Homework 7 Cache Report

Justin Birdsall & Richard Rory
Professor Lalejina
CIS 351 Computer Orginization
11 December 2023

```c
/* i.e., 64KB.  The array needs to be at least twice the size of the
   8KB cache so that the array doesn't fit in memory. */
#define ARRAY_SIZE 64*1024

#define NUM_LOOPS 10000000

int main() {
  _Alignas(64)  /* make sure that the array aligns with the cache. */
  char array[ARRAY_SIZE];
  register int outer_loop;
  register int inner_loop;
  register int solution = 0;

  for (outer_loop = 0; outer_loop < NUM_LOOPS; outer_loop++) {

      //This will go into set 1 32-63 for the 32 blocks and set 0 (0-63)
     //for 64 bit block cache
      solution *= array[32];
    //8192 is the first digit that wraps back around to set 0 for both
    //since set 0 is taken already in the 64 bit block it
    //will boot and replace it.Next iteration of the loop will then boot
    //out this value for 32
    //For 32 bit blocks 0 is free so once it gets
    //stored after first miss
    //we won't have to boot it out again
      solution *= array[8192];

  }

  return solution;
}
```

Command:

Parameters: 1 way 8192 bit cache size with 32 and 64 bit block sizes

“for each 32 and 64 bit size block in a 8192 bit cache that is 1 way run valgrind cachegrind”

for block in 32 64;   do valgrind --tool=cachegrind --cachegrind-out-file=/dev/null --D1=8192,1,${block} ./a.out 2> output_${block}; done

Output Hit rates:

32 bit block: output_32:==380070== D1 miss rate: 0.0% (0.0%   +   12.1%)

64 bit block: output_64:==380071== D1 miss rate: 99.8% (99.9%   +   7.6%)

Problem 3:

Code:

```
#define ARRAY_SIZE 64*1024

#define NUM_LOOPS 10000000

int main() {
  _Alignas(64)  /* make sure that the array aligns with the cache. */
  char array[ARRAY_SIZE];
  register int outer_loop;
  register int inner_loop;
  register int solution = 0;

  for (outer_loop = 0; outer_loop < NUM_LOOPS; outer_loop++) {
      //2way will always miss since you constantly having to kick least
     //recently used out. 4 way will miss on the initial loading in
     //but afterwards data is loaded in it won't have to worry about
    //freeing up a block in the set
      solution *= array[1024];
      solution *= array[768];
      solution *= array[512];
      solution *= array[256];

  }
  return solution;
}
```

Commands:

Parameters: (2 or 4 way) 256 bit cache size with 32 bit block sizes

“for each 32 bit size block in a 256 bit cache that is 2 way run valgrind cachegrind”

for block in 32;   do valgrind --tool=cachegrind --D1=256,2,${block} ./a.out 2> output_${block}; done

“for each 32 bit size block in a 256 bit cache that is 4 way run valgrind cachegrind”

for block in 32;   do valgrind --tool=cachegrind --D1=256,4,${block} ./a.out 2> output_${block}; done

Output Hit rates:

2 way: output_32:==326736== D1 miss rate: **99.9%** (100.0%   +   36.7%)

4 way: output_32:==327181== D1 miss rate: **0.0%** (0.0%   +   35.9%)

Problem 4:

Code:

```
#define ARRAY_SIZE 64*1024

#define NUM_LOOPS 10000000


int main() {
  _Alignas(64)  // make sure that the array aligns with the cache.
  char array[ARRAY_SIZE];
  register int outer_loop;
  register int inner_loop;
  register int solution = 0;

  for (outer_loop = 0; outer_loop < NUM_LOOPS; outer_loop++) {

      solution *= array[64];
      solution *= array[128];
      solution *= array[256];
  }
  return solution;
}
```

Commands:

