CS341 Cache HW Part B

Ryan Scherbarth

November 2024

1. **Problem 6.34** **Consider the following matrix transpose routine:** 15/15

```
typedef int array[4][4];

void transpose2(array dst, array src)
{
  int i,j;
  for (i = 0; i < 4; i++){
    for (j = 0; j < 4; j++){
      dst[j][i] = src[i][j];
    }
  }
}
```

Assume this code runs on a machine with the following properties;

- $sizeof(int) = 4$
- **the** *src* **array starts at address 0 and the** *dst* **array starts at address** 64 **(decimal).**
- **There is a single** $L1$ **data cache that is direct-mapped, write-through, write-allocate, with a block size of 16 bytes**
- **The cache has a total size of 32 data bytes, and the cache is initially empty**
- **Access to the** *src* **and** *dst* **arrays are the only sources of read and write misses, respectively.**

(a) **For each** *row* **and** *col***, indicate whether the access to** $src[row][col]$ **and** $dst[row][col]$ **is a hit** (h)**, or a miss** (m)**. For example, reading** $src[0][0]$ **is a miss and writing** $dst[0][0]$ **is also a miss.**

	Col 0	Col 1	Col 2	Col 3
Row 0	m	**m**	**m**	**m**
Row 1	**m**	**m**	**m**	**m**
Row 2	**m**	**m**	**m**	**m**
Row 3	**m**	**m**	**m**	**m**

	Col 0	Col 1	Col 2	Col 3
Row 0	m	**m**	**m**	**m**
Row 1	**m**	**m**	**m**	**m**
Row 2	**m**	**m**	**m**	**m**
Row 3	**m**	**m**	**m**	**m**

Cache	m	C	B	E	S	t	s	b
R1	32	32	16	1	2	27	1	4

So our cache will be of format: $[T = 27][S = 1][b = 4]$.

Simulating reads:

Read	Array	Index	Address	Set Bit	Hit/Miss	S0 cover	S1 cover
1	src	[0][0]	0000 0000 $(0)_{10}$	0	Miss	0 - 16	
2	dst	[0][0]	0100 0000 $(64)_{10}$	0	Miss	64 - 80	
3	src	[0][1]	0000 0100 $(4)_{10}$	0	Miss	4 - 20	
4	dst	[1][0]	0101 0000 $(80)_{10}$	1	Miss	4 - 20	80 - 96
5	src	[0][2]	0000 1000 $(8)_{10}$	0	Hit	4 - 20	80 - 96
6	dst	[2][0]	0110 0000 $(96)_{10}$	0	Miss	96 - 112	80 - 96
7	src	[0][3]	0000 1100 $(12)_{10}$	0	Miss	12 - 28	80 - 96
8	dst	[3][0]	0111 0000 $(112)_{10}$	1	Miss	12 - 28	112 - 128
9	src	[1][0]	0001 0000 $(16)_{10}$	1	Miss	12 - 28	16 - 32
10	dst	[0][1]	0100 0100 $(68)_{10}$	0	Miss	68 - 84	16 - 32
11	src	[1][1]	0001 0100 $(20)_{10}$	1	Hit	68 - 84	16 - 32
12	dst	[1][1]	0101 0100 $(84)_{10}$	1	Miss	68 - 84	84 - 100
13	src	[1][2]	0001 1000 $(24)_{10}$	1	Miss	68 - 84	24 - 40
14	dst	[2][1]	0110 0100 $(100)_{10}$	0	Miss	100 - 116	24 - 40
15	src	[1][3]	0001 1100 $(28)_{10}$	1	Hit	100 - 116	28 - 44
16	dst	[3][1]	0111 0100 $(116)_{10}$	1	Miss	100 - 116	116 - 132
17	src	[2][0]	0010 0000 $(32)_{10}$	0	Miss	32 - 48	116 - 132
18	dst	[0][2]	0100 1000 $(72)_{10}$	0	Miss	72 - 88	116 - 132
19	src	[2][1]	0010 0100 $(36)_{10}$	0	Miss	36 - 52	116 - 132
20	dst	[1][2]	0101 1000 $(88)_{10}$	1	Miss	36 - 52	88 - 104
21	src	[2][2]	0010 1000 $(40)_{10}$	0	Hit	36 - 52	88 - 104
22	dst	[2][2]	0110 1000 $(104)_{10}$	0	Miss	104 - 120	88 - 104
23	src	[2][3]	0010 1100 $(44)_{10}$	0	Miss	44 - 60	88 - 104
24	dst	[3][2]	0111 1000 $(120)_{10}$	1	Miss	44 - 60	120 - 136
25	src	[3][0]	0011 0000 $(48)_{10}$	1	Miss	44 - 60	48 - 64
26	dst	[0][3]	0100 1100 $(76)_{10}$	0	Miss	76 - 92	48 - 64
27	src	[3][1]	0011 0100 $(52)_{10}$	1	Hit	76 - 92	48 - 64
28	dst	[1][3]	0101 1100 $(92)_{10}$	1	Miss	76 - 92	92 - 108
29	src	[3][2]	0011 1000 $(56)_{10}$	1	Miss	76 - 92	56 - 72
30	dst	[2][3]	0110 1100 $(108)_{10}$	0	Miss	108 - 124	56 - 72
31	src	[3][3]	0011 1100 $(60)_{10}$	1	Hit	108 - 124	56 - 72
32	dst	[3][3]	0111 1100 $(124)_{10}$	1	Miss	108 - 124	108 - 124

	Col 0	Col 1	Col 2	Col 3
Row 0	m	**m**	**m**	**m**
Row 1	**m**	**m**	**m**	**m**
Row 2	**m**	**m**	**m**	**m**
Row 3	**m**	**m**	**m**	**m**

	Col 0	Col 1	Col 2	Col 3
Row 0	m	**m**	**h**	**m**
Row 1	**m**	**h**	**m**	**h**
Row 2	**m**	**m**	**h**	**m**
Row 3	**m**	**h**	**m**	**h**

I need to add text here to fix the page formatting...

