cs281: Introduction to Computer Systems

HW5 – cachelab

Assigned: October 24, Due: Mon., October 31, 11:59pm

1 Overview

This lab tests your understanding of memory cache by asking you to create a cache simulator. The simulator is like a cache, except that it has no requirement to actually store the memory contents–it only performs the bookkeeping of a cache and maintains performance related metrics including the number of hits, misses and evictions. You need to understand the following concepts in order to implement the cache simulator:

- cache hit, miss, eviction
- block offset bits
- set index bits
- associativity
- replacement policy

2 The Cache Lab

2.1 Generating Memory Traces

On all the machines in Olin 219, there is a tool called valgrind, which can be used to record all memory access of the execution of a given binary. As an example, you can execute:

```
linux> valgrind --log-fd=1 --tool=lackey --trace-mem=yes echo cachelab
```

The above command runs "`echo cachelab`" with valgrind and displays a trace of the memory accesses to `stdout`.

In the output, you should see entries such as the following:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

The format of memory reference lines in the trace is <op addr,size>. In the op (operation) field, "I" stands for instruction load; "L" stands for data load; "S" stands for data store; and "M" stands for data modify, which should be treated as a load immediately followed by a store. The memory address (addr) is given in hex format without the leading '0x', followed by comma and then size, indicating the number of bytes accessed.

2.2 Cache Simulator Specification

Your goal is to write a cache simulator that can take the memory traces from valgrind as input, and simulate a cache and output the number of cache hits, misses and evictions.

Your cache simulator should be able to handle different cache sizes and associativity. More specifically, your cache simulator should take the following arguments on the commandline:

-b: number of block bits (so 2^b is the block size B)

-s: number of set index bits (so 2^s is the number of sets S)

-E: associativity (number of lines per set)

-t: file name of the trace to replay/simulate

Note that we have adopted the same notation (s,S,b,B,E) as in your textbook CSAPP2e page 597. Your cache simulator should use the LRU (least recently used) replacement policy for evictions when $E > 1$.

Your job is to fill in the (mostly) empty cachesim.c file. To help you get started, we have included the code to parse commandline options.

Note that the skeleton cachesim.c provides for an optional argument -v, which can be used in your code to enable verbose output. Using this option will help you debug your cache simulator you could use it, for instance, to prints the hits, miss, and evictions after each reference or additional bookeeping information. Note that when verbose is off, you should print *nothing* except the results summary printed by the `printCachsimResults()` function.

2.2.1 Important Notes

- In order for me to evaluate your results, at the end of your cache simulator, you must include a call to the function `printCachesimResults()` with the number of cache hits, misses and evictions. While this currently simply prints the results in a standard form (that can be readily parsed by automated tools), the eventual goal is for this function to interact with a grading server in the way that our earlier projects do, so that immediate feedback becomes possible. You will find the call at the end of main() in cachesim.c, please do not delete that line or you will not get credit.
- In similar vein, it is important for your simulator, given valid arguments and a valid valgrind trace, perform **no other** output beyond the metrics at the end. To do so would make autograding and the immediate feedback feature of such a system unworkable.
- For this lab, we are interested only in *data* cache performance, so you should ignore the instruction cache accesses (lines starting with I). Notice that Valgrind always put I in the first column, and M, L, S in the second column. This may help you parse the trace.
- For the purpose of this lab, you should assume that memory accesses are aligned properly such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the Valgrind traces.
- The name of the executable must be `cachesim` with no extension.

- The basic language used in this course is C. And so the skeleton given you for this assignment is also a C source file. You *may*, however, use any language you like to complete the simulator. The caveat, however, is that I must be able to invoke the program and pass arguments in *exactly* the same way as the given skeleton. So executable name, command line arguments, and output format must all be the same. A port of what I have given you to C++ would probably not be a difficult task.
- This assignment must be completed on an *individual* basis. You are welcome to share trace files with one another and to cite your final results, given a particular trace. This allows a degree of checking for accuracy (or at least of consistent-between-students mistakes ;-).

2.3 Evaluation

I will run your simulator using different parameters (s, b, E) and on different traces. For each test case, outputting the correct number of cache hits, misses, AND evictions will give you full credit for that test case. Each of your reported number of hits, misses and evictions is worth $\frac{1}{3}$ of the credit for that test case. That is, if a particular test case is worth 3 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 2 points. There are seven test cases each worth 3 points respectively. The last two cases are particularly long traces that typically would fail if you have a memory leak or some other latent bug that takes a long and varied input sequence to show itself.

The performance tests above give you a total of 21 points. I will assess the quality of your source code granting up to 9 points for commenting, clarity, organization (decomposition into functions and functional use of data structures), and style. The total for this assignment, therefore, is 30 points.

2.4 Handin

Please email me your source file (nominally cachesim.c), and a Makefile so that all I need to do is type 'make' to build your cache simulator. If you need to convey any other information to me, you should also include a text file named README to make any additional comments/explanations.