Parameters: (1 or 2 way) 128 bit cache size with 32 bit block sizes

“for each 32 bit size block in a 128 bit cache that is 1 way run valgrind cachegrind”

```
for block in 32;   do valgrind --tool=cachegrind --D1=128,1,${block} ./a.out 2> output_${block}; done
```

“for each 32 bit size block in a 128 bit cache that is 2 way run valgrind cachegrind”

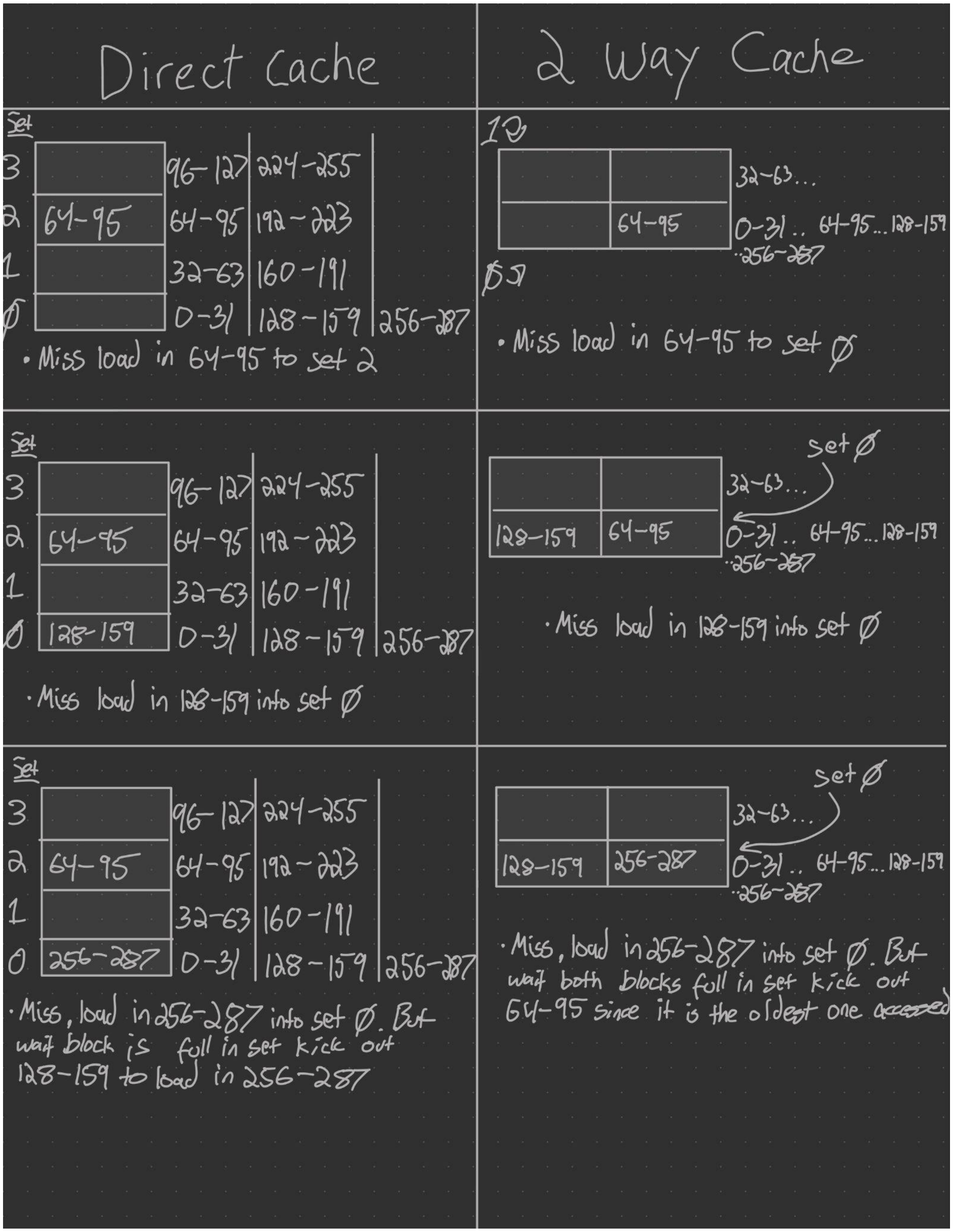
```
for block in 32;   do valgrind --tool=cachegrind --D1=128,2,${block} ./a.out 2> output_${block}; done
```