2. **Problem 6.35 Repeat Problem 6.34 for a cache with a total size of 128 bytes.** 16/16

The only thing that changes here is we now can cache 128 bits instead of the 64 from before. We're not able to load the entire array into memory for both locations. We will override our values each time we get to a new row, but will end up with better caching results regardless.

	Col 0	Col 1	Col 2	Col 3
Row 0	m	**h**	**h**	**h**
Row 1	**m**	**h**	**h**	**h**
Row 2	**m**	**h**	**h**	**h**
Row 3	**m**	**h**	**h**	**h**

	Col 0	Col 1	Col 2	Col 3
Row 0	m	**h**	**h**	**h**
Row 1	**m**	**h**	**h**	**h**
Row 2	**m**	**h**	**h**	**h**
Row 3	**m**	**h**	**h**	**h**

3. **Problem 6.36** **This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze:**

```
int x[2][128];
int i;
int sum=0;
for(int i=0; i<128; i++){
  sum += x[0][i] * x[1][i];
}
```

44/44

Assume we execute this under the following conditions:

- $sizeof(int) = 4$
- **Array x begins at memory address $0x0$ and is stored in row-major order.**
- **In each case below, the cache is initially empty**
- **The only memory accesses are to the entries of the array x. All other variables are stored in registers.**

Given these assumptions, estimate the miss rates for the following cases:

(a) **Case 1: Assume the cache is 512 bytes, direct-mapped, with 16 byte cache blocks. What is the miss rate?**

There are $\frac{512}{16} = 32$ sets, $\log_2(32) = 5$ set index bits, and $\log_2(16) = 4$ offset bits. The array contains $2 * 128 = 256$ integers of size 4, totaling 1024 bytes. The issue arises, however, when we're iterating through in this manner. At each step we call $x[1][i]$ immediately after calling $x[0][i]$. These both go back and forth overriding each other causing us to never reach a valid cache hit.

Therefore, our miss rate will be 100%

(b) **What is the miss rate if we double the cache size to 1,024 bytes?**

Doubling the cache size means $\frac{1024}{16} = 64$ sets, again with 4 offset bits, however this time we will have $\log_2(32) = 5$ set index bits. We now have a few cases where we get large enough numbers to jump into a different set. At each step we can store the full array in the cache, and we will only run into a miss every time we change sets to the next column, resulting in $\frac{1}{4}$ of the calls being missed.

Therefore, we will end up with a miss rate of 25%.

(c) **Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the cache miss rate?**

Adding Associativity 2-way allows $x[0][j]$ and $x[j][0]$ to both access the same cache set without evicting each other, since each can reside on one of the sets we have here; $\log_2(16) = 4$ set index bits.

The LRU policy causes misses on the first access to each unique block as usual, after which we have a large enough cache to store the entire row.

Therefore, in this situation we can also achieve 25% miss rate.

(d) **For the previous, will a larger cache size help to reduce the miss rate? why or why not?**

We already have 2 way associativity, so increasing the cache size will not help improve our miss rate at all on part c. The current cache size is not the limiting factor that is causing us to hit misses in this situation.

(e) **For case 3, will a larger block size help to reduce the miss rate? why or why not?**

Yes. Since items are stored in row major ordering, increasing block size would end up storing more of the values for extra columns where we were previously needing to take a miss and reload a new block into memory. Increasing block size could decrease our number of misses for this reason.

4. **Problem 6.38** **3M decides to make Post-its by printing yellow squares on white pieces of paper. As a part of the printing process, they need to set the CMYK (cyan, magenta, yellow, black) value for every point in the square. 3M hires you to determine the efficiency of the following algorithms on a machine with a 2,048 byte direct-mapped data cache with 32-byte blocks. You are given the following definitions:** 25/25

```
1 struct point_color {
2   int c;
3   int m;
4   int y;
5   int k;
6 };
7
8 struct point_color square[16][16];
9 int i,j;
```

Assume the following:

- $sizeof(int) = 4$
- **square begins at memory address 0.**
- **the cache is initially empty**
- **the only memory accesses are to the entries of the array** *square*. **Variables** i **and** j **are stored in registers.**

Determine the cache performance for the following code:

```
1 for (i=0; i<16; i++){
2   for (j=0; j<16; j++){
3     square[i][j].c = 0;
4     square[i][j].m = 0;
5     square[i][j].y = 1;
6     square[i][j].k = 0;
7   }
8 }
```

(a) **What is the total number of writes?**

We don't need to take into account caching at all as far as write calculations are concerned as there is no way to optimize / cache a write operation. We will have $16 * 16 * 4 = 1024$ write operations.

(b) **What is the total number of writes that miss in the cache?**

Our cache size is 2048, block size is 32, and an int size of 4B. The total size of the array struct will be 4096 bytes. We are accessing our structure in a wise way - iterating through to follow along with the row major ordering that C implements by default, which helps our cache performance. We can find two of the point_color structs into a single cache block at a time. We will only end up hitting a miss every 2 writes because of this

We will end up with a total of $\frac{256}{2} = 128$ cache misses.

(c) **What is the miss rate?** We will do a total of 1024 writes, after missing 128 of them we will have a miss rate of $\frac{128}{1024} = 12.5\%$.