

Proof of Study 9

Qianlang Chen

UI172983

For this analysis, I chose to analyze the three types of caches with the following configuration:

Fully-Associative: 8 entries 8 bytes per block. Since total bit count = (valid bit + tag size + LRU size + data block size) * entry count, this cache has $(1 + (16 - \lg(8)) + \lg(8) + 8 * 8) * 8 = 648$ bits. 192 of the allocated 840 bits remain unused. Moreover, the penalty of each DRAM access for this cache is $10 + 8 = 18$ cycles. Using this cache, each 16-bit address should be broken down as follows:

13	3
Tag	Offset

Direct Mapped: 8 rows and 8 bytes per block. Since this cache does not need LRUs, it has a total of $(1 + (16 - \lg(8) - \lg(8)) + 0 + 8 * 8) * 8 = 600$ bits, where 240 of the 840 bits remain unused. The penalty for accessing DRAM is also 18 cycles. Using this cache, each address is broken down as follows:

10	3	3
Tag	Row	Offset

Set-Associative: 4 rows, 2 ways, and 8 bytes per block. It uses a total of $(1 + (16 - \lg(4) - \lg(8)) + \lg(2) + 8 * 8) * (4 * 2) = 616$ bits, where 224 of the 840 bits remain unused. The penalty for accessing DRAM is also 18 cycles. Using this cache, each address is broken down like so:

11	2	3
Tag	Row	Offset

To analyze these caches and determine which of them works best, I wrote a C++ program to simulate each of them. Here it is (in the next few pages):

```

u1172983@lab1-8:cache_simulator
File Edit Options Buffers Tools C++ Help
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    int addr_size = 16;
    int block = 8; // block size (maximum offset)
    int rows = 8;
    int ways = 1; // num of mini fully assoc maps (in a set assoc map)

    int cpi = 1;
    int const_pen = 10; // constant extra penalty for misses

    int n = 32; // addr count
    int addrs[] = {
        4, 8, 12, 16, 20, 32, 36, 40, 44, 20, 32, 36, 40, 44, 64, 68,
        4, 8, 12, 92, 96, 100, 104, 108, 112, 100, 112, 116, 120, 128, 140, 144
    };

    cout << "-----" << endl
         << "address size: " << addr_size << endl
         << "block size: " << block << endl
         << "rows: " << rows << endl
         << "ways: " << ways << endl
         << "normal cpi: " << cpi << endl
         << "extra miss penalty: " << const_pen << endl
         << "addresses: ";
    for (int i = 0; i < n; i++) {
        cout << addrs[i] << " ";
    }
    cout << endl;

    int cache[rows][ways];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < ways; j++) {
            cache[i][j] = -1;
        }
    }
}

```

(continued...)

```
int pen = const_pen + cpi * block;

cout << "-----" << endl
     << "addr\ttag\trow" << endl;

int hits = 0, cycles = n;
for (int t = 0; t < 2; t++) {
    // only analyze the second run (when t == 1)
    for (int i = 0; i < n; i++) {
        int addr = addrs[i];
        int offset = addr % block;
        int row = addr / block % rows;
        int tag = addr / block / rows;

        bool hit = false;
        for (int j = 0; j < ways; j++) {
            if (cache[row][j] == tag) {
                hit = true;
                // put most recently used at the front and shift all
                int temp = cache[row][j];
                for (int k = j; k > 0; k--) {
                    cache[row][k] = cache[row][k - 1];
                }
                cache[row][0] = temp;
                break;
            }
        }
        if (hit) {
            if (t == 1) {
                cout << addr << "\t" << tag << "\t" << row << "\thit" << endl;

                hits++;
            }
            continue;
        }

        // miss
        if (t == 1) {
            cout << addr << "\t" << tag << "\t" << row << "\tmiss" << endl;

            cycles += pen;
        }

        // put loaded data at the front and shift all
        for (int k = ways - 1; k > 0; k--) {
            cache[row][k] = cache[row][k - 1];
        }
        cache[row][0] = tag;
    }
}
```