Output Hit rates:

1 way: output_32:==330536== D1 miss rate: 66.6% (66.7%   +   48.0%)

2 way: output_32:==330678== D1 miss rate: 99.9% (99.9%   +   41.0%)

Problem 5 (Problem 4 Explanation):



• Miss load in 64-95 to set 2

• Miss load in 64-95 to set Ø

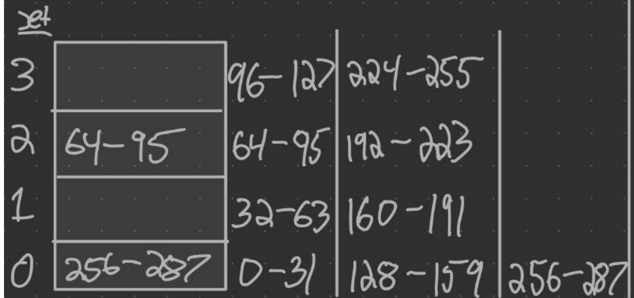
• Miss load in 128-159 into set Ø

• Miss load in 128-159 into set Ø

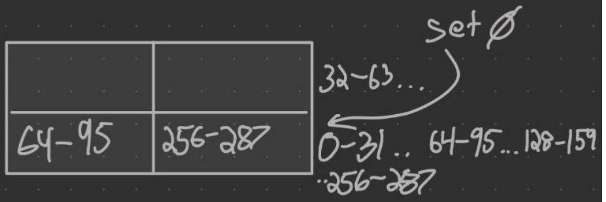
• Miss, load in 256-287 into set Ø. But wait block is full in set kick out 128-159 to load in 256-287

• Miss, load in 256-287 into set Ø. But wait both blocks full in set kick out 64-95 since it is the oldest one ~~accessed~~

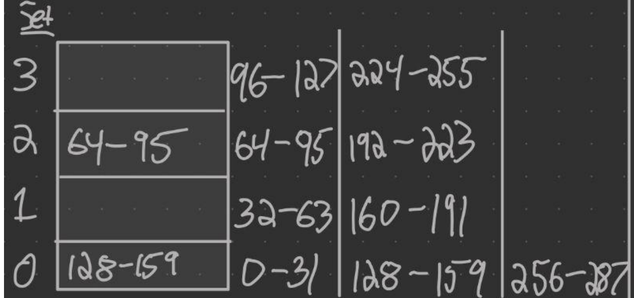
Loop 2



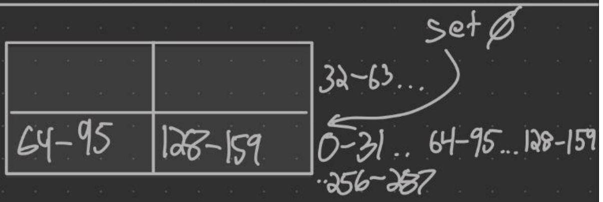
·Hit 64 is already loaded into Cache



·Miss, load in 64–95 into set Ø. But wait both blocks full in set kick out 128–159 since it is the oldest one accessed



