Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

I Spy

Diomidis Spinellis

Knowledge is power. —Sir Francis Bacon

The ultimate source of truth regarding a program is its execution. When a program runs, everything comes to light: correctness, CPU and memory use, and even interactions with (potentially buggy) libraries, operating systems, and hardware. Yet, this source of truth is also fleeting, rushing into oblivion at the tune of billions of instructions per second. Worse, capturing that truth can be a tricky, tortuous, or downright treacherous affair.

Peeking into a program's operation typically involves preparing a special version of it: we might compile it with specific flags or options, link it with appropriate libraries, or run it with suitable arguments. Often, we can't easily reproduce a problem, so we need to ship our carefully crafted program version to a customer, who then will have to wait for the problem to appear again. Irritatingly, some of the ways we instrument programs make the program too slow for production use or obfuscate the original problem.

A family of tools ...

We don't lack for ways to spy on a program. If we care about CPU use, we can run our program under a statistical profiler that will interrupt its operation many times every second and note where the program spends most of its time. Alternatively, we can arrange for the compiler or runtime system to plant code by setting a time counter at each function's beginning and end, and we can then examine the time difference between the two points. In extreme cases, we can even have the compiler instrument each basic code block with a counter. Some tools that use these approaches are gprof and gcov on Unix systems, the Extensible Java Profiler (EJP) and the Eclipse and NetBeans profiler plug-ins for Java programs, and NProf and the Common Language Runtime (CLR) Profiler for .NET code. Memory use monitors typically modify the runtime system's memory allocator to keep track of our allocations. Valgrind on Unix systems and the Java software developer's kit JConsole are two players in this category.

Locating a bug involves either inserting logging statements in our program's key locations or running the code under a debugger, which lets us dynamically insert breakpoint instructions. I discussed both approaches in the May/June 2006 column.

Nowadays, however, most performance problems and quite a few bugs involve using third-party libraries or interactions with the operating system. One way to resolve these issues is to look at the calls from our code to that other component. By examining each call's time stamp or by looking for an abnormally large number of calls, we can pinpoint performance problems. The arguments to a function can also often reveal a bug. Tools in this category include ltrace, strace, ktrace, and truss (on Unix) and APIS32 or TracePlus (on Windows). These tools typically work by using special APIs or code-patching techniques to hook themselves between our program and its external interfaces.

Finally, our program might work fine, only to have the operating system act up. In these cases, we need to put the operating system un-

 0740-7459/07/$25.00 © 2007 IEEE

der a microscope. Fortunately, modern operating systems zealously monitor their operation and expose various performance figures through tools such as vmstat, netstat, and iostat on Unix systems or the Event Tracing for Windows framework.

Most tools I've examined so far have been around for ages and can help solve a problem once we've located its approximate cause. They also have several drawbacks: they often require us to take special actions to monitor our code, they can decrease our system's performance, their interfaces are idiosyncratic and incompatible with each other (each one shows only a small part of the overall picture), and sometimes important details are simply missing.

... And one tool to rule them all

The "gold winner" in the *Wall Street Journal*'s 2006 Technology Innovation Awards contest was a tool that addresses all the shortcomings I outlined. DTrace, Sun's dynamic-tracing framework, provides uniform mechanisms for spying comprehensively and unobtrusively on the operating system, application servers, runtime environments, libraries, and application programs. It's open source under Sun's fairly liberal Common Development and Distribution License. At the time of writing, DTrace is part of Sun's Solaris 10, and it's also being ported to Apple's Mac OS X version 10.5 and FreeBSD. If you don't have access to DTrace, you can experiment with it by installing a freely downloadable version of Solaris Express on an unused x86 machine. I must warn you, however, that I've found DTrace to be seriously addictive.

Unsurprisingly, DTrace isn't a summer holiday hack. The three Sun engineers behind it worked for several years to develop mechanisms to safely instrument all operating system kernel functions, any dynamically linked library, any application program function or specific CPU instruction, and the Java virtual machine. They also developed a safe interpreted language in which we can write sophisticated tracing scripts without damaging the operating system's functioning, and they came up with aggregating functions that can summarize traced data in a scalable way without excessive memory overhead. DTrace integrates technologies and wizardry from most existing tools and some notable interpreted languages to provide an all-encompassing platform for program tracing.

We typically use the DTrace framework through the dtrace command-line tool. To this tool, we feed scripts we write in a domain-specific language called D; dtrace installs the traces we've specified, executes our program, and prints its results. D programs simply consist of pattern and action pairs like those found in the awk and sed Unix tools and in many declarative languages. A pattern (called a *predicate* in the DTrace terminology) specifies a *probe*—an event we want to monitor. DTrace comes with thousands of predefined probes (49,979 on my system). Additionally, system programs (such as application servers and runtime environments) can define their own probes, and we can also set a probe anywhere we want in a program or in a dynamically linked library. For example, the command

```
dtrace -n syscall:::entry
```

will install a probe at the entry point of all operating system calls. The default action will be to print the name of each system call executed and the process ID of the calling process. We can combine predicates and other variables using Boolean operators to specify more complex tracing conditions.

In the previous invocation, `syscall` specifies a *provider*—a module providing some probes. Predictably, `syscall` supplies probes for tracing operating system calls (463 probes on my system). For example, one of these probes, `syscall::open:entry`, is the entry point to the `open` system call. DTrace contains tens of providers, providing access to statistical profiling, all kernel functions, locks, system calls, device drivers, I/O events, process creation and termination, the network stack's management information base (MIB), the scheduler, virtual-memory operations, user program functions, arbitrary code locations, synchronization primitives, kernel statistics, and Java virtual machine operations.

With each predicate, we can also define an *action*. This action specifies what DTrace will do when a predicate's condition is satisfied. For example, the command

```
dtrace -n
'syscall::open:entry
{trace(copyinstr(arg0));}'
```

will list the name of each opened file.

Actions can be arbitrarily complex: they can set global or thread-local variables, store data in associative arrays, and aggregate data with functions like `count`, `min`, `max`, `avg`, and `quantize`. For instance, the following program will summarize the number of times each process gets executed over the dtrace invocation's lifetime:

```
proc:::exec-success {
@proc[execname] = count()}
```

In typical use, DTrace scripts range from one-liners, like the ones I gave here, to tens of lines containing multiple predicate-action pairs. A particularly impressive example listed on a DTrace blog illustrates the call sequence from a Java method invocation, through the C libraries and an operating system call, and down to the operating system device drivers. As software engineers, we've spent a lot of effort creating abstractions and building walls around our systems; more impressively, it looks like we've also found ways to examine our isolated blocks holistically.

Post your comments online by visiting the column's blog: **www.spinellis.gr/tools**

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.