(continued...)

```
cout << "-----" << endl
    << "contents after simulation:" << endl
    << "row\t(v) tags in ways..." << endl;

for (int i = 0; i < rows; i++) {
    cout << i << "\t";
    for (int j = 0; j < ways; j++) {
        if (cache[i][j] < 0) {
            cout << "(0)" << "\t";
        } else {
            cout << "(1) " << cache[i][j] << "\t";
        }
    }
    cout << endl;
}

cout << "-----" << endl
    << "hits: " << hits << endl
    << "misses: " << (n - hits) << endl
    << "hit rate: " << (double)hits / n << endl
    << "penalty for each miss: " << pen << endl
    << "total miss penalty: " << cycles - n << endl
    << "total cycles: " << cycles << endl
    << "cpi: " << (double)cycles / n << endl
    << "lru bits used: " << (int)(log(ways) / log(2)) << endl
    << "total bits used: " << (int)(rows * ways
        * (1 + addr_size - log(rows * block) / log(2)
        + block * 8 + ceil(log(ways) / log(2))))
    << endl
    << "-----" << endl;

return 0;
}
```

```
-UU-:----F1 simulator.cpp All L1 (C++/l Abbrev) -----
loading cc-flags done
```

This simulator simulates and outputs how each address is treated (hit or miss), the final state of the cache, and the total cycle count. Note that the total cycle count = the number of addresses + miss penalty * miss count. Here are the simulation results:

Fully-Associative:

```

u1172983@lab1-8:cache_simulator
-----
address size: 16
block size: 8
rows: 1
ways: 8
normal cpi: 1
extra miss penalty: 10
addresses: 4 8 12 16 20 32 36 40 44 20 32 36 40 44 64 68 4 8 12 92 96 100 104 108 112 1
00 112 116 120 128 140 144
-----
addr    tag    row    result
4        0        0    miss
8        1        0    miss
12       1        0    hit
16       2        0    miss
20       2        0    hit
32       4        0    miss
36       4        0    hit
40       5        0    miss
44       5        0    hit
20       2        0    hit
32       4        0    hit
36       4        0    hit
40       5        0    hit
44       5        0    hit
64       8        0    miss
68       8        0    hit
4        0        0    hit
8        1        0    hit
12       1        0    hit
92       11       0    miss
96       12       0    miss
100      12       0    hit
104      13       0    miss
108      13       0    hit
112      14       0    miss
100      12       0    hit
112      14       0    hit
116      14       0    hit
120      15       0    miss
128      16       0    miss
140      17       0    miss
144      18       0    miss
-----
contents after simulation:
row      (v) tags in ways...
0        (1) 18 (1) 17 (1) 16 (1) 15 (1) 14 (1) 12 (1) 13 (1) 11
-----
hits: 18
misses: 14
hit rate: 0.5625
penalty for each miss: 18
total miss penalty: 252
total cycles: 284
cpi: 8.875
lru bits used: 3
total bits used: 648
-----
[u1172983@lab1-8 cache_simulator]$

```

As seen from the output, this fully associative cache missed 14 out of 32 addresses, providing an overall CPI of $(32 + 18 * 14) / 32 = 8.875$. (Note that this simulator can be used to simulate all three types of caches, so this fully associative cache is equivalent to a set-associative cache with only one row. This explains why, in the final state, there is only one row in the cache.)

Direct Mapped: (irrelevant output omitted)

```
u1172983@lab1-8:cache_simulator
addr    tag    row    miss
4        0        0    miss
8        0        1    miss
12       0        1    hit
16       0        2    miss
20       0        2    hit
32       0        4    miss
36       0        4    hit
40       0        5    miss
44       0        5    hit
20       0        2    hit
32       0        4    hit
36       0        4    hit
40       0        5    hit
44       0        5    hit
64       1        0    miss
68       1        0    hit
4        0        0    miss
8        0        1    hit
12       0        1    hit
92       1        3    hit
96       1        4    miss
100      1        4    hit
104      1        5    miss
108      1        5    hit
112      1        6    hit
100      1        4    hit
112      1        6    hit
116      1        6    hit
120      1        7    hit
128      2        0    miss
140      2        1    miss
144      2        2    miss
-----
contents after simulation:
row      (v) tags in ways...
0        (1) 2
1        (1) 2
2        (1) 2
3        (1) 1
4        (1) 1
5        (1) 1
6        (1) 1
7        (1) 1
-----
hits: 20
misses: 12
hit rate: 0.625
penalty for each miss: 18
total miss penalty: 216
total cycles: 248
cpi: 7.75
lru bits used: 0
total bits used: 600
```

This cache missed 12 out of 32 addresses and yielded an overall CPI of 7.75 (248 / 32).

