ACM Queue Architecting Tomorrow's Computing Why Join ACM?

Hidden in Plain Sight

view issue

by Bryan Cantrill | February 23, 2006

Topic: Performance

HIDDEN IN PLAIN SIGHT
IMPROVEMENTS IN THE OBSERVABILITY OF SOFTWARE CAN HELP YOU DIAGNOSE YOUR MOST CRIPPLING PERFORMANCE PROBLEMS.
BRYAN CANTRILL, SUN MICROSYSTEMS

In December 1997, Sun Microsystems had just announced its new flagship machine: a 64-processor symmetric multiprocessor supporting up to 64 gigabytes of memory and thousands of I/O devices. As with any new machine launch, Sun was working feverishly on benchmarks to prove the machine's performance. While the benchmarks were generally impressive, there was one in particular—an especially complicated benchmark involving several machines—that was exhibiting unexpectedly low performance. The benchmark machine—a fully racked-out behemoth with the maximum configuration of 64 processors—would occasionally become mysteriously distracted: Benchmark activity would practically cease, but the operating system kernel remained furiously busy. After some number of minutes spent on unknown work, the operating system would suddenly right itself: Benchmark activity would resume at full throttle and run to completion. Those running the benchmark could see that the machine was on course to break the world record, but these minutes-long periods of unknown kernel activity were enough to be the difference between first and worst.

Given the importance of both the benchmark and the machine, Sun's top development and support engineers were called in, and work proceeded on the problem around the clock and around the globe. Beginning with the initial symptom of the problem—high lock contention in the networking stack—progress on the problem was very slow and painful. Based on available data, a hypothesis would be formed, but because there was no way to explore the hypothesis without gathering additional data, custom data-gathering kernel modules had to be created and loaded. Creating these modules was a delicate undertaking. The entire cycle of hypothesis creation through instrumentation and analysis was taking hours—and that was assuming that no mistakes were made. Mistakes were painful, as any error in creating the custom module would result in a reboot—and reboots of this particular machine and configuration took on the order of 90 minutes.

While iterations on the problem revealed several ways in which the networking stack could be improved to relieve the observed symptoms, it took nearly a week to determine the true underlying root cause: The benchmark machine had been misconfigured to act as a router. Normally, this wouldn't be a problem, but because of intermittent router failure elsewhere in the lab, the benchmark machine would occasionally begin to route packets furiously between two other machines associated with the benchmark. This routing activity would induce enough contention in the networking stack to force the system into a heretofore unseen failure mode—one that induced yet more contention.

Was this an obvious problem? Absolutely, in retrospect. But the cascading symptoms of the problem were so far removed from the root cause that it was impossible to see the larger problem without modifying the system itself. Perhaps most disconcertingly, this problem was a relatively simple one, involving relatively few layers of software. Had more components of the software stack been involved—had the system had to be modified at multiple layers just to observe it—it would have been exponentially more difficult to diagnose.

The larger problem here is software observability, or more accurately, the pronounced lack of it. We have built mind-bogglingly complicated systems that we cannot see, allowing glaring performance problems to hide in broad daylight in our systems. How did we get here? And what can be done about it?

ROOTS OF THE PROBLEM

The origins of the software observability problem, as with so many other software problems, can be found in software's strange duality as both information and machine: Software has only physical representation, not physical manifestation. That is, running software doesn't reflect light or emit heat or attract mass or have any other physical property that we might use to see it. In antiquity, computers addressed this by having explicit modes in which every instruction was indicated as it was executed. In the age of the microprocessor—and with the ability to execute billions of instructions per second—such facilities have long since become impractical, if not impossible. There is, therefore, no physical way to determine which instructions are being executed; the only way to see the software being executed is to modify the software itself. Software engineers have long done exactly that, adding constructs that allow their software to be optionally seen. While the exact manifestations vary, these constructs typically look something like this:

```
if (tracing_enabled)
  printf("we got here!\n");
```

There are many variants of this concept—for example, a piece of data may be logged to a ring buffer instead of being explicitly printed, or the conditional may be derived from an environment variable instead of a static variable—but the common idea is conditional execution based upon the value of the data. Constructs of this nature share an unfortunate property: They make the system slower even when they are not in use. That is, the mere presence of the conditional execution induces a load, a compare, and a branch. While clearly not problematic to execute every once in a while, the cost of these instructions becomes debilitating if they are used extensively: If one were to litter every function or every basic block with this kind of construct, the system would be too slow to ship. It has therefore become a common software engineering practice to retain such constructs, but to automatically strip them out of production code using techniques such as conditional compilation. While this allows the infrastructure to remain, it unfortunately bifurcates software into two versions: one that can be seen that is used in development and test, and one that can't be seen that is shipped or deployed into production.

