The Department of Electrical and Computer Engineering

The University of Alabama in Huntsville

CPE 435 Lab-09

Code Testing & Performance Analysis and Energy Consumption

Introduction

You will use a few utilities that will help you debug and analyze your software. Please run all the experiments and write down your observations.

Part I: Valgrind

Valgrind is a tool suite that provides a number of debugging and profiling tools. You can make your program faster and more correct using Valgrind. Of the many tools available, we will use ***memcheck*** and ***cachegrind*** tools. Please visit the quick start tutorial at the official site.

Part I.a: Cachegrind

Cachegrind is a tool available in Valgrind that simulates how your program interacts with the machine's cache hierarchy and branch predictor. It simulates independent first-level caches: I and D-cache and unified second level cache. Please go through the tutorial at this link, and try to answer the following question:

> Most of the modern processors have three levels of cache. As presented above, cachegrind simulates two levels of caches. How does cachegrind handle the machines that have more than two levels of caches? Why does cachegrind do what it does for handling three levels of cache hierarchy? **[5+5]**

Assignment I:

You are given the following two codes. Please compile and run the codes. Please paste the demonstration run in your report. Write what each of the programs is doing. Please note the difference between the two programs. **[5]**

```
/*
File: test1.cpp
compile as g++ test1.cpp -o test1
*/
using namespace std;
#include <iostream>
main()
{
        int array[1000][1000];
        int i,j;
        for(i=0;i<1000;i++)
                for(j=0;j<1000;j++)
                        array[i][j]=0;
        cout << "array[0][0] was " << array[0][0] << endl;
}
```

```
/*
File: test2.cpp
compile as g++ test2.cpp -o test2
*/
using namespace std;
#include <iostream>
main()
{
        int array[1000][1000];
        int i,j;
        for(i=0;i<1000;i++)
                for(j=0;j<1000;j++)
                        array[j][i]=0;
        cout << “array[0][0] was " << array[0][0] << endl;
}
```

Assignment II:

Now run each of the programs with valgrind using the cachegrind tool. **Please grab the screenshot of the outputs using cachegrind where you should highlight the D1 cache misses for each of the programs**. You might want to run the programs as below in your command line. **[3]**

```
valgrind -v --tool=cachegrind ./test1
valgrind –v --tool=cachegrind ./test2
```

Do you see any difference in the numbers? Please answer the following questions based on your observations:

1. Which program is better in terms of performance? **[2]**
2. Why do you think one program is better? **Explain in detail [5]**

Part I.b memcheck

Memcheck is the default tool in valgrind. You may specify `--tool=memcheck` , but it is not needed. In our exercise, we will try to find memory leaks in our code exercise. Please go through the official documentation as this link to learn more on how to use this tool.

Valgrind allows you to run your program in Valgrind's own environment that monitors memory usage such as calls to malloc and free (or new and delete in C++). If you use uninitialized memory, write off the end of an array, or forget to free a pointer, Valgrind can detect it. Since these are particularly common problems, this tutorial will focus mainly on using Valgrind to find these types of simple memory problems, though Valgrind is a tool that can do a lot more.

Assignment III:

You are given the following codes. Please compile and run them. Please write what is the problem with **each** of the codes. **[3]**

```
//file: test 3 compile as test3
#include <stdlib.h>
int main()
{
        char *x = (char*)malloc(100); /* or, in C++, "char *x = new char[100] */
        return 0;
}
```

```
//file: test4 compile as test4
#include <stdlib.h>
int main()
{
        char *x = (char*)malloc(10);
        x[10] = 'a';
        return 0;
}
```

```
//file: test5 compile as test5
#include <stdio.h>
int main()
{
        int x;
        if(x == 0)
        {
                printf("X is zero"); /* replace with cout and include iostream for C++ */
        }
        return 0;
}
```

Assignment IV:

Run each of the following programs with valgrind so that you will test the memory leaks. You might want to use the following commands in the terminal to run them. Please grab screenshots **[3]** and explain each of them with focus on error/s indicated for **each** of the programs. **[7]**

`valgrind --tool=memcheck --leak-check=yes ./test3`

`valgrind --tool=memcheck --leak check=yes ./test4`

`valgrind --tool=memcheck --leak-check=yes ./test5`

Part II: Power Consumption measurement

The next section of this lab involves using a power measurement tool to determine which is the best sorting algorithm based on power consumption and work performed. If you are running linux natively, you should use the RAPL model. If you are using a VM on a Windows host, it is recommended to use the Hardware Monitor Pro application. If you are using a MAC, it is recommended to use the Activity Monitor. **You will only need to use one of the tools** for this part of the lab.

Using RAPL model (Native Linux)

Running Average Power Limit (RAPL) is one of the energy modeling techniques that can be used in some supported CPUs platforms. Intel introduced RAPL first in the Intel Sandy Bridge architecture and it continues in successive generation architectures like the Haswell and Ivy Bridge. There is a set of hardware counters that can be accessed, known as Model-Specific Registers (MSRs).

There are two benefits of using the RAPL model: it can be used to profile the real-time power consumption or to control it by passing the corresponding parameters. In general, there are multiple benefits of using the RAPL; hardware counters are updated without software intervention. Also, these registers are updated automatically. However, these registers are just 32-bit length so it is possible that the registers will overflow after a certain amount of time after starting the machine.

Using Hardware Monitor Pro (Virtual Machines)

You may download and install the Hardware Monitor Pro from the following link. Please only use the trial version of the Hardware Monitor Pro for the lab. The Pro version will allow you to log power consumption over a period of time.