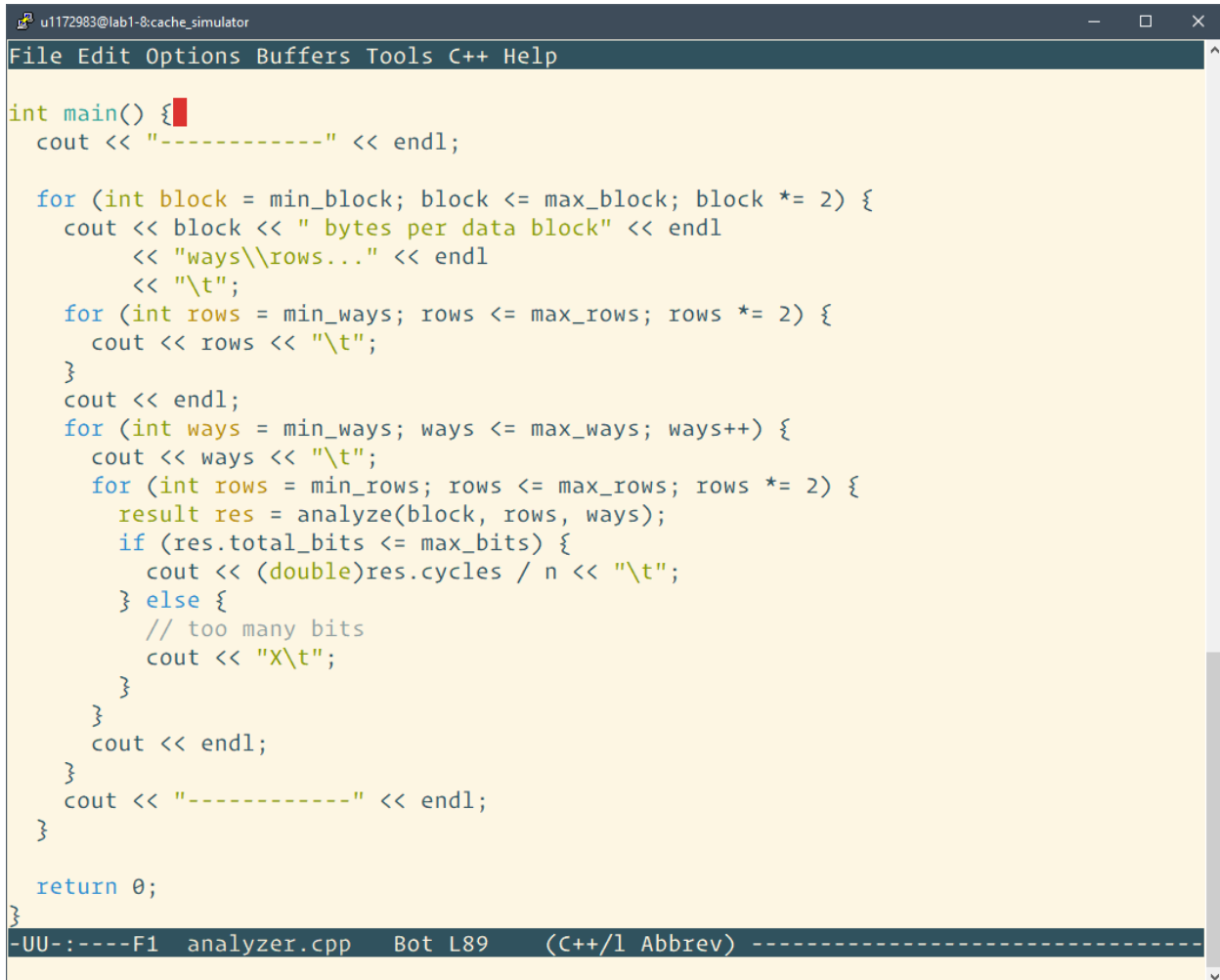
Set-Associative: (irrelevant output omitted)

```
u1172983@lab1-8:cache_simulator
addr    tag    row    result
4        0        0    miss
8        0        1    miss
12       0        1    hit
16       0        2    miss
20       0        2    hit
32       1        0    miss
36       1        0    hit
40       1        1    miss
44       1        1    hit
20       0        2    hit
32       1        0    hit
36       1        0    hit
40       1        1    hit
44       1        1    hit
64       2        0    miss
68       2        0    hit
4        0        0    miss
8        0        1    hit
12       0        1    hit
92       2        3    hit
96       3        0    miss
100      3        0    hit
104      3        1    miss
108      3        1    hit
112      3        2    miss
100      3        0    hit
112      3        2    hit
116      3        2    hit
120      3        3    hit
128      4        0    miss
140      4        1    miss
144      4        2    miss

-----
contents after simulation:
row      (v) tags in ways...
0        (1) 4    (1) 3
1        (1) 4    (1) 3
2        (1) 4    (1) 3
3        (1) 3    (1) 2
-----
hits: 19
misses: 13
hit rate: 0.59375
penalty for each miss: 18
total miss penalty: 234
total cycles: 266
cpi: 8.3125
lru bits used: 1
total bits used: 616
-----
[u1172983@lab1-8 cache_simulator]$
```

