CS320 – Spring 2016: Project 2

Blackboard submission due: Saturday, April 30th at 10pm
Project presentations: lab session on Tuesday, May 3rd and office hours

The goal of this project is to measure the effectiveness of cache subsystem organizations using traces of memory instructions obtained from the realistic programs. Each trace contains about 1 million memory instructions with two values provided for each instruction: a flag indicating whether this is a load or a store (L stands for a load, S stands for a store), and the byte memory address targeted by this instruction. Three traces are provided.

Your goal is to write a program in C or C++ that would use these traces to measure the **cache hit rate** of various data cache organizations and prefetching techniques (note: we are not estimating the instruction cache performance in this project, only the data cache). Specifically, the following cache designs have to be implemented.

1) **[10%] Direct-Mapped Cache.** Assume that each cache line has a size of 32 bytes and model the caches sized at 1KB, 4KB, 16KB and 32KB
2) **[20%] Set-Associative Cache**. Again, assume that the cache line size is 32 bytes and model a 16KB cache with associativity of 2, 4, 8 and 16. Assume that the least recently used (LRU) replacement policy is implemented.
3) **[20%] Fully-Associative cache.** Assume that each cache line is 32 bytes and the total cache size is 16KB. Implement Least Recently Used (LRU) and hot-cold LRU approximation policies. For the hot-cold LRU approximation policy the initial state of all hot-cold bits should be 0 corresponding to the case where the left child is "hot" and the right child is "cold". Furthermore, the policy should be utilized (and updated) for **all** accesses, including placing the initial blocks into the cache as well as replacements once the cache is full.
4) **[10%] Set-Associative Cache with no Allocation on a Write Miss.** In this design, if a store instruction misses into the cache, then the missing line is not written into the cache, but instead is written directly to memory**.** Evaluate this design for the same configurations as in question (2) above.
5) **[20%] Set-Associative Cache with Next-line Prefetching.** In this design, the next cache line will be brought into the cache with every cache access. For example, if current access is to line X, then line (x+1) is also brought into the cache, replacing the cache's previous content. Evaluate this design for the same configurations as in question (2) above. Note that prefetched blocks **should**

update the LRU order of the corresponding set meaning that the prefetched block should become the most recently used block in its set.

6) **[20%] Prefetch-on-a-Miss.** This is similar to part (5) above, but prefetching is only triggered on a cache miss. (Prefetched blocks should update the LRU order as in part 5).

Extra credit problem [20%]: Propose and implement a new cache replacement or prefetching mechanism that outperforms either the LRU replacement or the next-line prefetcher. Evaluate your proposal on the designs listed in question (2) above. Feel free to use any supplementary literature that you can find on this subject. If you choose to do the extra credit you should provide an explanation of what you have done, how to test it, and why it works in the README file in your submission.

<u>Materials on Blackboard:</u>

There is a tar/gzipped archive of materials on Blackboard that contains the following:

A directory called project2/, containing the following files:

- sample_output.txt – Sample output file with comments
- traces/ – Directory containing three trace files
- correct_outputs/ – The correct output for each of the provided traces

To access these materials, download a copy from Blackboard, cd into the directory where you placed the tar/gzipped archive and issue the following command:

```
tar -xzvf project2.tar.gz
```

This will create a new directory (named project2/) containing the files mentioned above.

<u>Submission requirements:</u>

Please submit ONLY the following things, and pay close attention to naming. Remove any trace files or test output files. Make sure your Makefile builds a correctly named executable. Ensure that you have included an ASCII text file named README (not README.txt just README, exactly like that no lowercase!)

You will need to submit your source code, so that we can compile it and test for correctness. For checking your code, we will be using the same three traces that have been provided to you, plus one more trace that you will not have access to.

The code that you submit should compile into a single executable called **cache-sim** with a simple `make` command. This executable should run all of the caches on the given trace, which will be specified via command line options as follows:

```
./cache-sim input_trace.txt output.txt
```

Where:

-input_trace.txt – file name of file containing branch trace

-output.txt – file name of file to write output statistics

The output file should have the following format: (an example text file is on Blackboard too with comments, which should not be output by your program)

```
x,y; x,y; x,y; x,y; x,y;

x,y; x,y; x,y; x,y;

x,y;

x,y;

x,y; x,y; x,y; x,y;

x,y; x,y; x,y; x,y;

x,y; x,y; x,y; x,y;
```

Where each x,y; pair corresponds to the number of cache hits (x) and the total number of accesses (y) of one of the cache configurations. The first line provides the results for the direct mapped caches, second line for set associative, the third line for the fully associative cache with LRU replacement, the fourth line for the fully associative cache with hot-cold replacement, the fifth for the associative caches without store allocation, the sixth line for associative caches with next line prefetching and the seventh line for associative caches with next line prefetching only for cache misses. The numbers within each line should be separated by a single space.

Submissions will be checked using a script that will compare your output file to the correct output file using the UNIX `diff` tool, so if your output does not **EXACTLY** match the correct output the grading program will mark it as wrong. I will have to check such submissions by hand which will result in at least a few points being deducted.

<u>Submission Rules:</u>

You must submit all of the following:

1.) All source code

2.) A Makefile

3.) A README, which minimally contains the Name, BU-ID (everything before the @ in your Binghamton University e-mail) and B Number of each person in the group. Other things to include might be: what works/what doesn't, things you found interesting, etc.

These materials should be turned in as follows: (using Jesse's name and BU-ID as an example)

My e-mail is jelwell1@binghamton.edu so my BU-ID is jelwell1

1.) Create a new directory whose name is your BU-ID:

```
mkdir jelwell1/
```

2.) Copy all relevant files into this new directory

3.) Create a tar/gzipped archive whose name is also your BU-ID from the directory as follows:

```
tar -czvf jelwell1.tar.gz jelwell1/
```

(Should output name of all archived files, make sure there are no .o files, executables, traces, outputs, etc. in this list before submission)

4.) Submit tar/gzipped archive via Blackboard