This gives rise to a paradox: Performance problems are increasingly likely to be seen in production, but they can be understood only in development. To address a performance problem seen in production, the problem must therefore be reproduced in either a development or test environment. In a software system, as in any sufficiently complicated system, disjointed pathologies can manifest the same symptoms: Reproducing symptoms (e.g., high CPU load, heavy I/O traffic, long latency transactions, etc.) does not guarantee reproducing the same underlying problem. To phrase this phenomenon in the vernacular of IT, you might have heard or said something like this:

> *"Good news: We were able to reproduce it in dev, and we think that we found the problem. Dev has a fix, and we're scheduling downtime for tonight to put it into prod. Let's keep our fingers crossed..."*

Only to have it followed the next day by something like this:

> *"Hey, yeah, about that fix...well, the good news is that we are faster this morning; the bad news is that we're only about 3 percent faster. So*

RECENTLY ON SLASHDOT

- Communications Surveillance: Privac and Security
- Making Sense of Revision Control Systems
- Privacy, Mobile Phones and Ubiquitou Data Collection

RELATED CONTENT

MODERN PERFORMANCE MONITORING
The modern Unix server floor can be a diverse universe of hardware from several vendors and software from several sources. Often, the personnel needed to resolve server floor performance issues are not available o for security reasons, not allowed to be present at the very moment of occurrence. Even when, as luck might have it, the right personnel are actuall present to witness a performance "event," the tools to measure and analyze the performance of the hardw and software have traditionally been sparse and vendor-specific.
by Mark Purdy | February 23, 2006
0 comments

PERFORMANCE ANTI-PATTERNS
Performance pathologies can be found almost any software, from user to kernel, applications, drivers, etc. At Su we've spent the last several years applying state-of-the-art tools to a Uni kernel, system libraries, and user applications, and have found that man apparently disparate performance problems in fact have the same underlying causes. Since software patterns are considered abstractions o positive experience, we can talk about the various approaches that led to thes performance problems as anti-pattern —something to be avoided rather than emulated.
by Bart Smaalders | February 23, 2006
0 comments

BROWSE THIS TOPIC:

PERFORMANCE

it's back to the war room..."

If this sounds eerily familiar, it's because you have fallen into the trap endemic to trying to understand production problems by reproducing their symptoms elsewhere: You found a problem that was not the problem.

BUT WAIT, IT GETS WORSE

The lack of observability endemic to production software is bad enough, but the layering of software abstraction makes the performance problem much more acute. Normally, software abstraction is good news, for it is abstraction that empowers one to develop an application without having to develop the Web server, application server, database server, and operating system that the application rests upon. This power has a darker side, however: When developing at higher layers of abstraction, it is easier to accidentally induce unintended or unnecessary work in the system. This is tautologically true: To be at a higher layer of abstraction, less code must induce more work, meaning it takes less of a misstep to induce more unintended consequences.

Unfortunately, this unnecessary or unintended work tends to multiply as it cascades down the stack of abstraction, so much so that performance problems are typically first understood or characterized at the very lowest layer of the stack in terms of high CPU utilization, acute memory pressure, abnormal I/O activity, excessive network traffic, etc. Despite being a consequence of higher-level software, symptoms at the lowest layer of the stack are likely to be most immediately attributable to activity in the next lowest layer—for example, the operating system or the database.

This presents another paradox: System performance problems are typically introduced at the highest layers of abstraction, but they are often first encountered and attributed at the lowest layers of abstraction. It is because of this paradox that we have adopted the myth that the path to performance lies nearly exclusively with faster hardware: faster CPUs, more networking bandwidth, etc. When this fails, we have taught ourselves to move to the next layer of the stack: to demand faster operating systems, faster databases, and better compilers. Improving these components undoubtedly improves performance, but (to use a perhaps insensitive metaphor) it amounts to hunting vermin: Depending on the relatively small iterative improvements at the lowest layers of the stack amounts to trying to feed a family on the likes of squirrel and skunk. If we can move up the stack—if we can find the underlying performance problems instead of merely addressing their symptoms—we can unlock much more substantial performance gains. This is bigger game to be sure; by focusing on performance problems higher in the stack, we can transition from hunting vermin to hunting cow—big, slow, stupid, tasty cow.