This cache missed 13 out of 32 addresses and yielded an overall CPI of 8.3125 (266 / 32).

According to the results, my direct mapped cache did the best job out of the three caches I simulated since it had the lowest overall average CPI of 7.75. This brings up a problem: what type of cache is the best, and with what configuration? To answer this problem, I made another C++ problem (based on the simulator I had) that automatically plugs in different block sizes and row/way counts and simulates the results for each configuration. Here is the critical part of the new program where it tries different values and analyzes them (the rest of the program is essentially the same as the simulator I had above, but with print statements removed):



```

int main() {
    cout << "-----" << endl;

    for (int block = min_block; block <= max_block; block *= 2) {
        cout << block << " bytes per data block" << endl
            << "ways\\rows..." << endl
            << "\\t";
        for (int rows = min_rows; rows <= max_rows; rows *= 2) {
            cout << rows << "\\t";
        }
        cout << endl;
        for (int ways = min_ways; ways <= max_ways; ways++) {
            cout << ways << "\\t";
            for (int rows = min_rows; rows <= max_rows; rows *= 2) {
                result res = analyze(block, rows, ways);
                if (res.total_bits <= max_bits) {
                    cout << (double)res.cycles / n << "\\t";
                } else {
                    // too many bits
                    cout << "X\\t";
                }
            }
            cout << endl;
        }
        cout << "-----" << endl;
    }

    return 0;
}

