CS-320 Spring 2025 Project 2

1 Project Goals

The goal of this project is to understand the effects of cache hardware by simulating the performance of different cache configurations.

Several benchmark programs have been run with special monitoring software to capture each memory reference address, and whether that reference is a read from memory (load) or a write to memory (store). All trace records collected by running a single benchmark program are put into a trace file.

Your challenge for this project is to write a C or C++ program which reads a trace file and simulates cache hardware to determine whether there is a cache hit or cache miss. Your program will need to keep information normally kept in the cache hardware, such as valid bits or cache tags, so it can determine if the next memory access is a hit or a miss.

You will need to collect statistics on several different cache orientations – the details are described in section 2.4 on page 3.

You will be provided with the correct results for a couple of benchmarks so you can check your work. Your project submission will be checked against both the published results, against benchmarks whose results have not been published, and using traces from benchmarks which have not been provided to you.

2 Project Specifications

2.1 Program Specifications You must submit both C or C++ code and a Makefile to build an executable file. Your program should read a single trace file from standard input. The specifications for the trace file appear in section 2.2 on the next page.

For each trace line, your program should determine whether the cache hits or misses for each cach configuration defined in section 2.4 on page 3. Statistics should be kept for each cache configuration. See section 2.3 on the next page for details on which statistics to keep.

After all trace lines have been read, your program should print a report for each cache configuration, and determine which cache configuration had the highest hit rate. See section 2.6 on page 5 for specifics on what reports to produce.

2.2 Trace File Specfications The trace file is an ASCII file that consists of trace records separated by new-lines (\n). Each trace record consists of two blank separated fields:

- A single letter, "S" or "L", to indicate whether this memory access is a store (memory write) or load (memory read).
- The 64 bit address of the memory to be read or write, expressed in hexadecimal, 0xDDDDDDDDDDDDDDDD, where D is a hexadecimal digit 0-f. Leading zeroes may be truncated.

You may assume each memory read is a single byte (and therefore never crosses into a new cache line.)

The following C scanf invocation reads a single line of the trace file:

```
char SorL;
long int address;
while(!feof(stdin)) {
... scanf("%c 0x%lx ",&SorL,&address) ...
```

The following C++ code performs a similar function:

```
string SorL;
long int address;
while(std::cin >> SorL >> std::hex >> address) {
   ...
```

2.3 General Cache Specifications Your simulator needs to support several cache configuration. Each configuration should support a a "probe" function which gets arguments that define whether the current request is a memory read or write, and the address of the data requested.

A real cache would need a third argument with the data value for a memory write, and a return of the memory value for a memory read; but we won't need that level of detail for our simulation.

Two statistics should be kept for each cache configuration. The first keeps track of the total number of requests, and should be incremented once for each probe. The second keeps track of how many cache hits occurred for this cache configuration; how many probes for which the requested memory was already in the cache.

2.4 Required Cache Configurations The following cache configurations need to be supported. See the table at section 2.5 on page 5 for a summary of the configurations.

2.4.1 Direct Map Caches where the selection of which cache line is determined from the low order bits of the cache line index. All cache lines will be 32 bytes long, but the total size of the cache varies. The cache sizes for direct map caches will be 1Kb, 4Kb, 16Kb, and 32Kb.

2.4.2 Fully Associative Caches where any cache line may be used for any request. Cache lines will be 32 bytes long and the total cache size will be 16Kb. Use a cache replacement policy to determine which cache line should be evicted on a cache miss.

Two different replacement policies need to be supported: a true least recently used (LRU) replacement policy, and a Pseudo-LRU policy that uses hot and cold bits to determine the victim cache line.

Each hot/cold bit should be 0 if the left child is "hot" (recently used), or 1 if the right child is "hot". At the top level, the hot/cold bit represents whether the first half of the cache or the second half of the cache was used recently - the next level divides each half in half again. Support enough levels to determine if each line was most recently used or not.

To determine a victim, follow the "cold" half at each level to get to the victim line.

The following is an example of hot/cold bits for set associativity of 16. (The indexes and set values are represented in hexadecimal to save table space.)

index	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
level	0	1		2				3							
0 from-to	0-7	0-3	8-B	0-1	4-5	8-9	C-D	0	2	4	6	8	A	C	E
1 from-to	8-F	4-7	C-F	2-3	6-7	A-B	E-F	1	3	5	8	9	B	D	F
hc[0]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
hc[1]	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1

In this table, the "index" row is the bit index of the hot/cold bits. The "level" row shows which level of hot/cold these bits apply to. The "0 from - to" rows show the sets that are the "hot" half when the hot/cold bit is zero. The "1 from - to" row shows the sets that are the "hot" half when the hot/cold bit is one. Then the hc[i] rows represent the hot/cold bits at time i.

To start with, all hot/cold bits are initialized to zero. To find the first victim, look at hc[0] at index 0 to determine that the cold half are sets 8-15. Then look at index 2 to narrow that down to 12-15, index 6 to narrow that down to 14-15, and index 14 to determine that the first victim should be set 15.

Once we have chosen 15 as our first victim, we load the required line into set 15, and update our access bits to indicate that 15 is the "hot" set. to do so, we need to set the hot/cold bits at index 14, 6, 2, and 0 to one, resulting the hc[1].

For Pseudo-LRU, use the hot/cold bits for **all** accesses, including the cases where the initial data is written to the cache lines, as well as selecting victims when the cache is already full.

