COMPSCI 630 Systems **Spring 2017**

Lecture 16: Performance Analysis

Lecturer: Emery Berger *Scribe(s): Pankaj Bhambhani,Anish Pimpley*

16.1 The need for performance optimization

Back in the 80s, although computers were slow, performance was close to expected. There wasn't much focus on optimizing code, it was expected that purchasing newer hardware would solve performance issues. This is because the newer hardware was generally considerably faster, on an average upto 10X faster. Moore's law predicted that this would continue for a period of time. Today, however, there is no more an increase of such magnitude in hardware power, since making bigger processors adds more heat which would burn your pocket if you had that processor in your mobile phone. Hence, with hardware no longer supporting the expected increase in performance, code optimization becomes imperative to achieve higher speeds.

16.2 Effect of layout on performance; Stabilizer

Memory layout affects performance - changing a program by moving code around will change its layout, and there is way to measure the effect of this change in isolation. For e.g., the first time we run a program, it is loaded off the disk. If we run it again immediately, it would be pre-loaded in memory, so the runtime would be considerably faster. Thus execution time can different due to such non-deterministic factors such as loading a program off memory as well as other factors such as time-slicing/scheduling. It is therefore recommended, while measuring performance statistics, to run a program a few times (e.g. 20 or 30) as opposed to running it just once.

Another way code layout can affect the performance is if the code lines map to the same cache set, then there is a high possibility of conflict misses which can affect the running speed. This is also true for branch predictors and TLBs, as is true for any application where some sort of hashing of memory addresses is involved. Even common things such as a change of username or a change in the working directory can cause a change in layout, affecting performance

In short, measuring statistics by running the program on just 1 memory layout/configuration is equivalent to generating a sample by polling just 1 person. Layout is brittle and cause measurement bias. Stabilizer is a program which attempts to eliminate the effect of layout on performance by randomizing the layout during execution of the program. This includes randomizing function addresses, stack frame sizes, heap allocations, among other things, and doing it over and over again.

Also, proper (unbiased) statistical testing requires forgetting the eyeball test and instead testing for the absence of null hypothesis - this involves checking the *p-value* (probability value) after running the tests. If the results are higher than the p-value, then its highly unlikely that they're random. The choice for a distribution is that of a Normal one, owing tot he central limit theorem. The procedure for this is to first build benchmarks with stabilizer and then evaluate the performance on each benchmark across the entire suite. Stabilizer generates a new random layout every 0.5 second, to avoid bias, and the total execution time is the sum of all time periods.

16.3 coz - Causal Profiling

As mentioned in the above section, we are now at a point where both hardware and compilers arent getting faster at a significant rate. In such a case, code optimization is the the only remaining avenue for improving performance significantly.

Profilers serve as a vital tool for identifying critical areas to focus for optimization. Gprof is one such profiler. Traditional profilers keep track of number of runs, and the amount of time spent on working on each function and its descendants. The goal is to locate frequently executed code that also takes long to run. However, the frequently run or most time consuming criteria might not be the ideal objective to pursue. For example optimizing the code that draws the loading screen animation as an app opens, will not make the program run faster, yet satisfies both of the aforementioned criteria. Gprof also struggles with parallel programs, both due to the way we implement locks and a basic methodological conflict.

Coz is a profiler developed at UMass Amherst by Charlie Curtsinger and Emery Berger, that takes a different approach to profiling named Causal Profiling. Causal profiling precisely identifies where optimization efforts should be focused and predicts the impact of the aforementioned optimization effort. Coz does this by running performance experiments to establish cause and effect.

Hypothetically, these experiments can be run by individually speeding up specific portions of the program, and calculating the effective speedup because of this change. In reality however, such speedups that are necessary to run performance experiments cannot be achieved. As a work around, Coz uses virtual speedup. Here, instead of speeding up one program, we slow everything else down while keeping the code that we wish to speed up, unchanged. By changing the extent to which everything is slowed down, we get a precise simulation of what the speedup for the untouched portion would be, over multiple readings of virtual speed up. This is done by putting progress points which add latency, measuring the throughput at that point, and then using Little's law $W = \frac{L}{\lambda}$. Thus, Coz provides us with a precise graph for every code block plotting the individual speed up of the clock versus effective speedup.