Cache Emulator

README.MD

Description

Name: Jiang Wang

Email: jiangwang@uchicago.edu

This program is written in `C++ 11` and developed on Ubuntu 18.04 LTS. In order to compile and produce an executable, just simply go to the project directory and prompt

```
make
```

which automatically produces a executable called `./cache-sim`.

This project has two subdirectories `lib` and `src`. `lib` contains the definitions and declarations of `FIFO` and `LRU` queues. `src` includes the definitions and declarations of the components of cache, such as `CPU`, `Cache`, and `Rule` (which is the addressing rule).

The executable is "./cache-sim", which supports the following flags:

1. -c val, which cache size (bytes) is determined by val
2. -b val, which block size (bytes) is determined by val
3. -n val, which associativity is determined by val
4. -r str, which replacement policy is determined by str
5. -a str, which the algorithm executed is determined by str
6. -d val, which the input size is determied by val
7. -p, which prints the output the prompt
8. -f, the blocking factor of the block mxm algorithm
9. -l, includes loading data (memory reads) into the results reported

This program provides a fast way to examine the outputs:

2.1 Correctness check

```
make test-daxpy
make test-mxm
make test-mxmblock
```

2.2 Associativity

```
make test-associativity
```

2.3 Memory Block Size

```
make test-block-size
```

2.4 Total Cache Size

```
test-cache-size
```

2.5 Problem Size and Cache Thrashing

```
make test-cache-thrashing-1
make test-cache-thrashing-2
make test-cache-thrashing-3
```

2.6 Replacement Policy

```
make test-replacement-policy
```

Part 1 Specifications

Each word is 8 bytes.

1.1.1 Variable total RAM Size

For the daxpy algorithm, the RAM size is

$$3 * dimension * 8$$

For the mxm and blocked mxm algorithm, the RAM size is

$$3 * dimension * dimension * 8$$

To be careful, the actual RAM size has to be (is padded to) a multiple of block size, despite the terminal shows the number calculated by the two formulas above.

1.1.2 Variable total cache size

The cache size is specified by the `-c` flag. Note that it has to be a power of 2 and a multiple of block size.

1.1.3 Variable block size

The block size is specified by the `-b` flag. Note that it has to be a power of 2.

1.1.4 Block placement strategy

Associativity is specified by the `-n` flag.

(a) Direct-mapped cache is specified by `-n 1`.

(b) Fully associative cache is specified by `-n x`, where x = cache_size / block_size.

(c) n-way ssociative cache is specified by `-n n`.

1.1.5 Block replacement strategy

(a) Random, implemented with random number generator `rand()%set_size`.

(b) Least Recently Used (LRU), implemented with a vector of unordered_maps and a vector of lists. Each element in the vector refers to a set.

(c) First In, First Out (FIFO), implemented with a vector of deques. Each element in the vector refers to a set.

1.1.6 Write Policy

(a) Write-through: The function `Cache::setDouble()` in `src/Cache.cc` shows a write will both update the value in the Cache and the value in the RAM.

(b) Write allocation: The function `Cache::updateBlock()` in `src/Cache.cc` shows when a write miss occurs, the cache controller will fetch the corresponding block from the RAM and put it into Cache.

1.1.7 Algorithms

Please refer to the struct `Algorithms` and the file `src/Algorithm.cc`.

1.4 Measuring Events

Please refer to the class `Result` and the struct `CPU`. The result will print to `stdout` after each run.

One thing to note is that the measuring events will only measure the instructions in the main loop. That is, it will not calculate the instructions when loading dummy data or printing out the results. To give a precise measurement, **the program will clear all the contents in the cache before entering the main loop**. If you want to calculate the events when bothing loading data and executing the main loop, put `-l` flag. You will likely to see a decrease in read miss rate and an increase in write miss rate when you put the `-l` flag.

1.5 Interface

Please refer to the `Parameters`. Also please refer to section 1.4 for information on the `-l` flag.

Part 2 Analysis

2.1 Correctness

You may use the following commands to check correctness of the program:

(1) daxpy

```
make test-daxpy
```

The input is: D = 3; Vector A = [0,1,2,3,4,5,6,7,8]; Vector B = [0,2,4,6,8,10,12,14,16];

The result is: Vector C = [0,5,10,15,20,25,30,35,40];

(2) mxm

```
make test-mxm
```

The input is (A*B):