Miss, kick out 256–287 so we can load in 128–159 into set Ø

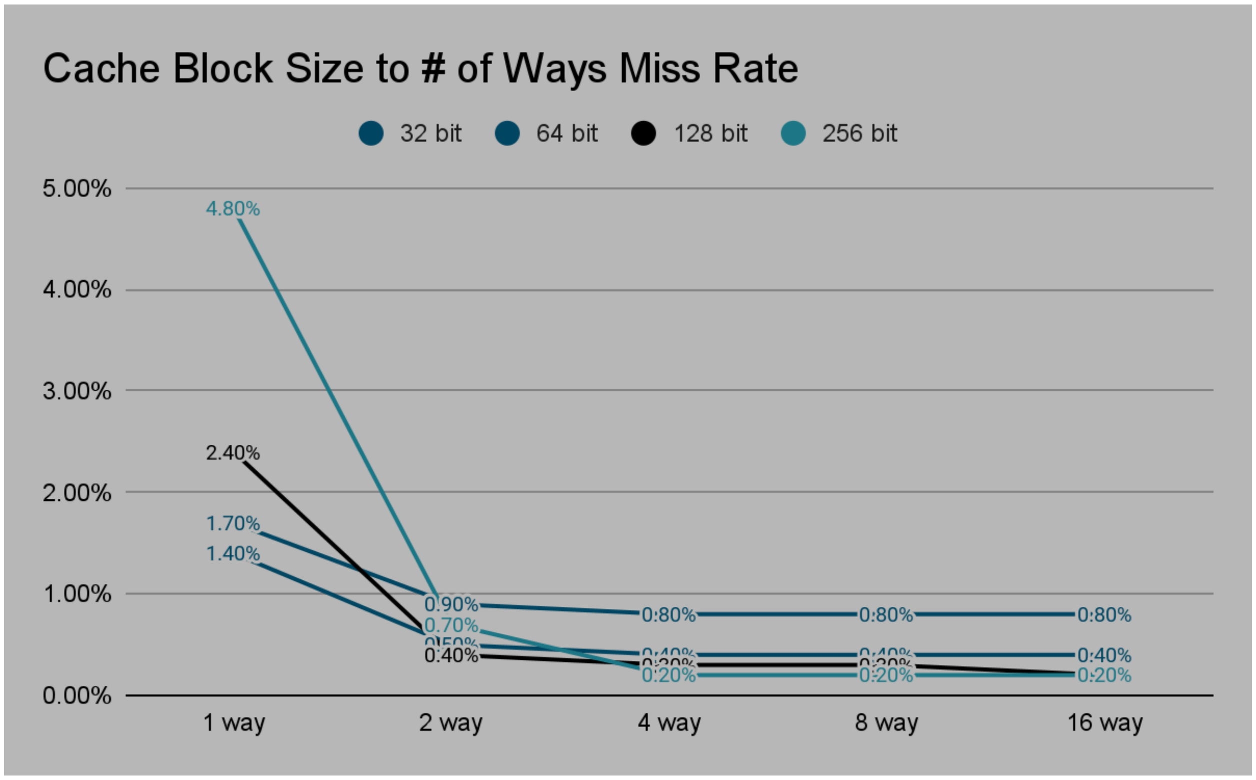


·Miss, load in 128–159 into set Ø. But wait both blocks full in set kick out 256–287 since it is the oldest one accessed

What we can see by going through the steps our two way cache will always be booting out a value even if set Ø fills due to a miss occurring on every step of our loop. On the other hand our direct cache will have a 66% miss rate. once we bring 64–95 into our set 3 of our cache. That value never gets booted out. However since 128–159 & 259–289 are both mapped to set Ø they will constantly be alternating holding that spot in cache. $\frac{1}{3}$ is a hit and $\frac{2}{3}$ = miss giving us that 66% miss rate which was proven to be correct in our simulation and 100% miss rate for 2 way

Problem 6:

	32 bit	64 bit	128 bit	256 bit
1 way	1.70%	1.40%	2.40%	4.80%
2 way	0.90%	0.50%	0.40%	0.70%
4 way	0.80%	0.40%	0.30%	0.20%
8 way	0.80%	0.40%	0.30%	0.20%
16 way	0.80%	0.40%	0.20%	0.20%


Cache Block Size to # of Ways Miss Rate
● 32 bit ● 64 bit ● 128 bit ● 256 bit
1 way
2 way
4 way
8 way
16 way