INDEX];
225 
226     // Get block
227     block = (*cache).blocks[index];
228 
229     // Compare block tag value with current tag at index
230     // Hit
231     if((*block).tag == tag){
232         (*cache).hits++;
233 
234         return 1;
235     }
```

```
    // Miss
    else{
        // Block is empty, add value
        if((*block).tag == 0){
            (*cache).misses++;
            (*block).tag = tag;

            return 1;
        }

        // Block is full, try another hashing function
        // Hashing function = index bit-flipping
        // Convert index back to binary, and flip bits
        strncpy(new_index, "", sizeof(new_index) - 1);

        int i = 0;
        for(c = INDEX - 1; c >= 0; c--){
            k = index >> c;

            if (k & 1){                 /* Replace 1 with 0 for flipped bits*/
                strcat(new_index, "0");
            }
            else{                       /* Replace 0 with 1 for flipped bits*/
                strcat(new_index, "1");
            }
            i++;
        }

        // Convert new_index binary string into integer
        int n_index = binary_to_int(new_index, INDEX);


        // Get new block
        new_block = (*cache).blocks[n_index];

        // Compare block tag value with current tag at index
        // Hit
        if((*new_block).tag == tag){
            (*cache).hits++;

            // Switch block tags
            (*new_block).tag = (*block).tag;
            (*block).tag = tag;

            return 1;
        }
        else{
            (*cache).misses++;
            (*new_block).tag = tag;
        }

    }

}

/*      Main function                   */
int main(int argc, char **argv){
    // Intro
    printf("\nWelcome to La Cache - a Column Associative Cache Simulation\n");
    printf("by Anna DeVries\n");
    printf("5 November 2019\n\n");


    // Variables
    FILE *fp;
```

```
301     Cache cache;
302     Instruction instruction;
303     char line[100];
304     char *bin_buffer;
305 
306     // Check arguments
307     if(argc < 2){
308         printf("Usage: ./cache <trace file>\n");
309         printf("For help: ./cache --help\n");
310 
311         return 0;
312     }
313 
314     // Check if help is requested
315     if(strcmp(argv[1], "--help") == 0){
316         printf("\nUsage: ./cache <trace file>\n\n");
317         printf("Trace File Format:\nLABEL   = 0       read data\n= 1       write data\n= 2
        instruction fetch\n= 3       escape record (treated as unknown access type)\n= 4       escape
         record (causes cache flush)\no <= ADDRESS <= ffffffff where the hexadecimal addresses are
        NOT preceded by 0x.\n\nExample Trace File Formats:\n2 0    This is an instruction fetch at
        hex address 0.\no 1000   This is a data read at hex address 1000.\n1 70f60888  This is a data
         write at hex address 70f60888.\n");
318         printf("\nCache Simulation Description:\n");
319         printf("%dB cache size, %dB block size, %d-bit addresses, column associative\n\n", SIZE,
        BSIZE, ASIZE);
320         printf("\nCache simulation created by Anna DeVries\n");
321         return 0;
322     }
323 
324     // Open trace file
325     fp = fopen(argv[1], "r");
326 
327     // Check trace file
328     if(fp == NULL){
329         printf("Error opening file.\n");
330 
331         return -1;
332     }
333 
334     // Initialize cache
335     cache = create();
336 
337     // Initialize instruction
338     instruction = create_instr();
339 
340     // Begin instruction flow
341     // Grab instructions from each line
342     while(fgets(line, sizeof(line), fp) != NULL){
343         // Parse values into instruction object
344         sscanf(line, "%d %x", &((*instruction).command), &((*instruction).addr));
345 
346         // Convert address to binary
347         bin_buffer = hex_to_binary((*instruction).addr);
348 
349         // Format memory address
350         // Format offset
351         char *partial_offset = (char *)malloc(sizeof(char)*OFFSET);
352         strncpy(partial_offset,"", sizeof(partial_offset) - 1);
353         for(int i = OFFSET; i > 0; i--){
354             partial_offset[OFFSET - i] = bin_buffer[TAG + INDEX + OFFSET - i];
355         }
356         // Convert binary to decimal int
357         (*instruction).offset = binary_to_int(partial_offset, OFFSET);
358 
359         // Format index
```