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 \\ 27 & 28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 \\ 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 \\ 45 & 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 \\ 54 & 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 \\ 63 & 64 & 65 & 66 & 67 & 68 & 69 & 70 & 71 \\ 72 & 73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 \end{bmatrix} \cdot \begin{bmatrix} 0 & 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 \\ 18 & 20 & 22 & 24 & 26 & 28 & 30 & 32 & 34 \\ 36 & 38 & 40 & 42 & 44 & 46 & 48 & 50 & 52 \\ 54 & 56 & 58 & 60 & 62 & 64 & 66 & 68 & 70 \\ 72 & 74 & 76 & 78 & 80 & 82 & 84 & 86 & 88 \\ 90 & 92 & 94 & 96 & 98 & 100 & 102 & 104 & 106 \\ 108 & 110 & 112 & 114 & 116 & 118 & 120 & 122 & 124 \\ 126 & 128 & 130 & 132 & 134 & 136 & 138 & 140 & 142 \\ 144 & 146 & 148 & 150 & 152 & 154 & 156 & 158 & 160 \end{bmatrix}$$

The result is (C):

$$\begin{bmatrix} 3672 & 3744 & 3816 & 3888 & 3960 & 4032 & 4104 & 4176 & 4248 \\ 9504 & 9738 & 9972 & 10206 & 10440 & 10674 & 10908 & 11142 & 11376 \\ 15336 & 15732 & 16128 & 16524 & 16920 & 17316 & 17712 & 18108 & 18504 \\ 21168 & 21726 & 22284 & 22842 & 23400 & 23958 & 24516 & 25074 & 25632 \\ 27000 & 27720 & 28440 & 29160 & 29880 & 30600 & 31320 & 32040 & 32760 \\ 32832 & 33714 & 34596 & 35478 & 36360 & 37242 & 38124 & 39006 & 39888 \\ 38664 & 39708 & 40752 & 41796 & 42840 & 43884 & 44928 & 45972 & 47016 \\ 44496 & 45702 & 46908 & 48114 & 49320 & 50526 & 51732 & 52938 & 54144 \\ 50328 & 51696 & 53064 & 54432 & 55800 & 57168 & 58536 & 59904 & 61272 \end{bmatrix}$$

(3) mxm_block

```
make test-mxmblock
```

The result is same with the `mxm` algorithm

2.2 Associativity

Note: The statistics are obtained from clearing cache before main loop and executing main loop only. Slightly different number are possible with different configurations.

PS: The number instructions for the mxm blocked algorithm of dimension d and blocking factor f_b is

$$IC = 4d^3 + 2\frac{d}{f_b} * d^2$$

For the default setting, $d = 480$ and $f_b = 32$, we would get

$$IC = 4 * 480^3 + 2\frac{480}{32} * 480^2 = 449280000$$

Results:

Table 1. Associativity Table

Cache Associativity	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
1	449280000	220590543	4049457	1.800%	3025920	430080	12.4000%
2	449280000	222812294	1827706	0.814%	3454292	1708	0.0494%
4	449280000	223609177	1030823	0.459%	3454032	1968	0.0569%
8	449280000	223633159	1006841	0.448%	3453332	2668	0.0772%
16	449280000	223620119	1019881	0.454%	3453860	2140	0.0619%
1024	449280000	223569741	1070259	0.476%	3455279	721	0.0209%

Comment:

We can observe that the miss rates do not effectively decrease when $n > 4$. Considering increased power consumption and hit time associated with increased associativity, setting $n = 8$ is a reasonable design decision.

2.3 Memory Block Size

Note: The statistics are obtained from clearing cache before main loop and executing main loop only. Slightly different number are possible with different configurations.

Results:

Table 2. Memory Block size table

Block Size (bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
8	449280000	215930718	8709282	3.880%	3444913	11087	0.321%
16	449280000	219863030	4776970	2.130%	3450272	5728	0.166%
32	449280000	221829247	2810753	1.250%	3452952	3048	0.088%
64	449280000	222812294	1827706	0.814%	3454292	1708	0.049%
128	449280000	223303857	1336143	0.595%	3454962	1038	0.030%
256	449280000	223549700	1090300	0.485%	3455462	538	0.016%
512	449280000	219021505	5618495	2.500%	3368214	87786	2.540%
1024	449280000	212478608	12161392	5.410%	3390769	65231	1.890%

Comment:

We can see the increasing block size to 512 bytes and 1024 bytes increases miss rate. This is because for a limited cache size, once we increase the block size, we have to decrease the number of blocks in the cache. This causes that there are not many free blocks where we can place new blocks. In fact there are only 64 sets (128 blocks) for the 512-byte block cache, and 32 sets (64 blocks) for the 1024-byte block cache. With so limited space for new blocks, the miss rate is very likely to increase.

2.4 Total Cache Size

Note: The statistics are obtained from clearing cache before main loop and executing main loop only. Slightly different number are possible with different configurations.