```

-UU-:----F1 analyzer.cpp Bot L89 (C++/1 Abbrev) -----

Once I ran the analyzer, I got this nice looking tabled output. Note that each table in the output represents a cache with a particular block size, and the rows and columns of a table represent different way and row counts. The numbers inside the table are the overall CPI for the cache with that particular setup; the 'X's indicate that the number of bits used by that particular configuration is too many (more than the provided 840), and we do not analyze it.

```
u1172983@lab1-8:cache_simulator
[ u1172983@lab1-8 cache_simulator]$ g++ analyzer.cpp; ./a.out
-----
4 bytes per data block
ways\rows...
      1      2      4      8     16
1      15     14.5625 12.8125 11.0625 8
2      14.5625 13.25  11.0625 8.4375  X
3      14.5625 11.9375 10.1875  X      X
4      14.125  11.9375 8      X      X
5      11.9375 11.5   X      X      X
6      11.9375 10.625 X      X      X
7      11.9375 9.3125 X      X      X
8      11.9375 9.3125 X      X      X
-----
8 bytes per data block
ways\rows...
      1      2      4      8     16
1      12.8125 12.25  10      7.75  X
2      12.25  9.4375 8.3125  X      X
3      10      9.4375 X      X      X
4      10      8.875  X      X      X
5      10      7.75  X      X      X
6      8.875  X      X      X      X
7      8.875  X      X      X      X
8      8.875  X      X      X      X
-----
16 bytes per data block
ways\rows...
      1      2      4      8     16
1      12.375  9.125  8.3125  X      X
2      9.125  9.125  X      X      X
3      9.125  X      X      X      X
4      8.3125 X      X      X      X
5      8.3125 X      X      X      X
6      X      X      X      X      X
7      X      X      X      X      X
8      X      X      X      X      X
-----
32 bytes per data block
ways\rows...
      1      2      4      8     16
1      12.8125 10.1875 X      X      X
2      8.875  X      X      X      X
3      7.5625 X      X      X      X
4      X      X      X      X      X
5      X      X      X      X      X
6      X      X      X      X      X
7      X      X      X      X      X
8      X      X      X      X      X
-----
64 bytes per data block
ways\rows...
      1      2      4      8     16
1      12.5625 X      X      X      X
2      X      X      X      X      X
3      X      X      X      X      X
4      X      X      X      X      X
5      X      X      X      X      X
6      X      X      X      X      X
7      X      X      X      X      X
8      X      X      X      X      X
-----
[ u1172983@lab1-8 cache_simulator]$
```

As we can see from the output, the King of all configurations (for this particular accessing pattern) is a **fully associative cache with 3 entries and a data block size of 32**. That configuration would provide an overall CPI of 7.5625, which is the lowest out of all configurations that use no more than 840 bits. The output also shows what setups the other two types of caches should use to receive the best performance: a direct map with 8 rows and 8 bytes per block, and a 5-way set associative cache with 2 rows and 8 bytes per block, each achieving a 7.75 overall CPI.

The output of my analyzer also helped understand the relationship between the performance and setup of a cache. As the number of bytes per data block increases, the overall CPIs tend to decrease, and this decrease is more than what the number of rows and ways can affect. This feature suggests that spatial locality is more crucial than temporal locality. Moreover, by plotting the number of bits each configuration uses, we can see a more clear and direct association between it and the decrease in CPI of caches; that is, a cache that uses up more bits tend to provide a better performance.

4 bytes per data block					
ways\rows...					
	1	2	4	8	16
1	47	92	180	352	688
2	96	188	368	720	X
3	147	288	564	X	X
4	196	384	752	X	X
5	250	490	X	X	X
6	300	588	X	X	X
7	350	686	X	X	X
8	400	784	X	X	X

8 bytes per data block					
ways\rows...					
	1	2	4	8	16
1	78	154	304	600	X
2	158	312	616	X	X
3	240	474	X	X	X
4	320	632	X	X	X
5	405	800	X	X	X
6	486	X	X	X	X
7	567	X	X	X	X
8	648	X	X	X	X

16 bytes per data block					
ways\rows...					
	1	2	4	8	16
1	141	280	556	X	X
2	284	564	X	X	X
3	429	X	X	X	X
4	572	X	X	X	X
5	720	X	X	X	X
6	X	X	X	X	X
7	X	X	X	X	X
8	X	X	X	X	X

32 bytes per data block					
ways\rows...					
	1	2	4	8	16
1	268	534	X	X	X
2	538	X	X	X	X
3	810	X	X	X	X
4	X	X	X	X	X
5	X	X	X	X	X
6	X	X	X	X	X
7	X	X	X	X	X
8	X	X	X	X	X

64 bytes per data block					
ways\rows...					
	1	2	4	8	16
1	523	X	X	X	X
2	X	X	X	X	X
3	X	X	X	X	X
4	X	X	X	X	X
5	X	X	X	X	X
6	X	X	X	X	X
7	X	X	X	X	X
8	X	X	X	X	X
