**CSE-443/543: High Performance Computing**

**Homework #01**

Max Points: 55

**Objective**: The objective of this exercise is to:

* Learn how to use Profile Guided Optimization (PGO) to improve the performance of serial code
* Continue to build familiarity with the Linux perf hardware-assisted profiler.
* Build familiarity with statistical analysis, in particular Student’s T test.
* Learn how to use the Linux [ulimit](https://linux.die.net/man/1/bash) command, which is built into bash, to change system limits

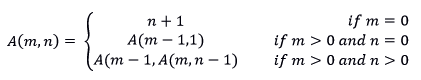
Fill in answers to all of the questions in this document and the corresponding Google spreadsheet. You may discuss the questions with your instructor and the TA.

## Background

In class, we learned about Profile Guided Optimization (PGO) as a technique to use statistics gathered during trial runs of a program to optimize it. See <https://en.wikipedia.org/wiki/Profile-guided_optimization> for more information about PGO. In PGO, you compile a program with a flag that makes the program capture different kinds of performance information when the program runs. This is called *instrumentation*, and the executable produced by this compilation is called an *instrumented program*. When the instrumented program is executed, performance information is captured in a file. After several instrumentation runs, a second compilation occurs, this time using the results the instrumented program produced to inform the compiler about opportunities to reorganize instructions, unroll loops, restructure conditional statements, etc.

For gcc and g++, the compiler flag to enable instrumentation is -fprofile-generate and the flag to use the instrumentation data is -fprofile-use.

In this homework you will be applying PGO to a simple, highly recursive, function, the Ackermann function (<https://en.wikipedia.org/wiki/Ackermann_function>) defined as:



Ackermann’s function was chosen because its highly recursive nature provides a short yet scalable benchmark involving millions of method calls. Its stack-based approach provides excellent locality of reference enabling very efficient use of instruction and data caches and minimizing the influence of other variables on the machine. You will be computing A(m,n) for m=3 and for various values of n.

In previous labs, you used the Linux perf hardware-assisted profiler to compare the performance of different codes. In this homework, you will again profile code using perf with the goal of comparing the pre- and post-PGO performance of the code and identifying which perf measures are improved when applying PGO to this algorithm.

You have previously applied [Student’s T-test](https://en.wikipedia.org/wiki/Student%27s_t-test) to check whether the difference between benchmark results were statistically significant. You will be applying this test in this experiment to determine the statistical significance of the performance improvement and differences in perf stat metrics. In our case, we will need to use the “Equal or unequal sample sizes, unequal variances” version of the test. If is the pre-optimization average value, its standard error, is the post-optimization average value, and its standard error, then the T statistic can be computed as:

Because the Ackermann function is recursive, for large arguments it can run out of [stack memory](https://en.wikipedia.org/wiki/Stack-based_memory_allocation). Recall from your Systems courses that the stack is memory where the variables that functions use are stored. When the function exits, this memory is automatically released. For a recursive function, however, the memory allocated by all function calls will persist until the function begins retreating from its deepest recursion. Even if the memory used by the function is small, if there are enough calls to a function the stack can be exhausted, resulting in abnormal termination of the program.

Many versions of Linux set the default stack size at 8 MB (8192 KB). That is not sufficient for the jobs you will be running for this homework. The bash built-in command ulimit permits you to view defined system limits, like the stack size, and, in some cases, to change them. Try running ulimit -a to see a set of system limits and their default values. Note that the stack size is set to 8192. If you issue the command ulimit -s, ulimit will display the current value of the stack size in its native units, in this case kbytes) . If you specify a numeric argument to ulimit -s, the stack size will be set to that value. It’s possible to estimate how much stack is required, but a simpler approach is to set the stack to the largest amount available which is accomplished via the command ulimit -s unlimited. Try issuing the ulimit -s command, or ulimit -a, before and after using ulimit -s unlimited to see the impact.