Results:

Table 3. Total Cache Size table

Cache Size (bytes)	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %	Note
4096	449971200	93791050	130848950	58.200%	0	3456000	100.000%	32 Sets
8192	449971200	200352973	24287027	10.800%	1847145	1608855	46.600%	64 Sets
16384	449971200	210906109	13733891	6.110%	3236156	219844	6.360%	
32768	449971200	220271316	4368684	1.940%	3452419	3581	0.104%	
65536	449971200	222812294	1827706	0.814%	3454292	1708	0.049%	
131072	449971200	223497813	1142187	0.508%	3455348	652	0.019%	
262144	449971200	223845631	794369	0.354%	3455663	337	0.010%	
524288	449971200	223948173	691827	0.308%	3455928	72	0.002%	
32768	449280000	223537141	1102859	0.491%	3451702	4298	0.124%	8-way

Comment:

A 256 KB cache is required to achieve a data read miss rate of equal to or less than 0.5% (Note that including loading instructions and/or not clearing out the cache before the main loop may require a smaller cache, such as 128 KB).

The last row of the table shows that a 8-way 32 KB cache is able to yield a data read miss of 0.491%, less than 0.5%.

2.5 Problem Size and Cache Thrashing

Note: The statistics are obtained from clearing cache before main loop and executing main loop only. Slightly different number are possible with different configurations.

Results:

Table 4: Matrix Multiply Problem size table - Associativity = 2

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-	442598400	111819337	109364663	49.400%	0	230400	100.000%
480	Blocked	32	449280000	222812294	1827706	0.814%	3454292	1708	0.049%
488	Regular	-	465095232	216576463	15852081	6.820%	208376	29768	12.500%
488	Blocked	8	493910656	244048288	2907040	1.180%	14525244	1540	0.011%
512	Regular	-	537133056	133891042	134544414	50.100%	0	262144	100.000%
512	Blocked	32	545259520	117211577	155418183	57.000%	0	4194304	100.000%

Table 5: Matrix Multiply Problem size table - Associativity = 8

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-	442598400	109190240	111993760	50.600%	0	230400	100.000%
480	Blocked	32	449280000	223633159	1006841	0.448%	3453332	2668	0.077%
488	Regular	-	465095232	216200250	16228294	6.980%	208104	30040	12.600%
488	Blocked	8	493910656	244134390	2820938	1.140%	14525466	1318	0.009%
512	Regular	-	537133056	133595449	134840007	50.200%	0	262144	100.000%
512	Blocked	32	545259520	117003791	155625969	57.100%	0	4194304	100.000%

Table 6: Matrix Multiply Problem size table - Fully Associative

Matrix Dimension	MxM Method	Blocking Factor	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
480	Regular	-	442598400	204347589	16836411	7.610%	201600	28800	12.500%
480	Blocked	32	449280000	223569741	1070259	0.476%	3455279	721	0.021%
488	Regular	-	465095232	212538262	19890282	8.560%	208376	29768	12.500%
488	Blocked	8	493910656	244141746	2813582	1.140%	14524997	1787	0.012%
512	Regular	-	537133056	246514644	21920812	8.170%	229376	32768	12.500%
512	Blocked	32	545259520	271328511	1301249	0.477%	4193386	918	0.022%

Comment:

2.5.1

Both of the algorithms yield different performance results when different problem sizes are given.

Note: The 2-way associative cache has 512 sets and 1024 blocks.

(i) **`mxm` algorithm**

The `mxm` algorithm yields extremely high miss rate when $d = 480$ and $d = 512$, but the performance increased significantly when $d = 488$. Let's inspect the inner loop of the `mxm` algorithm to explain this:

```
r0 = 0; // Line 1
      for (auto k=0; k!=test_size; ++k) { // Line 2
        r1 = myCpu.loadDouble(a[row*test_size + k]); // Line 3
        r2 = myCpu.loadDouble(b[k*test_size + col]); // Line 4
        r3 = myCpu.multDouble(r1, r2); // Line 5
        r0 = myCpu.addDouble(r0, r3); // Line 6
      } // Line 7
      myCpu.storeDouble(c[row*test_size+col], r0); // Line 8
```

Important facts: $d = 480$ means a row has 60 blocks, $d = 488$ means a row has 61 blocks, $d = 512$ means a row has 64 blocks.

(i-1) $d = 512$. This problem size implies that the elements in the same position in the three matrices are mapped to the same set in the cache.

Read Miss. Observe that Line 3 loads the data from `a` sequentially, but Line 4 loads the data from `b` skipping 64 blocks. That is in each iteration, elements from `b` only access the blocks x, x+64, x+128, ..., x+512, which x and x+512 maps to the same set. That is if we load data from `b` only, we needs to replace cache blocks starts from the 17th iteration (1,9,17 maps to the same set, but with associativity = 2, we need to replace the block). We need to replace 512-16 = 496 times at least when we load data from `b`.