```
360         char *partial_index = (char *)malloc(sizeof(char)*INDEX);
361         strncpy(partial_index,"", sizeof(partial_index) - 1);
362         for(int i = INDEX; i > 0; i--){
363             partial_index[INDEX - i] = bin_buffer[TAG + INDEX - i];
364         }
365         // Convert binary to decimal int
366         (*instruction).index = binary_to_int(partial_index, INDEX);
367 
368         // Format tag
369         char *partial_tag = (char *)malloc(sizeof(char)*TAG);
370         strncpy(partial_tag,"", sizeof(partial_tag) - 1);
371         for(int i = TAG; i > 0; i--){
372             partial_tag[TAG - i] = bin_buffer[TAG - i];
373         }
374         // Convert binary to decimal int
375         (*instruction).tag = binary_to_int(partial_tag, TAG);
376 
377         // Check if command reads (0), writes (1) or fetches (2) data. Only concerned about
    instruction fetches.
378         if((*instruction).command == 2){
379             play_with_cache(cache, (*instruction).offset, (*instruction).index, (*instruction).
    tag);
380 
381         }
382     }
383 
384     // Print results
385     int total = (*cache).hits + (*cache).misses;
386     float hit_rate = ((float)(*cache).hits / (float)total) * 100;
387     float miss_rate = ((float)(*cache).misses / (float)total) * 100;
388 
389     printf("\nResults: \n");
390     printf("-----------------------------------------------------------------\n");
391     printf("Instruction Cache\n");
392     printf("Description: %dB cache size, %dB block size, %d-bit addresses, column associative\n\n
    ", SIZE, BSIZE, ASIZE);
393     printf("Cache Hits:     %d (%.2f %%)\n", (*cache).hits, hit_rate);
394     printf("Cache Misses:    %d (%.2f %%)\n", (*cache).misses, miss_rate);
395     printf("Total Fetches:    %d\n\n", total);
396 
397 
398     /* Gracefully terminate */
399     fclose(fp);
400     destroy(cache);
401 
402     return 1;
403 
404 }
```

Algorithm 8: Extra Credit 1 Cache Simulation

7 EXTRA CREDIT 2

7.1 Pin Simulation

Description: Using Pin, develop a data cache model of a 8KB data cache that has a 32 byte line and is two-way set associative, and that uses as input an data address trace of Dhrystone. Change the number of iterations (i.e., the LOOPS constant) and compile with different levels of optimization. Discuss how these changes impact the cache hit rate.

7.2 Solution

For this simulation, I utilized the dcache.so library from the pin tool. I ran the following code with different loop counts and optimizations on Dhrystone: pin -t dcache.so -c 8 -a 2 -b 32 – ./dry. The results for each optimization are in Tables 6, 7, 8, and 9.
First, the hit rate improved with increased loop counts. Dhrystone utilizes loops of mathematical equations to test computer performance. One downfall of Dhrystone, however, is that it may fit into a computer's cache - thus producing better results then the real-world workload may produce. This is illustrated in the loop counts. Higher loop counts produced more cache hits because the values were present in the cache prior and were not removed from the cache due to size constraints.
Second, while loop count 100000 produced more accurate results in the -O0 optimization, all other loop counts improved with greater optimizations. These optimizations improve hit rate by moving instructions around to reduce number of memory accesses and to de-conflict cache conflicts. Specifically, the -O3 optimization flag optimizes execution time and includes the -funswitch-loops and -fpredictive-commoning flags. These two flags move loop invariant conditions out of loops and reuse computations from previous loops. Both of these work together to decrease the number of memory accesses in the program. Fewer requests to cache result in fewer cache line overwrites and prevents misses.

Table 6: -O0 optimization results.

Loop Count	Total Hits (Hit Rate %)	Total Misses
100000	33945543 (99.98%)	5904
500000	168796164 (99.55%)	756050
1000000	339046017 (100%)	6202
5000000	1680046295 (99.11%)	15005927
10000000	3390046037 (100%)	6184
50000000	16950046247 (100%)	5979

Table 7: -O1 optimization results.

Loop Count	Total Hits (Hit Rate %)	Total Misses
100000	10044798 (99.93%)	6635
500000	50046252 (99.99%)	5949
1000000	100045088 (99.99%)	6350
5000000	500046305 (100%)	5906
10000000	1000045964 (100%)	6246
50000000	5000046316 (100%)	5897

Table 8: -O2 optimization results.

Loop Count	Total Hits (Hit Rate %)	Total Misses
100000	8045268 (99.92%)	6163
500000	40045574 (99.99%)	5857
1000000	80045485 (99.99%)	5951
5000000	400045817 (100%)	6388
10000000	800046228 (100%)	5976
50000000	4000045789 (100%)	6418

Table 9: -O3 optimization results.

Loop Count	Total Hits (Hit Rate %)	Total Misses
100000	6745594 (99.91%)	5838
500000	33545550 (99.98%)	5882
1000000	67045991 (99.99%)	6215
5000000	335045994 (100%)	6212
10000000	670046266 (100%)	5939
50000000	3350045809 (100%)	6399

REFERENCES

[1] Options that control optimization. URL https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Optimize-Options.html. [Online; accessed 3-November-2019].

[2] A. Jaleel, M. Mattina, and B. Jacob. *Last Level Cache (LLC) Performance of Data Mining Workloads On a CMP - A Case Study of Parallel Bioinformatics Workloads*.