CONSTRAINTS ON A SOLUTION

To hunt cow in its native habitat, the focus of observability infrastructure must make two profound shifts: from development to production, and from programs to systems. These shifts have several important implications. First, the shift from development to production implies that observability infrastructure must have zero disabled probe effect: The mere ability to observe the system must not make the delivered system any slower. This constraint allows only one real solution: Software must be optimized when it ships, and—when one wishes to observe it—the software must be dynamically modified. Further, the shift from programs to systems demands that the entire stack must be able to be dynamically instrumented in this way, from the depths of the operating system, through the system libraries, and into the vaulted heights of higher-level languages and environments. There must be no dependency on compile-time options, having source code, restarting components, etc.; it must be assumed that the first time a body of software is to be observed, that software is already running in production.

Dynamically instrumenting production systems in this way is a scary proposition, especially when the operating system kernel is brought into the mix. This leads us to the most important implication of the shift in focus from development to production: Observability infrastructure must be absolutely safe. This safety constraint cannot be overemphasized: In production systems, outage is always unacceptable, and even an outage resulting from operator error is likely to be blamed as much on the system as on the operator. If the use of a tool is in any way involved with a production outage, it is likely that the tool will be banished forever from the production environment. Safety in observability infrastructure, like security in an operating system, cannot be an afterthought: It must be considered an absolute, non-negotiable constraint on the architecture.

ONE SOLUTION: DTRACE

At Sun we have developed DTrace, a facility for dynamic instrumentation of production systems. DTrace is most clearly differentiated from prior work by its focus on production systems and especially by its strict adherence to the safety constraint. While it is a component of the operating system (it was originally developed for Solaris 10), DTrace itself is open source and as such can be ported to other systems (in particular, a port of DTrace to FreeBSD has been initiated1). DTrace is a sophisticated system (see Cantrill, Shapiro, and Leventhal from the 2004 Usenix conference2 and the Solaris Dynamic Tracing Guide3 for more information), but it's worth elucidating its higher-level principles and how these principles guided the evolution of its architecture.

To assure zero disabled probe effect, DTrace was designed around the idea of dynamic instrumentation. While the specific techniques for dynamic instrumentation are often instruction-set-specific and arcane in nature, they share a general principle: They modify running code to revector control flow to a body of interposition code that collects some information, and then returns control flow to just beyond the point of instrumentation. The mechanism for revectoring control varies, but it is most often an instruction either to issue a software trap or to unconditionally branch to interpositioned code. In sharing this general principle, dynamic instrumentation techniques also share a common failing: No technique can work in all contexts in the system. That is, in any system there are some contexts that are simply too delicate to be dynamically instrumented. (Such contexts often include low-level code for interrupt handling, context switching, or synchronization.)

Abiding by the safety constraint while still allowing dynamic instrumentation thus poses something of a challenge: How does one give the flexibility of dynamic instrumentation without compromising the safety required in a production system? In DTrace, we achieved this by separating the way the system is instrumented from the framework that consumes that data. The methodology for instrumenting the system lies in providers; these providers make available probes via the DTrace framework. Safety of instrumentation thus becomes the responsibility of the providers, which publish only those probes that can be safely enabled. By constraining DTrace consumers to enabling only published probes, the system—not the user—becomes responsible for determining what can be safely instrumented. This is a compromise of sorts in that it comes at some loss of flexibility: For example, a provider may be found to be unnecessarily conservative when deciding that a particular body of code can't be instrumented. This minor loss of flexibility is required to assure that the user cannot accidentally or willfully sacrifice system integrity. This is an assurance that any tool must be able to make before it can be used on production systems.

The providers implement the mechanism for instrumenting different parts of the system, but how to act on that instrumentation? That is, what action does one wish to take when instrumenting a component of the system? Our experiences with some of the more primitive tracing facilities that predated DTrace often left us frustrated: It always seemed as if the data that we wanted at a given point was just beyond the data provided. In designing DTrace, we wanted to obviate these frustrations by making actions entirely programmable.

