CMPEN 431 - Project 1

Dr. Aasheesh Kolli
Penn State University

Solutions to project assignments are to be developed individually, without collaboration. However, as the projects in this class require the use of software tools and frameworks that students may have uneven prior familiarity with, discussion and assistance among students in gaining expertise with these software tools constitutes acceptable behavior. Note that **this assistance and discussion cannot include the sharing of access to any code produced in solution to the project assignments.** In order to avoid potential ambiguity in what constitutes "code produced in solution to the project assignment," students wishing to aid their peers with auxiliary supporting scripts, mechanisms, or examples are directed to pass any such artifacts to the course staff for vetting and possible inclusion on project-specific FAQs rather than share it with their peers directly.

This project has two sections. In the first section, you will be introduced to the *perf* command in Linux which uses hardware-provided performance counters to collect different performance characteristics in systems. In the second part, you will implement some simple functions in a cache simulator that calculates cache performance in different cache configurations.

For both of these sections, you may use CSE Linux Lab machines (i.e. csep204instXX.cse.psu.edu which XX is a 2-digit number from 01 to 40). We will test your results on those machines.

1. Perf

Intel processors provide different performance counters that programmers can use to collect different characteristics of any program in hardware. Some of the important performance counters that Intel processors provide are IPC (Instruction per Cycle), number of instructions, branches, cache hit/miss rate, memory references, etc. You can see different performance counters that are provided with the following command.

perf list

You can also look at the following link for more information regarding "perf".

`https://perf.wiki.kernel.org/index.php/Main_Page`

Imagine that you are targeting an instruction set architecture that does not have an integer division or modulus operation.

At first, you should write a function in C that takes two signed, 32-bit integers as arguments and returns a 2-element struct containing the quotient and remainder as 32-bit integer fields. For division by zero, return $\{0,0\}$. Please implement your function in *divide* function of p1.c. The running time of this function should not depend on the input and the given implementation of division algorithm in p1.c is not acceptable. In other words, you should implement the binary division by shift and subtract algorithm. You

may use the following link that explains this algorithm in "Integer division (unsigned) with remainder" section:

`https://en.wikipedia.org/wiki/Division_algorithm`

If you compile p1.c and run the created executable file, it will calculate the quotient and remainder of the division of two random numbers, for thousand times.

Now, you should use *perf* to determine the following statistics:

1. number of instructions
2. number of branches
3. IPC
4. number of memory accesses
5. L1I cache miss rate
6. L1D cache miss rate

Compilers use different optimization techniques to improve the performance of applications.
If you compile a C code with -O0 flag (gcc -O0 p1.c -o p1_unoptimized), it will compile without any optimization.
If you compile it with -O3 flag (gcc -O3 p1.c -o p1_optimized), compiler will try to apply different optimizations to improve the performance of your application. One way to see this difference is using hardware-provided performance counters from the output of *perf*.

2. Cache Simulation

In this section, you are going to complete some functions of a cache simulator. The provided cache simulator, simulates the performance of loads/stores of a combination of naive matrix multiplication algorithm and matrix transpose algorithm. The given code takes the dimensions of the matrix, number of iterations that two matrix algorithm are going to perform, and cache hierarchy configuration as an input. Then, it will simulates memory accesses, which are generated from all matrix multiplication and transpose iterations, through the given cache hierarchy and give some statistics such as cache hits/misses, evictions, writebacks, etc. Please look at the implementation of two matrix algorithms at the end of NMM-cachesim.c file.

Most of the functionality for this program has already been provided. However, certain key functions, which are needed to properly perform caching, are currently implemented as stub functions that either do nothing or return zero, causing the program to crash if they are relied upon. Your job will be to implement these missing functionalities within the functions defined in "YOURCODEHERE.c".

You will need/want to use the existing functions defined in csim.c, specifically "perform_access", and the cache structure defined in csim.h For this reason, you may want to familiarize yourself with the existing functions and fields in the other files, although you are not allowed to modify them.

Your project, once complete, will be able to correctly execute all tests invoked by "make test" as well as other cache and matrix configurations not present in the test list. Only cache hierarchies with monotonically non-decreasing block sizes (in integer multiples of 8-bytes) throughout the cache hierarchy will be tested. Similarly, only cache

hierarchies with monotonically non-decreasing capacity from upper to lower caches will be tested.

Ensure that your environment is correctly configured (e.g. with default gcc, etc.) by running "make test". You can verify correct initial state of your environment/files by noting the following:

1. The code should compile without any errors or warnings.
2. The first test case (no cache instantiated) should run to completion and match the output in the included copy of the output from running make test on a completed version of the program.
3. The second test case should quickly generate a segfault.

Modify YOURCODEHERE.c – this is the only file you will be modifying and turning in. Your project MUST compile without modification to the makefile, or any other source files. Your code will be recompiled against the other files in their original state, on CSE servers. Any reliance on additional modifications will likely result in non-compiling status under test and a zero for the project. Please ensure that any code you develop on a non-CSE platform works on the CSE servers, as the code is NOT GENERALLY PORTABLE.

There is missing code in each of the functions in YOURCODEHERE.c. Descriptions of the functionality of each function are in YOURCODEHERE.h.

Most of the function bodies can be written in 1-5 lines of normally-formatted C code, excepting the one that requires setting several variables in the cache structure.

Continue to test your project. All tests in "make test" should run to completion (expected total run time 1-2 minutes, mostly in last test). Matrix sizes $N \leq 8$ should match the provided output. Statistics for larger matrix sizes should be very similar but output may not be identical.

3. Deliverables

You should submit a zip file named Project1-{YOUR_CSE_ID}.zip (e.g. Project1-abc123.zip) that have the following files:

1. p1.c: This file should have the correct implementation of division algorithm with shift and subtract method. (20 points)
2. Report: In this file, you should report the output of perf from the 1^{st} section. Please compare your implementation with the given implementation of division algorithm according to 6 parameters that are introduced at the end Section 1 (i.e. number of instructions, number of branches, etc.). Moreover, explain how compiler optimizations affect different performance counters in the perf output. In addition, please report the perf command that you used to collect these results. (30 points)
3. YOURCODEHERE.c: This file should have the correct implementation of cache simulator functions. (50 points)