CMPSCI 630 Systems **Fall 2019**

Lecture 18

Lecturer: Emery Berger *Scribe: Sahil Mishra*

18.1 Profilers

The class began with a review from the previous class which covered profiling.

What is gprof?

gprof is a profiling program that uses statistical sampling to periodically interrupt programs and extrapolate the distribution of execution times if there are enough samples.

Profilers, today, in general, are not very useful. Profiling was easy back in the day! Most profilers were created in the era of single-threaded programs.

Multi-threaded programs can run slower even if some parts are made to run faster. This is due to contention - some parts may be accessing the same resources leading to multiple locks. This can also lead to ”accidental” Denial-of-Service attacks.

Here’s an example. Let’s say you are given two CPUs; CPU **A** with a clock speed of 3GHz and CPU **B** with a clock speed of 1GHz. Would you be able to correctly predict which one is faster?

The answer is that we don’t know. Actually, there’s a lot we don’t know. In order to correctly answer our question we first need to first answer a few other questions: How big is the cache? How many registers do they have? How big is the TLB? What kind of bus are they connected to? How fast is the RAM?

Unfortunately, a generic profiler does not answer any of these questios. Hence, **performance prediction** is extremely difficult! It has been shown that you could a run a program on your machine on a Monday and it may run 20/30% faster/slower the next day. This is due to variety of reasons. Prior to running the program, your machine will first load the program into memory and load the environment variables as well. Environment length can change depending upon a variety of variables like time, date, current working directory and username. Hence, some environment variables may fit in the cache and some may not. This may result in conflict/capacity misses. As a result, programs may take up to 30% longer due to cache misses.

Modern on-chip hardware have in-built performance counters. They provide the number of cache misses and branch prediction misses.

Another problem with most Profilers is that most of them only inform which thread/function takes the longest. But where a program spend most of its time is not necessarily where one must optimize. For example, a numpy function may take a lot of time to run within a Python program even though numpy functions are highly optimized. Instead, a Profiler must be able to inform the programmer what to optimize.

18.2 Performance Evaluation

Performance evaluation is hard! Typical performance evaluation involves running two different versions of a program just once. This is not good enough because there is variation due to cache misses, context stitches, etc. The speed of execution of a program depends heavily on its layout. Even a malloc can change the layout of program. Layout is brittle; it is predictable but can break very easily.

In order the remove the effect of layout performance evaluation is done using a Stabilizer. A Stabilizer completely randomized the layout before running a program, thus, removing the bias. Usually, a Stabilizer generates a new random layout every 1/2 second.

Here's a fallacy; where a programs spends most of its time *does not* imply where you should optimize! Sometime profilers give non-actionable actions. For example, a profiler may state that a numpy function or a system (Linux) level function is taking the most time. Well, a programmer can't do much about it. Hence, a *good* profiler must be able to tell where we should optimize!

18.3 Performance Analysis

In order the correctly analyse the performance of a program, one must conduct a null hypothesis significance testing. Basically, we are trying to answer if $A^{'} > A$, what is the probability of measuring a speed up this large by chance? If there is a low probability, then speed up is real.