CS 350c, Spring 2016, Laboratory 1
Memory Chase

Assigned: Thursday, February 4, 2016
Due: Thursday, February 18, 2016, by 10 am

1 Introduction

In this lab, you will learn about the memory latency of an x86-compatible microprocessor. By investigating use-after-read delays of a particular program, we can expose the latency of the memory system. We have provided a program that will help you understand the performance reduction when it is necessary to repeatedly refer to main memory instead of finding a desired value in a cache.

The memory-latency program you will use is provided as a part of this laboratory assignment. Below, we describe how to compile and use the resulting binary to observe the effects of repeatedly reading various amounts of memory. When you have completed the lab, you will hopefully have a better appreciation for the latency involved when many memory accesses can only be satisfied by accessing the main memory. As was mentioned in Laboratory 0, memory performance is the dominant factor in the performance of programs. It is important that you internalize how modern microprocessor memory systems effect the execution of your programs.

The more you understand about how a contemporary x86 microprocessor memory system functions, the better you will be able to take advantage of the mechanisms provided by the memory system. The challenge in this laboratory is to initialize the data to make the included "pointer-chasing" program execute as fast as possible and as slow as possible!

2 Logistics

You are expected to work on this lab alone. However, you may communicate with others concerning your understanding of C code and the various tools, e.g., the compiler, assembler, linker, loader, and other systems issues. The results that you submit in response to laboratory must be created and provided by you alone.

Any clarifications and revisions to the laboratory will be posted on the top-level course webpage.

There are many sources of information about x86 processors and their associated systems. We will make use of material that can be downloaded from the Web. We have provided a local copy of Agner Fog's information about the x86 family of processors. In addition, we have provided a local copy of Intel's x86 documentation (circa, Q4 2015). This x86-focused information is referenced on the top-level class webpage.

3 Handout Instructions

You will find the file `mem-chase.tar` referenced on the homework page of the class website. You will need to download this file so you can use its contents. There may be changes or updates, so please be sure to download the latest version – check the top-level class webpage for any updates or corrections.

You are only allowed to alter the file "mem-chase.h". This file contains two functions where you will provide C-language subprograms to initialize an array so that the program is defined in "mem-chase.c". For a variety of memory sizes, your initialization must make the program in "mem-chase.c" run both as fast a possible and as slow as possible. Included in "mem-chase.h" is a simple program that provides a straightforward initialization – just so as to provide an example of how an initialization function can be written – and although this initialization function works fairly well approximating the fastest way to arrange indices in the array, it certainly does not provide an initialization that provides the longest-running example. There is more discussion about this below.

4 Evaluation

You are expected to write a report that explains timing results of running the "mem-chase" code on a variety of dataset sizes. Running the program is straightforward; the hard part is creating initialization functions that makes the processor execute the included program as slowly as possible.

The maximum score for this laboratory is 100 points. The value of the individual components is as follows.

- Timing results from the various runs of the mem-chase program on different dataset sizes (20 points).
- A 3-D presentation of the timing results of each run (10 points). The various runs are described below.
- An explanation of the varying timing results data (20 points).
- An explanation of the code that you wrote that documents your approach and rationale for initializing the array (30 points).
- Answers to the two challenge questions that are listed below (10 points for each answer).

5 Running the Mem-Chase Code

Your task is to compile and run the mem-chase code. For the code provided, we recommend that you use a UTCS Department Linux system running on a computer with an X86 architecture; any of the UTCS Department machines are suitable. An x86-based laptop can also be used, but you will want a *gcc* or `LLVM` based C-language compiler. The mem-chase source files are contained in the `mem-chase.tar` file.

The program is contained in the files `clock.c`, `fcyc.c`, and `mem-chase.c`, along with three associated header files `clock.h`, `fcyc.h`, and `mem-chase.h`.

Below is a command sufficient to compile this program so you may run it; you only need to type:

```
   gcc -Wall -O2 -o mem-chase *.c
```

The resulting compiled program can be run by just typing:

```
    ./mem-chase <log_of_memory_size> <any_non-zero_natural_number>
```

This program will not produce any result – you need to measure the length of time it takes to run. Below are several examples of calls to the `mem-chase` program. The presence of a second argument – no matter what non-zero, natural-number characters are entered – indicates that our program should use the initialization that makes the program run slowly.

```
    ./mem-chase   9
    ./mem-chase  10
...
    ./mem-chase  27

    ./mem-chase   9 1
    ./mem-chase  10 1
...
    ./mem-chase  27 1
```

To complete your laboratory, you need to graph your results. In addition, you need to write down the model and speed of the microprocessor system that you use to get the requested results. You can find some of the information by typing the commands:

```
    cat /proc/cpuinfo
    cat /proc/meminfo
```

With that information, you can then lookup on Intel's website the configuration of the internal caches. You may find it somewhat difficult to exactly identify the processor in the system you are using, but on this point, don't be afraid to ask anyone for help.

You will spend much of your time for this laboratory studying the x86 memory system. The *trick* to making this program run slowly is to understand how the cache system works and make sure that each time the next memory location is *chased*, the value found is not in any cache. Note, there are many caches, not just the L1, L2, and L3 caches, but also there are also caches for other tasks. For instance, the TLB and its backing cache are used to rapidly translate addresses; you will want to make this cache also miss. And, there are other, smaller caches to deal with repeated translations, and so on. The more caches that your access pattern causes to miss, the slower your program will run.

Below are two questions about this laboratory. It is important that you give thoughtful answers to these questions (jointly worth 20%). Your answers need not be more than a few paragraphs, but your answers need to reveal clearly that you understand how the x86 memory hierarchy operates.

1. Compile the file `mem-chase.c` so as to produce x86 assembler. Is the code produced efficient? Approximately, what is the loop overhead as compared to the instructions that perform the *chasing*? Would it be valuable to increase the number of statements in the primary loop (at the bottom of the file)?

2. In the file `mem-chase.c`, in the *mem-chase* loop, there are four C-language statements identified by lines that contain "//W". Uncomment these instructions and attempt to improve (meaning, make the code run even more slowly) the initialization code for the `init_data_slow` so that the resulting program executes even more slowly. Think carefully about what changes you might make for various dataset sizes. What are the results of your investigation?

Hints

For the larger dataset sizes, it is relatively straightforward to initialize the memory so that the performance for the slow case is 2% to 3% of the fast case. Conversely, for the smaller dataset sizes, it's tricky to make any initialization that runs much slower than the fast case.

Exploring different initialization configurations may be valuable; a configuration that makes one particular size run slowly might not provide the slowest result at a substantially different memory size. So, the code you write for the `init_data_slow` may need to be parameterized by the input size provided to the initialization routines. This may also be true for the `init_data_fast`, but the differences will be much less dramatic.

If the initialization for the *slow* case(s) runs at less than 1% of your fast case(s), then you are surely understanding the memory system. When presenting your insights, please describe why your code runs so slowly.

Hand In Instructions

Please follow the instructions below for turning in your work.

- Make sure you have included your identifying information in your submission. In this case, you will update the file "mem-chase.h" and submit that file along with your report.

- To handin your mem-chase laboratory, prepare a report that you can submit as a PDF file. Name the file with your report `<YourUTID>.pdf`. The Canvas system identifies you in this manner.

  Submit your laboratory report by using CANVAS.