ECE459: Programming for Performance Winter 2011

Lecture 3 — January 11, 2011

Patrick Lam *version 1*

Profiling

If you want to make your programs or systems fast, you need to find out what is currently slow and improve it. (duh!)

How profiling works:

- sampling-based (traditional): every so often, query the system state; or,
- instrumentation-based, or probe-based/predicate-based (traditionally too expensive) : query system state under certain conditions, like conditional breakpoints.

We'll talk about both per-process profiling and system-wide profiling.

If you need your system to run fast, you need to start profiling and benchmarking as soon as you can run the system. Benefits:

- establishes a baseline performance for the system;
- allows you to measure impacts of changes and further system development;
- allows you to re-design the system before it's too late;
- avoids the need for "perf spray" to make the system faster, since that spray is often made of "unobtainium"[1].

You are looking for abnormalities; in particular, you're looking for deviations from the following rules:

- time is spent in the right part of the system/program;
- time is not spent in error-handling, noncritical code, or exceptional cases; and
- time is not unnecessarily spent in the operating system.

For instance, "why is `ps` taking up all my cycles?"; see page 34 of Cantrill[2].

[1] `http://en.wikipedia.org/wiki/Unobtainium`
[2] `http://queue.acm.org/detail.cfm?id=1117401`

Development vs. production

You can always profile your systems in development, but that might not help with complexities in production. (You want separate dev and production systems, of course!) We'll talk a bit about DTrace, which is one way of profiling a production system. The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

Userspace per-process profiling

Sometimes—or, in this course, often—you can get away with investigating just one process and get useful results about that process's behaviour. We'll first talk about `gprof`, the GNU profiler tool[3], and then continue with other tools.

`gprof` does sampling-based profiling for single processes: it requests that the operating system interrupt the process being profiled at regular time intervals and figures out which procedure is currently running. It also collects information about which procedures call other procedures. There also used to be a Java profiler which collected the same information, but it's obsolete now.

"Flat" profile. The obvious thing to do with the profile information is to just print it out. You get a list of procedures called and the amount of time spent in each of these procedures.

The general limitation is that procedures that don't run for long enough won't show up in the profile. (There's a caveat: if the function was compiled for profiling, then it will show up anyway, but you won't find out about how long it executed for).

"Call graph". `gprof` can also print out its version of a call graph, which shows the amount of time that either a function runs (as in the "flat" profile) as well as the amount of time that the callees of the function run. Another term for such a call graph is a "dynamic call graph", since it tracks the dynamic behaviour of the program. Using the `gprof` call graph, you can find out who is responsible for calling the functions that take a long time.

Limitations of `gprof`. Beyond the usual limitations of a process-oriented profiler, `gprof` also suffers limitations from running completely in user-space. That is, it has no access to information about system calls, including time spent doing I/O. It also doesn't know anything about the CPU's built-in counters (e.g. cache miss counts, etc). Like the other profilers, it causes overhead when it's running, but the overhead isn't too large.

Alternative to `gprof`. You would probably actually use the Google CPU profiler:

`http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html`

Documentation is sparse, but it seems to be like `gprof`, except that it supposedly works for multi-threaded programs, and produces niftier (potentially graphical) output.

[3] `http://sourceware.org/binutils/docs/gprof/`

System-level profiling

Most profiling tools interrogate the CPU in more detail than `gprof` and friends. These tools are typically aware of the whole system, but may focus on one application, and may have both per-process and system-wide modes. We'll discuss a couple of these tools here, highlighting conceptual differences between these applications.

Solaris Studio Performance Analyzer. This tool[4] supports `gprof`-style profiling ("clock-based profiling") as well as kernel-level profiling through DTrace (described later). At process level, it collects more process-level data than `gprof`, including page fault times and wait times. It also can read CPU performance counters (e.g. the number of executed floating point adds and multiplies). As a Sun application, it also works with Java programs.

Since locks and concurrency are important, modern tools, including the Studio Performance Analyzer, can track the amount of time spent waiting for locks, as well as statistics about MPI message passing. More on lock waits below, when we talk about WAIT.

VTune. Intel and AMD both provide profiling tools; Intel's VTune tool costs money, while AMD's CodeAnalyst tool is free software.

Intel uses the term "event-based sampling" to refer to sampling which fires after a certain number of CPU events occur, and "time-based sampling" to refer to the `gprof`-style sampling (e.g. every 100ms). VTune can also correlate the behaviour of the counters with other system events (like disk workload). Both of these sampling modes also include the behaviour of the operating system and I/O in their counts.

VTune also supports an instrumentation-based profiling approach, which measures time spent in each procedure (same type of data as `gprof`, but using a different collection scheme).

VTune will also tell you what it thinks the top problems with your software are. However, if you want to understand what it's saying, you do actually need to understand the architecture.

Something Totally Different

Here's a short tangent. Many of the concepts that we've seen for code also apply to web pages. Google's Page Speed tool[5], in conjunction with Firebug, helps profile web pages, and provides suggestions on how to make your web pages faster. Note that Page Speed includes improvements for the web page's design, e.g. not requiring multiple DNS lookups, as well as traditional profiling for the JavaScript on your pages.

[4] You can find a high-level description at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/oss-performance-tools-183986.pdf`

[5] `http://code.google.com/speed/page-speed/`