Memory Communication Coding (Evaluation)

CSCI 389: Computer Systems

Spring 2022

This assignment is designed to give you hands on experience working with the memory hierarchy. Feel free to collaborate wih others and use resources, but all code and writeups must be your own.

Due Date: Friday, March 18th at 10:00 am.

1. (30 points) **Benchmarking the Memory Hierarchy.** The main goal of this exercise is to empirically determine the sizes of the different cache layers in a processor by building a micro-benchmark that measures memory performance for different request sizes. It turns out it's not so easy to expose gaps in memory performance because the compiler and CPU work hard to hide these gaps from the application, so that memory accesses are as fast as possible in most normal circumstances. You will have to trick the CPU to reveal its secrets to us.

   (a) (18 points) **Benchmarking.** Write a program (in C or C++) that measures the latency of reading a byte from memory out of a buffer of N bytes, while varying N through different sizes. Your program should iterate on different buffer sizes (whatever makes sense to get a meaningful range). For each size N, it should loop many times to read a byte from a pre-allocated buffer of size N, and measure the latency (end-to-end time in nanoseconds) of this copy, then print it out. Your program's output should consist of two columns where the first column is the length of the buffer (N) in bytes, and the second column is the estimated mean time in nanoseconds (10-9s) to copy a single byte (not the entire buffer). Explain what your program is doing to get an an accurate measurement.

   Hints and Tips

   - Use the "-O3" compiler flag to ensure your compiler produces the fastest code it can. If you want to understand what it produces in the asm level, add the "-S -g" flags and inspect the resulting asm file.
   - Use the same mechanism as in Galaxy Explorers to time your code. A single memory read from L1 should take less than a nanosecond, which is immeasurable by the OS. Find a way to overcome this limitation.
   - Verify that you're timing memory reads and little else. Even under ideal experimental conditions, latency can vary by large amounts (the hardware, VM, and OS can both interrupt your loop mid-flight and add significant latency). Find a way to deal with this.
   - The hardware prefetcher will work hard to load values into L1 or L2 caches before you access them if it can infer your access pattern, even if your buffer is much larger than the cache. Think about how to foil the prefetcher.

   (b) (3 points) **Graphing.** Produce a graph showing the mean latency per memory read (in ns, on the Y-axis) vs. buffer size in KB (on the x-axis). Ensure that your graph is readable and useful.

   (c) (9 points) **Analysis.** Using the data you've collected, try to identify the sizes of your CPU's different caches. Explain your reasoning, as well as any abnormalities in your data.

   Look up which specific CPU you have in your test computer (hint: /proc/cpuinfo on Linux) and what its cache sizes are supposed to be. How do these numbers compare to your analysis? If there are deviations, what might have caused them?

2. (30 points) **Comparing Barriers.** The goal of this exercise is to explore the different forms of barriers and compare their performance.

(a) (6 points) **Implementation.** Write software (in C or C++) that implements the following types of barriers:

1. Centralized Barrier
2. Dissemination Barrier
3. MCS Barrier

Your barriers should take as input the number of threads that will synchronize upon creation. When running an instance of the barrier, you should take as input the thread ID of the thread calling the barrier function.

(b) (6 points) **Correctness Testing.** Prove that your barriers work correctly. This can come in the form of suite of tests (you should definitely at least some of these), or in analysis. Consider different arrival rates of threads, as well as pauses in your program. Summarize your proof of correctness.

(c) (6 points) **Performance Testing.** Write a program to compare the performance of your different barrier implementations. Your program should run multiple threads concurrently, where each thread repeatedly delays for a random amount of time before arriving at the barrier. You should test your program for a variety of thread numbers.

Hints and Tips

- Use the "-O3" compiler flag to ensure your compiler produces the fastest code it can. If you want to understand what it produces in the asm level, add the "-S -g" flags and inspect the resulting asm file.
- Use the same mechanism as in Galaxy Explorers to time your code. In expectation, your delays should not affect the relative performance of your barriers, but this may not hold in practice. Make sure you account for this when designing your tests.
- You'll need to create multiple threads in your program for this testing. The execution of this is relatively straightforward, but may lead to interesting bugs. Experiment and have fun.

(d) (3 points) **Graphing.** Produce a graph showing the mean overhead per synchronization (on the Y-axis) vs. number of threads (on the x-axis) for the different barrier types that you implemented. Ensure that your graph is readable and useful.

(e) (9 points) **Analysis.** Using the data you've collected, compare the performance of the different barrier types. Discuss the scalability of the different barriers, as well as which you would choose for what inputs.

What insights can you gain about the hardware and scheduler of the machine that you are running on from this data? Look up the number of cores and number of threads supported for your machine. Do your insights match the recorded data? If not, speculate about what caused the discrepancies.