This immediately implied two classes of architectural decisions: First, in what language should actions be specified? Second, how can the system be designed to assure that those programmable actions are executed in a safe manner? Given that safety was our constraint, we addressed the latter of these first: to allow programmable actions to be executed safely in arbitrary contexts, it was clear that we needed to develop a virtual machine that could act as a target instruction set for a custom compiler. This was clear because the alternative—executing user-specified code natively in the kernel—is untenable from a safety perspective: Even if you could implement the substantial amount of static analysis necessary to assure that native code does not perform an illegal operation, you are still left with the intractability of the Halting Problem.

We therefore developed a simple, purpose-built virtual machine designed to execute user-specified code in the operating system kernel without side effect. Safety is assured by carefully limiting the instruction set architecture: Our virtual instruction set has no backwards branches, supports calls only to defined subroutines in the runtime, doesn't allow arbitrary stores, etc. (Note that we obviously didn't solve the Halting Problem; we merely avoided it by designing a virtual machine that does not allow Turing-complete languages.) Importantly, the virtual machine does allow arbitrary loads; loads to unmapped memory (or worse, memory-mapped devices for which even loads may have side effects) are caught, flagged, and handled gracefully.

With the ability to execute arbitrary actions safely, we could focus on designing the language in which those actions should be expressed. For this purpose, we developed D, a C-like language with DTrace-specific extensions such as associative arrays, thread-local variables, and so on. The design of

D reflects the safety constraint: It explicitly lacks loop constructs, user-defined functions, stores to arbitrary addresses, and other features that could compromise safety. Because our virtual machine has a mechanism for safe loads, D can (and does) allow for C-like pointer chasing.

Given a safe foundation for dynamic instrumentation and arbitrary actions, the next series of problems focused on data management. Most obviously, we needed a mechanism for filtering out unwanted data at the source. For this, we added the notion of a predicate: a conditional D expression attached to an action such that the action will be taken only if the expression evaluates to true. Predicates, combined with the power of D and the heterogeneous instrumentation of DTrace, allow for sophisticated filtering. For example, figure 1 shows a script that uses predicates, a thread-local variable, and providers for both user- and kernel-level code to follow flow control through an application and into the operating system kernel; figure 2 shows the output from running this script.

FIG 1

```
pid$target::fwrite:entry
{
        self->follow = 1;
}

pid$target:::entry,
pid$target:::return,
fbt:::entry,
fbt:::return
/self->follow/
{
        trace(probemod);
}

pid$target::fwrite:return
/self->follow/
{
        self->follow = 0;
        exit(0);
}
```

D script to follow code flow from the "fwrite(3C)" library call in a target process, instrumenting both the target process and the operating system kernel. The lines containing colons are probe descriptions; the expressions between the slashes are predicates, and the lines contained within the braces constitute actions. The probe descriptions describe what to instrument, and the actions are taken if (and only if) their corresponding predicate evaluates to true. The "self->" syntax denotes a thread-local variable. This script thus sets a thread-local variable ("follow") to 1 when a thread calls "fwrite", and instruments every function entry and return in both the process and the kernel—with the predicate that the thread-local variable is set. (Thread-local variables have the value 0 if they have not been set.) This has the effect of following flow-control from fwrite through the system.

As figure 2 shows, DTrace allows the ability to cut through different components in the stack of abstraction, even across protection boundaries. Any framework for observability must not only be able to cut through layers of components, but also allow for patterns to be seen in the interactions. To effect this in DTrace, we elevated the aggregation of data to a first-class notion: Instead of having to clog the system with data to be post-processed, DTrace allows data to be keyed on an arbitrary n-tuple and aggregated at the source. This reduces the data flow potentially by a factor of the number of data points—without introducing any of the statistical inaccuracy endemic in sampling methodologies. Importantly, the DTrace aggregation mechanism scales linearly: Information is kept on a per-CPU basis and then aggregated across CPUs periodically at the user level. This is no mere implementation detail; tools for observing parallel systems must always scale better than the software they are trying to observe, lest their own bottlenecks obscure the scalability of the underlying system. The DTrace aggregation mechanism, as demonstrated in figure 3, allows questions about the system to be answered concisely.

