ECE459: Programming for Performance Winter 2011

Lecture 4 — January 13, 2011

Patrick Lam *version 1*

Today we'll talk about more profiling tools, including some modern tools that have recently appeared in the research literature. First, I'll talk about two more standard tools.

By the way, here is a good reference on profilers in general:

`http://ait.web.psi.ch/services/linux/hpc/hpc_user_cookbook/tools/profilers/index.html`

CodeAnalyst. AMD also provides a profiling tool. Unlike Intel's tool, AMD's tool is free software (the Linux version is released under the GPL), so that, for instance, Mozilla suggests that people include CodeAnalyst profiling data when reporting Firefox performance problems [1].

CodeAnalyst is a system-wide profiler. It supports drilling down into particular programs and libraries; the only disadvantage of being system-wide is that the process you're interested in has to execute often enough to show up in the profile. It also uses debug symbols to provide meaningful names; these symbols are potentially supplied over the Internet.

Like all profilers, it includes a sampling mode, which it calls "Time-based profiling" (TBP). This mode works on all processors. The other modes are "Event-based profiling" (EBP) and "Instruction-based sampling" (IBS); these modes use hardware performance counters.

AMD's CodeAnalyst documentation points out that your sampling interval needs to be sufficiently high to capture useful data, and that you need to take samples for enough time. The default sampling rate is once every millisecond, and they suggest that programs should run for at least 15 seconds to get meaningful data.

The EBP mode works like VTune's event-based sampling: after a certain number of CPU events occur, the profiler records the system state. That way, it knows where e.g. all the cache misses are occuring. A caveat, though, is that EBP can't exactly identify the guilty statement, because of "skid": in the presence of out-of-order execution, guilt gets spread to the adjacent instructions.

To improve the accuracy of the profile information, CodeAnalyst uses AMD hardware features to watch specific x86 instructions and "ops", their associated backend instructions. This is the IBS mode[2] of CodeAnalyst. AMD provides an example[3] where IBS tracks down the exact instruction responsible for data translation lookaside buffer (DTLB) misses, while EBP indicates four potential guilty instructions.

oprofile. The Linux version of CodeAnalyst builds upon the free `oprofile` tool, which is a standard system-wide profiler. `oprofile` includes a Linux kernel module (which manipulates hardware

[1] `https://developer.mozilla.org/Profiling_with_AMD_CodeAnalyst`

[2] Available on AMD processors as of the K10 family—typically manufactured in 2007+; see `http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf`. Thanks to Jonathan Thomas for pointing this out.

[3] `http://developer.amd.com/cpu/CodeAnalyst/assets/ISPASS2010_IBS_CA_abstract.pdf`

counters) and utilities to control profiling. It also includes hooks to attribute time to code emitted by just-in-time engines, e.g. Java.

Shark. Shark [4] is Apple's systemwide profiler for Mac OS X. It also provides hints about what it thinks you can optimize or vectorize. You can consult a Shark tutorial here:

`http://developer.apple.com/tools/shark_optimize.html`

Moving away from traditional profiling tools, I'm going to talk about DTrace and WAIT, two newer tools for understanding system performance.

DTrace

DTrace[5][CSL04] is an instrumentation-based system-wide profiling tool designed to be used on production systems. It supports custom queries about system behaviour: when you are debugging system performance, you can collect all sorts of data about what the system is doing. The two primary design goals were in support of use in production: 1) avoid overhead when not tracing and 2) guarantee safety (i.e. DTrace can never cause crashes).

DTrace runs on Solaris and some BSDs. There is a Linux port, but it doesn't seem to be finished.

Probe effect. "Wait! Don't 'instrumentation-based' and 'production systems' not go together?"

Nope! DTrace was designed to have zero overhead when inactive. It does this by dynamically rewriting the code to insert instrumentation when requested. So, if you want to instrument all calls to the `open` system call, then DTrace is going to replace the instruction at the beginning of `open` with an unconditional branch to the instrumentation code, execute the profiling code, then return to your code. Otherwise, the code runs exactly as if you weren't looking.

Safety. As I've mentioned before, crashing a production system is a big no-no. DTrace is therefore designed to never cause a system crash. How? The instrumentation you write for DTrace must conform to fairly strict constraints.

DTrace system design. The DTrace framework supports instrumentation *providers*, which make *probes* (i.e. instrumentation points) available; and *consumers*, which enable probes as appropriate. Examples of probes include system calls, arbitrary kernel functions, and locking actions. DTrace also supports typical sampling-based profiling in the form of timer-based probes; that is, it executes instrumentation every 100ms. This is tantamount to sampling.

You can specify a DTrace clause using probes, predicates, and a set of action statements. The action statements execute when the condition specified by the probe holds and the predicate evaluates to true. D programs consist of a sequence of clauses.

[4] `http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/shark.1.html`

[5] `http://queue.acm.org/detail.cfm?id=1117401`

Example. Here's an example of a DTrace query from [CSL04].

```
syscall::read:entry {
        self->t = timestamp;
}

syscall::read:return
/self->t/ {
        printf("%d/%d spent %d nsecs in read\n"
            pid, tid, timestamp - self->t);
}
```

The first clause instruments all entries to the system call `read` and sets a thread-local variable `t` to the current time. The second clause instruments returns from `read` where the thread-local variable `t` is non-zero, calling `printf` to print out the relevant data.

The D language design ensures that clauses cannot loop indefinitely (since they can't loop at all), nor can they execute unsafe code; providers are responsible for providing safety guarantees. Probes might be unsafe because they might interrupt the system at a critical time. Or, action statements could perform illegal writes. DTrace won't execute unsafe code.

Workflow. Both the USENIX article [CSL04] and the ACM Queue article referenced above contain example usages of DTrace. In high-level terms: first identify a problem; then, use standard system monitoring tools, plus custom DTrace queries, to collect data about the problem (and resolve it).

WAIT

Another approach which recently appeared in the research literature is the WAIT tool out of IBM. Unfortunately, this tool is not generally available. Let's talk about it anyways.

Like DTrace, WAIT is suitable for use in production environments. It uses hooks built into modern Java Virtual Machines (JVMs) to analyze their idle time. It performs a sampling-based analysis of the behaviour of the Java VM. Note that its samples are quite infrequent; they suggest that taking samples once or twice a minute is enough. At each sample, WAIT records the state of each of the threads, which includes its call stack and participation in system locks. This data enables WAIT to compute (using expert rules) an abstract "wait state". The wait state indicates what the process is currently doing or waiting on, e.g. "disk", "GC", "network", or "blocked".

Workflow. You run your application, collect data (using a script or manually), and upload the data to the server. The server provides a report which you use to fix the performance problems. The report indicates processor utilization (idle, your application, GC, etc); runnable threads; waiting threads (and why they are waiting); thread states; and a stack viewer.

The paper presents six case studies where WAIT helped solve performance problems, including deadlocks, server underloads, memory leaks, database bottlenecks, and excess filesystem activity.

References

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.