To log power consumption, select Tools>Logs>Start Recording. Once your program has finished, end the recording with Tools>Logs>Stop Recording. A new window should appear with the current logs. Navigate from this window to the subfolder containing the logs in graphical form. The package power graph is what you will need to reference for your power measurements. Additionally, under Tools>Options you can also save a corresponding csv file that those graphs are based on.

If you are using this method, you will need to log your host machine's power consumption normally, to act as a baseline when you log power consumption during program execution.

Using Activity Monitor (for MACs)

If you are using a MAC, you make use of the Activity Monitor. You may refer to this link to help get started. **If you are using this method, you will need to record your host machine's idle power consumption before running any programs to act as a baseline.**

Assignment V:

Write a function or cite a source for each of the following three sorting algorithms. If you are running linux natively, you will edit a source file provided (*rapl_read*.c) at around line **809** to make a call to each of these functions separately (one at a time).

Quick Sort
Merge Sort
Insertion Sort

You can create a top level function e.g. *quick_sort_top_level*() that can be called from the location specified in *rapl_read*.c if you are running linux natively or from main if you are using the other methods. Your top level function should accept a long integer *n* as an argument, generate *n* integer elements randomly and call the actual sorting function. The top level function should also call another function *sort_verify*() that verifies if the sorting is done correctly (you can iterate through all the items of the sorted array to see if every *i+1*th item is greater or equal to *i*th item). You will also need to include some means of timing program execution, preferably something similar to the following:

```
#include <stdio.h>
#include <stdlib.h>

#define TIMER_CLEAR  (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0)
#define TIMER_START  gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   (double) (tv2.tv_usec-tv1.tv_usec)/1000000.0+(tv2.tv_sec-tv1.tv_sec)
#define TIMER_STOP    gettimeofday(&tv2, (struct timezone*)0)
struct timeval tv1,tv2;

int main(int argc, char* argv[])
{
    int i;
    TIMER_CLEAR;
    TIMER_START;
    // Call to function that you want to time
    // example
    sleep(5); //Timer will give us ~5 seconds
    TIMER_STOP;
    printf("Time elapsed = %f seconds\n", TIMER_ELAPSED);
    return 0;
}
```

If you are using either the Hardware Monitor Pro or Activity Monitor, you will need this timing to convert the recorded information from Watts into Joules (Watts X Second). Remember you will also need to measure a baseline power consumption for your CPU when idle (typically this is package power if you are using Hardware Monitor). This baseline should be subtracted from the average power consumed.

You should time your programs to run for around 5 seconds where possible. You will need to set the number of elements for each of your sorting algorithms. The number of elements may be different for each algorithm.For some of these algorithms that may require tens of millions of elements. You cannot allocate arrays of this size on the stack, you must allocate them in the global area or the heap. Algorithms obtained from the web may incur segmentation faults because of this issue. You may consider the following:

```
#include <sdtdio.h>
int arraya[1000000];                    /* this is ok it is from global memory */
int main()
{
int arrayb[1000000];                    /*this is not ok it is from the stack and will fail with
a segmentation error for large enough arrays */
int *arrayc = malloc(1000000);          /* this is ok it is from the heap */
}
```

For the following sections you will be running your algorithms 5 times and taking the average of their results to come up with the average energy and time spent per array element. You will include a sample run's output for each set of 5 runs. The data may be summarized as follows:

Trial	Algorithm					
	Insertionsort		Mergesort		Quicksort	
	Energy	Time	Energy	Time	Energy	Time
1						
2						
3						
4						
5						
Average Energy						
Energy per element						
Average Time						
Time per element						

Note that the number of elements will also need to be listed in the report.

Make sure to make proper use of allocating and deleting the elements of the array. **[20]**

Assignment VI:

Your program should generate at least 1M elements of an array (for insertionsort it may be less, say around 10000) and run for around 5 seconds. Please use pointer for dynamic array using new/delete or malloc/free rather than using static array. Compile and run code with each of the algorithms 5 times and report all of the values of energy consumption in Joules as well as the time taken for each run. Find the average values of the five runs for both energy consumption and execution time and derive the average energy consumption and time per element for each algorithm. Please make sure that you have turned off any compiler optimization (i.e. Please use -O0 flag while compiling) **[15]**

Assignment VII:

Perform the same set of experiments as in Assignment VI, but with optimization flag -O3. Run each of the sorting algorithms 5 times and report the average as the final output. **[15]**

Assignment VIII:

Which algorithm was the best in terms of energy consumption per element? Which algorithm was the best in terms of time per element? How does this compare to the time complexity of each of the algorithms? Did the compiler flag make things any better? **[12]**

Extra Credit Research:

1. Please research and discuss the MSR registers. **[10]**

Deliverables

Lab Report

The following material in each section is expected:

1. Cover page with your name, lab number, course name, and dates
2. Observations and Answers
    a. Please include answers to any questions from the lab document, as well as any necessary supporting documentation in the order they appear.
3. Conclusion
4. Appendix
    a. Source code for programs used in part 2

The report should be submitted as a single pdf document with the source code for your program within it.

Recorded Demonstration

The following material in each section is expected:

1. Introduce yourself and give the name of the lab
2. Walk through the second part of the lab and discuss your results. You do not need to describe the sorting algorithms used, but you must show that they work. Programs in part 2 must be shown to compile and run. Show how you recorded any measurements for each algorithm at least once and what conclusions you can make about the efficiency of the algorithms used.

The demonstration may be in person or recorded and submitted as an mp4 file alongside the report.