CSE 525 - Lab 5

Memory Performance and Virtualized Computing

Objective: This lab will teach concepts of memory usage optimization on computers with cache memory.

Environment: You will be using the Ubuntu 18.04 VirtualBox VM, and will be running an emulated version of the Raspberry Pi on QEMU. Follow the instructions here to launch your environment. You may need to enable Intel VT-x or AMD-V support in your system BIOS to ensure maximum performance of the hypervisor.

The VirtualBox image is available at
https://drive.google.com/open?id=1n1axt_uaP66VQxW61PQ48-NEoH7zjW48

1. Import your OVF file into Virtual box.
2. Click Start once imported to boot Ubuntu 18.04
3. See “Updates to Environment” - do this step only once
4. Right-click on the RPi folder on the desktop and type “Open in Terminal”
5. Type “./startpi.sh” and hit Enter to boot the ARM-based QEMU Raspbian image
6. Username and password are “pi” and “raspberry” as usual

Updates to Environment: Open your RPi folder on the desktop. Right-click on startpi.sh, and select “Open with Other Application”. Click “All Applications” and select Text Editor. Below the “#!/bin/sh” line, type in:

```
stty intr ^]
```

Then save the file and close the text editor. This will re-map the break command from Ctrl + C to Ctrl +]. That way, you can interrupt QEMU commands without stopping the entire emulator.

Compiling C program code:
Use a text editor such as vim or nano to edit your program code. Alternately, use SCP to copy it from your desktop computer. Using SSH/SCP is not covered in this document.

When you’re ready to compile, use the “-g” flag in your gcc command line when compiling C code. As an example, compile a program called program.c as follows:

```
gcc -Wall -Werror -pedantic-errors -g program.c -o runme
```

Debugging with GDB:

Type “gdb runme” to debug the executable code. See this handy cheat sheet to see what commands are available in GDB: https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

Exercise 1: This exercise highlights differences in row and column majority when it comes to cache memory implementation.

1. Write a C program that does a matrix summation, like we covered in class. (e.g., x = x + M[i][j] where M is a matrix and x contains the sum. Use nested FOR loops to perform the summation. The summation should be **row major**. Initialize the matrix to random data. Do this for matrix sizes of 20x20, 200x200, and 2000x2000. Use integer types.
2. Include some time benchmarking in your C code so that you can measure execution time **of the summation portion (the part in loops) only**. Research how to use time.h’s clock() function to do this.
3. Compile it as usual in gcc with debug options
4. Run the program and benchmark it with each size of matrix.
5. Rewrite the summation to make it column major, and note any execution time differences.
6. Did you see any differences? Why or why not?
7. Compile the code with the -S flag (to generate assembly) and view the assembly.
8. Did you see any differences?

Exercise 2: Disable compiler optimizations and try again.

1. Use -O0 (capital letter O and the number zero) to initially disable optimizations. 2. Run the experiments in Exercise 1 again. Do you notice any differences? 3. Compile the code with the -S flag (to generate assembly) and view the assembly. 4. Did you see any differences?

Exercise 3: Experiment with -floop-interchange

1. Read about -floop-interchange in https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/Optimize-Options.html
2. Enable the optimization and run Experiment 1 again.
3. Did anything change?
4. Compile the code with the -S flag (to generate assembly) and view the assembly.
5. Did you see any differences?

Exercise 4: Non-sequential access patterns

1. Re-write your access methods for Exercise 1 to load every 4th location in the matrix, but make sure that it can still wrap around and sum the remaining elements. In other words, make sure that matrix summation still works correctly.
2. Is the performance any different?

3. Re-write your access methods for Exercise 1 to use a **modulo 20 memory access pattern**. A modulo 20 pattern loads memory addresses that correspond to every 20th memory location. Ensure that you can still reach all the memory addresses in the array.

4. How did the performance change? Is there any difference between row and column majority? Based on your understanding of advanced memory systems, why is that the case?

Write your report in standard CSE 525 report format and submit to Blackboard by the due date.