## Experiment

You should download the file ackermann.cpp from Canvas, then create a bash script called Homework01.bash that accomplishes the following:

1. Invokes bash as a login shell. You should know how to do this by now…
2. Captures the information required to complete the Apparatus section below. You should know how to do this by now…
3. Compiles the code using these command-line options:

-O3 -g -Wall -std=c++17 –o ackermann

1. Sets the stack size to unlimited via the ulimit command.
2. Runs the resulting executable 6 times for n=16 (./ackermann 16), profiling it with perf. You should capture at least the events requested in the Observations tables in the Homework 01 – results Google spreadsheet. This establishes the baseline (“pre-PGO”) execution statistics. **Note: As a sanity check, the correct value of A(3,16) is 524285.**
3. Compiles the code using these command-line options to enable PGO instrumentation:

-O3 -g -Wall -std=c++17 –fprofile-generate –o ackermann\_pgo

1. Runs the resulting executable (ackermann\_pgo) 4 times once with the argument 9, then with 11, 13 and 15. Note that a file called ackermann.gcda will be created. This file contains the instrumentation data the compiler will use for optimization.
2. Compiles the code using these command-line options to use the instrumentation data:

-O3 -g -Wall -std=c++17emacs –fprofile-use –o ackermann\_pgo

1. Runs the resulting executable 6 times for n=16 (./ackermann\_pgo 16), profiling it with perf. You should capture at least the events requested in the Observations tables in the Homework 01 – results Google spreadsheet. This establishes the post-optimization execution statistics.

You may want to consider running your script with nohup or capturing the standard error of the perf command to files to keep your data organized better. You have redirected and appended standard error to a file in previous labs.

# Apparatus

The experiment documented in this report was conducted on the following platform (fill in the two lines of the Details column using information determined in your shell script):

|  |  |
| --- | --- |
| Component | Details |
| CPU Model | model name : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz |
| Main Memory (RAM) size | MemTotal: 8056076 kB |

# Observations

Record the elapsed time, and the numbers of branches, branch-misses, L1-dcache-loads and L1-dcache-load-misses in the tables in the Homework 01 – results Google spreadsheet.

# Analysis

Complete the Analysis tables in the Homework 01 – results Google spreadsheet using data from your Observations tables.

# Discussion

You should write up a summary of your observations using timing data to quantify the performance improvement you saw after using the PGO technique. Based on your T statistic, is this a statistically significant improvement? Are the changes in the other perf stat events observed statistically significant? Quantify your answers with your results.

Then, using counter data from perf stat, identify which events have changes commensurate with the performance improvement and which are the one or ones most likely to be the cause of it. Note that the elapsed time is a result of the improvement, not a cause. You will be graded on the completeness of your discussion and your use of quantitative measures, so please take time to understand the results you obtained and how you can explain them.

|  |
| --- |
| Several of the perf stat events were significantly different after using the PGO technique. After calculating the T statistic for elapsed time, branches, branch misses, L1-dcache-loads, and L1-dcache-load-misses, we can see that elapsed time, branches, and branch misses are significantly different. The T stats we get are 166.99 for elapsed time, 545.48 for branches, and -60.58 for branch misses. As for L1-dcache-loads and L1-dcache-load-misses, we see very small T stats, -0.38 and 0.4 respectively. Of the significant changes, the only one that is not significantly better is branch misses. The number of branches is far less, which is likely the primary reason that the elapsed time went down by over 20 seconds on each run. The post-PGO program did fewer instructions, which obviously resulted in less time. The number of branch misses has gone up, however our elapsed time is still much better than it was pre-PGO, so it is acceptable. The amount of memory (L1-dcache-loads and L1-dcache-load-misses) used for both pre-PGO and post-PGO was about the same, so any difference in performance cannot be attributed to this. The PGO technique was able to optimize our code to run much better than it did previously. |

# Submit files to Canvas

When you complete the homework, download this document from Google Drive as a Microsoft Word (.docx) file with the naming convention Homework01 - MUid.docx (example: Homework01 - ferrenam.docx).  You should save the corresponding Google spreadsheet file as a Microsoft Excel (.xlsx) file with the naming convention Homework01 – Results MUid.xlsx (example: Homework01 - Results - ferrenam.xlsx)

Then, submit the following files to Canvas:

1. The Microsoft Word file you downloaded from Google Drive.
2. The Microsoft Excel file you downloaded from Google Drive.
3. The Homework01.bash shell script you created for this homework.