FIG 2

```
-> fwrite                       libc.so.1
  -> _fwrite_unlocked           libc.so.1
    -> _realbufend              libc.so.1
      -> getxfdat               libc.so.1
      <- getxfdat               libc.so.1
    <- _realbufend              libc.so.1
    -> _flsbuf                  libc.so.1
      ...
      -> _write                 libc.so.1
        -> write                genunix
          -> getf               genunix
            -> set_active_fd    genunix
            <- set_active_fd    genunix
          <- getf               genunix
          ...
        <- write                genunix
      <- _write                 libc.so.1
      ...
    <- _flsbuf                  libc.so.1
  <- _fwrite_unlocked           libc.so.1
<- fwrite                       libc.so.1
```

Trimmed output from the script in figure 1. The lines from "libc.so.1" are functions executed at the user level; those with "genunix" are executed in the kernel.

FIG

```
{
        @[execname] = quantize(arg2);
}
```

(Above) Simple D script to instrument the write(2) system call, getting a power-of-two histogram of write sizes keyed by application name.

(Right) One such histogram in the output for the adjacent script. This output indicates that (for example) the application "acroread" made 93 write system calls sized between 16 and 31 bytes, inclusive.

```
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |                                         2
               4 |@                                        10
               8 |@@                                       11
              16 |@@@@@@@@@@@@@@@                          93
              32 |@@@@@@                                   43
              64 |@@@@                                     30
             128 |@@@@                                     32
             256 |@@@@                                     25
             512 |@@                                       14
            1024 |@                                        9
            2048 |@                                        6
            4096 |@                                        10
            8192 |                                         0
```

The architectural elements of DTrace—safe and heterogeneous dynamic instrumentation, arbitrary actions and predicates, and scalable, in situ data aggregation—allow for unprecedented observability of production systems. In using DTrace, however, we quickly encountered a thorny problem: Instrumenting the system effectively required knowledge of the implementation. To rectify this shortcoming, we introduced mechanisms for providers both to define the interface stability of their probes and to declare translators that can translate from implementation-specific data structures into more abstract, implementation-neutral ones. Together, these mechanisms allow for providers to abstract away from their implementation by providing probes and arguments that represent subsystem semantics. For example, instead of requiring users to know the functions that perform I/O in the operating system, we introduced an io provider with probes with names such as start and done and with implementation-neutral arguments relevant to an I/O operation such as information about the buffer, the device, the file, and so on. An example, including sample output, of using the I/O provider is shown in figure 4.

FIG 4

```
io:::start
{
        printf("%s %s %s\n", execname,
            args[0]->b_flags & B_READ ? "reads" : "writes",
            args[2]->fi_pathname);
}
```

D script using the "io" provider to print out all I/O operations as they happen. The definitions of the probe and its arguments are abstracted away from the implementation and are well documented in the Solaris Dynamic Tracing Guide (2005).

```
soffice.bin writes /var/tmp/sv2i7.tmp/sv30i.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30j.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30j.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30k.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30l.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30l.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30l.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30l.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30l.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30m.tmp
soffice.bin writes /.staroffice7/user/backup/queuearticle0.sxw
soffice.bin writes /.staroffice7/user/backup/queuearticle0.sxw
soffice.bin writes /.staroffice7/user/backup/queuearticle0.sxw
soffice.bin writes /Documents/queuearticle.sxw
soffice.bin writes /Documents/queuearticle.sxw
soffice.bin writes /Documents/queuearticle.sxw
soffice.bin writes /var/tmp/sv2i7.tmp/sv30n.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30n.tmp
soffice.bin writes /var/tmp/sv2i7.tmp/sv30n.tmp
fsflush writes /var/tmp/sv2i7.tmp/sv30h.tmp
fsflush writes /var/tmp/sv2i7.tmp/sv30h.tmp
```

Output from script above when run while saving the file that contains this article.

With the ability to instrument the system in terms of its semantics, a much larger problem came into focus: While we could instrument applications written in natively compiled languages such as C, C++, Fortran, Objective C, etc., we had no insight into applications written in interpreted languages or languages targeted at a virtual machine environment. This was a critical problem, because the stack of abstraction towers is much higher than the natively compiled languages: Any observability framework that claims systemic observability must be able to reach into the more abstract ilk of Java, Python, Perl, PHP, and Ruby.