2.4.3 Set Associative Caches a hybrid between direct map caches and fully associated caches in which bits from the requested address choose an index into the cache like a direct-map caches, but in which several sets are used to hold cache data like associative caches.

Each cache line will be 32 bytes, and the total cache size will be 16Kb. Support different levels of associativity, where the number of sets are 2, 4, 8, and 16; with a least-recently-used (LRU) replacement policy. Also support all four levels of associativity for three variations on set associative caches: no allocation on write miss; next-line prefetch; and next-line prefetch only on miss.

For the no allocation on write miss variation, if a store instruction is a miss, then the missing line is not saved in the cache, but instead written directly back to memory.

The next-line prefetch variation causes each access of line with index X to also ensure that line X+1 (where X+1 is one line away from X) is also in the cache. If memory at X+1 is not already in the cache, a victim line should be chosen using the LRU policy, and the X+1 memory should replace that line. Note that when line X is accessed, both line X and line X+1's LRU information should be updated. Line X+1 should become the least recently used line in the cache.

The next-line prefetch only on miss variation is the same as next-line prefetch, but the next line processing should **only** occur if accessing line X resulted in a cache miss.

2.5 Cache Configuration Summary The following table defines all the cache configurations that are required for this project:

	Description
1	direct map 1KB
2	direct map 4KB
3	direct map 16KB
4	direct map 32KB
5	fully assoc true-LRU
6	fully assoc psuedo-LRU
7	set assoc 2
8	set assoc 4
9	set assoc 8
10	set assoc 16
11	set assoc 2 skip on write miss
12	set assoc 4 skip on write miss
13	set assoc 8 skip on write miss
14	set assoc 16 skip on write miss
15	set assoc 2 prefetch
16	set assoc 4 prefetch
17	set assoc 8 prefetch
18	set assoc 16 prefetch
19	set assoc 2 prefetch on miss
20	set assoc 4 prefetch on miss
21	set assoc 8 prefetch on miss
22	set assoc 16 prefetch on miss

2.6 Cache Reports Each cache configuration needs to print out a report line once an entire trace file has been processed. Each trace line consists of the following fields:

```
<Hit Percent> - <Description>
```

The Hit Percent consists of 100.0 times the ratio of cache hits divided by the number of cache probes. The Hit Percent should be expressed using 7 total characters; three for the integer part of the percentage, one for the decimal point, and three decimal places. The result should be rounded to the nearest third decimal place, followed by a percent sign (%).

This result can be obtained using the following C printf statement:

```
printf(\%7.3lf%% - %s\n",
   (100.0 * hits)/probes),
   description_string);
```

Similar results can be formatted using C++, but I'll leave the details up to you.

The description should be exactly the description from the table in section 2.5 on the preceding page. The cache reports should be printed for all 22 cache configurations defined in section 2.5 on the previous page.

Finally, you should report on the highest hit rate percentage achieved with a line of the form:

```
Best hit rate:  <Hit Rate>
```

That reports on the highest hit rate percentage achieved by any configuration. (You do not need to report on which configuration... just the percentage.)

3 Working the Project

The proj2.tar.gz file is available for download from Brightspace. If you download this file and untar it using the command tar -xvzf proj2.tar.gz this will create a new directory in the current directory called proj2. This directory contains:

- Project2_instructions.pdf : These instructions.
- traces/trace1.txt : The first trace file.
- traces/trace2.txt : The second trace file.
- traces/trace3.txt : The third trace file.
- results/trace1_output.txt : The expected results from the first trace file.
- results/trace2_output.txt : The expected results from the second trace file.
- proj2sol/Makefile : An example Makefile.

Design and code a branch simulator in the proj2sol subdirectory using the executable file name "cacheSim" that produces the expected results for the first and second trace files, and valid results for the third trace file. The cacheSim executable should read a trace file from **standard input**, and write only the expected results to standard output. (If you want extra error messages, they must be written to standard error.)

Your solution should be kept in the proj2sol subdirectory, and should contain all the C or C++ code required to build the cacheSim executable. Edit the Makefile so that there is a "cacheSim" target to build your cacheSim executable. You may also provide a "clean" target to remove your cacheSim executable and any other files created by the Makefile.

As long as your solution is in the proj2sol subdirectory, you can run "make submit" to create a tar file to submit on Brightspace.

Warning – Brightspace prohibits the presence of executable files when you upload, so the cacheSim executable file must not be present in the proj2sol directory when the submission tar file is built.

You may submit as many times as you want before the project deadline. Only the last submission will be graded.

4 Grading Criteria

Your proj2.tar.gz file will be downloaded and unzipped on a remote.cs server. We will then build your cacheSim executable using your Makefile's cacheSim target. The cacheSim executable will be run using trace1.txt, trace2.txt, trace3.txt, and an unpublished trace file. You are expected to get correct results for all four trace files.

If the compiler produces error messages while building your code, we will attempt to fix the problem, with a 15 point deduction for each fix required.

If the compiler produces warning messages while building your code, we will deduct 10 points. (Warning messages often indicate potential bugs... it's worth fixing them!)

General errors (such as not reading trace files from standard input, or not providing a cacheSim target in your Makefile) will result in minor (10 points or less) deductions, depending on the severity of the error.

After your code is built, it will be run on the remote.cs servers using all four trace files. For each incorrect line produced, one point will be deducted.