Write Miss. Observe the algorithm, to calculate $c[i,j]$, we must at some point load $a[i,j]$ and $b[i,j]$ into the cache, and because the associativity is 2, one of the two loads will replace the c block that was loaded to cache in the previous iteration, so write miss rate is 100%.

(i-2) $d = 480$.

Read Miss. Line 4 loads data from `b` skipping 60 blocks. The data access pattern is x, x+60, x+120, ..., x+480. The gcd of 60 and 512 is 4. That is loads from `b` only uses cache blocks in sets whose index is a multiple of 4. The rest are not used.

Write Miss. The same reason in (i-1) but the calculation is more complicated. Loads from `a` and `b` replace the block of `c` in the cache that is loaded from previous iteration.

(i-3) $d = 488$.

Read Miss. Note that each row has 61 blocks, and the gcd of 51 and 512 is 1. So loads from `b` is possible to put blocks into any blocks in the cache, reducing the conflict and thus yielding better results.

Write Miss. Since loads from `b` is possible to load into any cache set, each iteration will only occupy 61 (from `a`) + 488 (from `b`) = 549 blocks. We have 1024 blocks in the cache so it is very likely the `c` block in the cache is not replaced.

(ii) **`blocked mxm` algorithm**

$d = 512$ yields to worst performance. This is because, as explained in (i-1), that loads from `b` will only occupy 8 (512/64) sets of the 512 sets in the cache, and conflicts will occur at least at the 17th load from `b`. With a blocking factor of 32, at least half of loads from `b` are misses.

2.5.2

No, the `blocked mxm` algorithm does not yield better performance compared with the naive `mxm` algorithm. This is because loads from `b` will only occupy 8 of the 512 sets of the cache in each iteration of the element in `c`.

2.5.3

Using fully associative cache will help improve the performance of the `mxm` and `block mxm` algorithm, especially when the dimension is 512. This is because without a fully associative cache, in each iteration of $c[i,j]$, loads from `b` can only be mapped to certain sets in the cache, and the others (sets and blocks) are wasted. Using a fully associative cache will make all the blocks avaiable and reduce conflicts.

A fully associative cache has several issues. One of them is the increase of hit time, we need to search through all the blocks (at worst) to find the block we need. Another is the increase of power consumption. The additional work would require more energy and is thus not suitable for devices that requires a long battery life.

2.5.4

The first advice would to use the **block mxm algorithm** with **a small blocking factor**. Note that the number of sets in the cache must be a power of 2, call it a. We know that gcd(a,512)!=0, and the fact conflicts would occur after several loads. We can make the blocking factor small to avoid conflicts in later loads. With an 8-way cache, and input dimension of 512, and a blocking factor of 32, we have a read miss rate of 57.1% and write miss rate of 100%. When reducing the blocking factor to 8, we have a read miss rate of 1.63% and 0.962%.

The second advice is to use the naive **mxm algorithm** with **loop interchange**. The naive code can be expressed as:

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    for (int k = 0; k < n; ++k)
      c[i][j] += a[i][k] * b[k][j];
```

Notice that each change of k would cause a cache miss when loading `b` into memory, by interchanging the loop of j and loop of k, we would have

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    for (int j = 0; j < n; ++j) // swap k and j
      c[i][j] += a[i][k] * b[k][j];
```

The inner loop of j only reads the memory sequentially and j increases (thus no conflict misses), so we can ensure as few misses as possible in the inner loop and the algorithm overall.

2.6 Replacement Policy

Note: The statistics are obtained from clearing cache before main loop and executing main loop only. Slightly different number are possible with different configurations.

Results:

Table 7: Replacement Policy Table

Replacement Policy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
Random	449280000	222904611	1735389	0.773%	3400022	55978	1.620%
FIFO	449280000	222909299	1730701	0.770%	3423073	32927	0.953%
LRU	449280000	222812294	1827706	0.814%	3454292	1708	0.049%

Table 8: Replacement Policy Table (Includes loading data)

Replacement Policy	Instructions	Read Hits	Read Misses	Read Miss %	Write Hits	Write Misses	Write Miss %
Random	449971200	222905338	1734662	0.772%	4004985	142215	3.430%
FIFO	449971200	223046909	1593091	0.709%	4059466	87734	2.120%
LRU	449971200	223266764	1373236	0.611%	4060800	86400	2.080%

Comment:

LRU has far lower write miss rate than FIFO and random, but slightly higher read miss rate than Random and FIFO. This is caused by the access pattern of the algorithm (that some sets are never accessed in an iteration).

I includes the new data including loading data in Table 8. Loading data is inherently more sequential and representative of the normal I/O pattern. Table 8 shows LRU results in the best performance among the three replacement policies.