Instrumenting these environments is difficult: Whether interpreted or compiled to bytecode, the system has very little visibility into such an environment's notion of program text—let alone how to instrument it. To allow for instrumentation of these environments, we developed a mechanism for user-level providers, allowing probes to be exported that correspond to semantically significant events in their environment. These events vary, but they typically include calling a function or method, creating an object, performing garbage collection, etc. As of this writing, providers have been implemented for Java, Perl, PHP, Python, Ruby, Tcl, and (we're not making this up) APL. Example output of extending the script in figure 1 to PHP is displayed in figure 5. These providers are largely still prototypes, but they are remarkable for the degree to which they have improved the debuggability of their respective environments.

```
-> launch                php
```

FIG

```
-> three                    php
  -> fwrite                 libc.so.1
    -> _fwrite_unlocked     libc.so.1
      -> _realbufend        libc.so.1
        -> getxfdat         libc.so.1
        <- getxfdat         libc.so.1
      <- _realbufend        libc.so.1
      -> _flsbuf            libc.so.1
        ...
        -> _write           libc.so.1
          -> write          genunix
            -> getf         genunix
              -> set_active_fd  genunix
              <- set_active_fd  genunix
            <- getf         genunix
            ...
          <- write          genunix
        <- _write           libc.so.1
        ...
      <- _flsbuf            libc.so.1
    <- _fwrite_unlocked     libc.so.1
  <- fwrite                 libc.so.1
-> two                      php
  -> fwrite                 libc.so.1
  ...
```

Trimmed output from a script similar to that in figure 1, but modified to follow execution from a PHP function, "launch." Note the execution through PHP, the system libraries, into the kernel and back.

EXPERIENCE

In the more than two years that DTrace has been publicly available, it has been used successfully on many interesting "cow hunts," both inside and outside of Sun. One particularly fruitful hunting ground was a production SunRay server inside of Sun. This was a 10-CPU machine with 32 gigabytes of memory and approximately 170 users on which DTrace was used to find many problems (the most infamous of which is discussed extensively in the aforementioned reference 2).

One common complaint about this system was its high level of CPU utilization, which led to a natural question: Which applications were consuming CPU cycles? There are several ways to answer this with DTrace; the simplest is to use the DTrace profile provider, whose probes fire at a uniform (and entirely programmable) interval. This allows for a sampling methodology, and it allows one to, for example, sample the currently running application 1,234 times per second on each CPU:

```
profile-1234hz
{
  @[execname] = count();
}
```

Running such a script on the server for 30 seconds or so yielded the following output (the left column is the application name and the right column is the number of samples):

mozilla-bin	662
xprop	719
dtmail	886
netscape-bin	1412
nautilus	1438
ps	2523
java	3451
Xsun	6197
netscape	6600

Much of this output wasn't too surprising: Given that this machine was serving up desktops to 170 users, one would expect to see X servers (Xsun) and Web browsers (netscape) and mail readers (dtmail) as the top CPU consumers. There was one very surprising data point, however: The fourth biggest CPU consumer was ps, the Unix command to list the processes in the system. This is an administrative command that is normally executed interactively. Were a ton of people running ps for some reason? The D script in figure 6 explores this question, with the output shown in figure 7.

FIG 7

```
ps #1 started at t+5 seconds, took 1338 milliseconds wall time
ps #2 started at t+12 seconds, took 1413 milliseconds wall time
ps #3 started at t+20 seconds, took 1505 milliseconds wall time
ps #4 started at t+41 seconds, took 1454 milliseconds wall time
ps #5 started at t+43 seconds, took 1412 milliseconds wall time
...
```

This revealed that the ps command was being run with disturbing regularity. (That each invocation was taking nearly a second and a half to run was not terribly surprising; with more than 3,000 processes, one would expect a complete process listing to be reasonably time consuming.) One would not expect the ps command to be run every ~10 seconds on a normal machine. Who was running this? And why? Answering this kind of question is a snap with DTrace:

```
proc:::exec
/args[0] == "/usr/bin/ps"/
```

```
{
@[execname, ustack()] = count();
}
```

This enabling aggregates on both the application name and a user stack backtrace. Running the above revealed that all invocations of ps were ultimately coming from an application named cam, and they were all coming from a function named SendUsageRec.

