# Cache Simulator

The evaluation and simulation of caches is the main focus of this activity. Several memory traces from actual benchmark programmes will be provided to you. You'll write a programme to mimic the actions of various caches on these traces. After that, you'll utilise your program and the provided traces to figure out what the optimal cache arrangement. The program has to be written in C++/C [not Java or Python for reason given below]. You're allowed to use the standard library of your chosen language as much as you would like to, but you are not allowed to use any additional (non-standard) libraries.

Your program should execute without ***memory errors or memory leaks***. Memory errors such as invalid reads or write, or uses of uninitialized memory, will result in a deduction of up to 10 percentage points. Memory leaks will result in a deduction of up to 5 percentage points.

You must provide a Makefile such that

- make clean removes all object files and executables, and
- make or make cacheSim compiles and links your program, producing an executable called csim

Your code should compile cleanly with gcc 9.x [on Ubuntu] using the *-Wall -Wextra --pedantic* compiler flags. Important: your Makefile must use these options. If your Makefile does not compile your code with these options, you will forfeit all of the points for design and coding style.

## Part A: Cache Simulator [cacheSim]

You will design and implement a cache simulator that can be used to study and compare the effectiveness of various cache configurations. Your simulator will read a memory access trace from standard input, simulate what a cache based on certain parameters would do in response to these memory access patterns, and finally produce some summary statistics to standard output. The file format of the memory access traces:

```
s 0x1fffff50 1
l 0x1fffff58 1
l 0x1fffff88 6
l 0x1fffff90 2
l 0x1fffff98 2
l 0x200000e0 2
l 0x200000e8 2
l 0x200000f0 2
l 0x200000f8 2
l 0x30031f10 3
s 0x3004d960 0
s 0x3004d968 1
s 0x3004caa0 1
s 0x3004d970 1
s 0x3004d980 6
l 0x30000008 1
l 0x1fffff58 4
l 0x3004d978 4
l 0x1fffff68 4
l 0x1fffff68 2
s 0x3004d980 9
l 0x30000008 1
```

Each memory access performed by a program is recorded on a separate line. There are three "fields" separated by white space. The first field is either l or s depending on whether the processor is "loading" from or "storing" to memory. The second field is a 32-bit memory address given in hexadecimal; the 0x at the beginning means "the following is hexadecimal" and is not itself part of the address. You can ignore the third field for this assignment.

Note that you should assume that each load or store in the trace accesses at most 4 bytes of data, and that no load or store accesses data which spans multiple cache blocks (a.k.a. “lines”.)

Your cache simulator will be configured with the following cache design parameters which are given as command-line arguments (see below):

- number of sets in the cache (a positive power-of-2)
- number of blocks in each set (a positive power-of-2)
- number of bytes in each block (a positive power-of-2, at least 4)
- *write-allocate* or *no-write-allocate*
- *write-through* or *write-back*
- *lru* (least-recently-used) or *fifo* evictions

Note that certain combinations of these design parameters account for direct-mapped, set-associative, and fully associative caches:

- a cache with n sets of 1 block each is direct-mapped
- a cache with n sets of m blocks each is m-way set-associative
- a cache with 1 set of n blocks is fully associative

The smallest cache you must be able to simulate has 1 set with 1 block with 4 bytes.

Your cache simulator should assume that loads/stores from/to the cache take one processor cycle; loads/stores from/to memory take 100 processor cycles for each 4-byte quantity that is transferred.

We expect to be able to run your simulator as follows:

```
./cacheSim 256 4 16 write-allocate write-back lru < sometracefile
```

This invocation should produce the following output:

```
Total loads: 318197
Total stores: 197486
Load hits: 314171
Load misses: 4026
Store hits: 188047
Store misses: 9439
Total cycles: 9845283
```

This would simulate a cache with 256 sets of 4 blocks each (aka a 4-way set-associative cache), with each block containing 16 bytes of memory; the cache performs write-allocate but no write-through (so it does write-back instead), and it evicts the least-recently-used block if it has to.

Example Traces are provided.

## Part B: Best Cache

For part (b), you’ll use the memory traces as well as your simulator to determine which cache configuration has the best overall effectiveness. You should take a variety of properties into account: hit rates, miss penalties, total cache size (including overhead), etc. In your README,

describe in detail what experiments you ran (and why!), what results you got (and how!), and what, in your opinion, is the best cache configuration of them all.

Submit a well formatted 3 page report.

## Part C : ~~Multi Level Cache Good~~ Intelligent Coding

Recall that we discussed that the way you write programs, has an impact on the performance. Matrices are 2-dimensional data structures wherein each data element is accessed via two indices. To multiply two matrices, we can simply use 3 nested loops, assuming that matrices A, B, and C are all n-by-n [dynamically allocated 2D arrays in C:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        C[i][j]=0;
        for (int k = 0; k < n; k++)
            C[i+j*n] += A[i+k*n] * B[k+j*n] OR C[i][j] = A[i][k]*B[k][j];
```

Matrix multiplication operations are at the heart of many linear algebra algorithms, and efficient matrix multiplication [*time taken*] is critical for many applications within the applied sciences. To compute the matrix multiplication correctly, the loop order doesn't matter. Appropriately initialize the matrices with random numbers. Compile the code with the -O3 flag in gcc in a linux box. Sweep across values of **n** to find out the cross over points where the order in which loops are executed makes a difference. Plot a graph to illustrate this. From the observations can you infer the size and other parameters of the cache in question?

Repeat the same for Matrix Transpose --- both in place and using an additional array.

Please submit a well formatted pdf [<=3 pages].

~~Part D : Caches in real Processors~~

## Submission:

Create a zipfile that has your Makefile, source and header files, and README file. All of the files can be in the top level directory of the zipfile or you could have a directory structure. The executable cacheSim must be generated at the root.

## Late Submission:

25% per 24 hours

## Generative AI usage:

To guard against indiscriminate [aka use without understanding], we will evaluate your submissions against a lab test which we will hold on the 21st of April[830AM-130PM]. Portions of Part C and Part D will be assigned to you at random which you will implement to augment the codebase that you submit for Parts A and B. Therefore, my suggestion would be that you do the assignments on your own. The lab test subsumes the viva.