This was immediately suspicious: Cam is a wrapper program written by Sun's IT department that is designed to track usage of an underlying application; one would not expect cam to be doing much work at all, let alone launching a ps command. Looking at the source to SendUsageRec shown in figure 8 revealed at least part of the problem.

```
void Usage::SendUsageRec(UsageRecType recordtype)
  // this function gets the usage values and sends a record to the daemon
  {
    ...
  // if this is not the final record find cputime from the 'ps' command
    else if (recordtype != FINAL)
      {
        cputime = 0;
        elapsedtime = time(NULL) - xinvocation -> Starttime();
        sprintf(cmd, "/bin/ps -s %d |cut -c13-", getpid());
        if ((ptr = popen(cmd, "r" )) != NULL)
        {
```

FIG 8

If you've done much Unix systems programming, looking at this code might provoke a verbal exclamation—like having your toes stepped on or being pricked with a pin. What is this code doing? The answer requires some explanation of the cam wrapper. To track usage, cam obtains a timestamp and launches the underlying application; later, when the underlying program exits, cam again obtains a timestamp, takes the difference, and sends the total usage time to a daemon. To deal with the case of an underlying application that doesn't exit before the machine is rebooted, cam also wakes up every 30 minutes and executes the (horribly inefficient) code in figure 8 to determine how much time the underlying application has been running, sending the usage time to the daemon. (Not only is this code horribly inefficient, it is also terribly broken: Cutting from column 13 includes not only the desired TIME field, but also the final two characters from the TTY field.) On a desktop, which might be running two or three cam-wrapped applications, this isn't a problem. On this server, however, with its 170 users, someone's 30 minutes were up every ten seconds.

Some back-of-the-envelope math revealed that at some point—probably around 400 users or so—the server would have (on average) one ps running all the time. Needless to say, neither ps nor its underpinnings in the /proc file system in the kernel were designed to be in this kind of constant use. Indeed, after having discovered this problem, several previously mysterious problems in the system—including abnormally high lock contention in the kernel's process management subsystem—began to make much more sense. So resolving this problem not only reduced the CPU usage in the system (thus reducing latency), but also improved scalability (thus improving throughput).

This is a good example of the kinds of problems that have been found with DTrace: It was a knuckleheaded problem high in the stack of abstraction, and the symptoms of the problem were systemic in nature. This problem could not have been understood in terms of a single program because it was only when multiple instances of the same program were taken together that the systemic problem could be seen. Although this problem might have been silly, the effects that it had on the system were substantial. Perhaps most distressing, before DTrace was brought in, there had been several months of intensive (but fruitless) effort spent trying to understand the system. This was not a failing of the sharp people who had been previously examining the system, it was a failing of the dull tools that they had been stuck using—tools that did not allow them to correlate symptoms at the bottom of the stack with the causes at the top.

FUTURE WORK

While DTrace has allowed for new degrees of system observability, there is still much work to be done. For example, although our user-level provider mechanism has allowed for some visibility into dynamic environments, it is too coarse-grained: With single probes corresponding to high-frequency events such as entering a method or function, the enabled probe effect is too high when one is interested in instrumenting only a small subset of a program. While DTrace can instrument points of text, it has no real way of recording data (or even arguments) from these environments. These problems are nontrivial; where they have been solved (using bytecode instrumentation or similar techniques), it is nearly always with a solution specific to the language instead of generic to the system.

Beyond DTrace, new capacities for software observability have opened up new potential in established fields such as software visualization. In particular, interesting system visualization work such as IBM's PV research prototype4 should now be possible on production systems, dramatically increasing their scope and impact.

Finally, while solving the single-system observability problem pours a critical foundation, it is but incremental work on a much more challenging problem: observing distributed systems. One potentially promising area is combining single-system observability infrastructure with OS-level virtualization technologies5 to obtain distributed debugging within the confines of a single machine. While promising, this is ultimately a solution limited to development systems: Observing production distributed systems remains an open problem. One noteworthy body of work in this domain is Google's Sawzall,6 a system designed to analyze logging data across many machines. With its filters and aggregators, Sawzall has some similar notions to DTrace, but differs in that it post-processes data, whereas DTrace processes data in situ. Still, the confluence of ideas from systems such as DTrace and Sawzall may provide some progress on the long-standing problem of distributed systems observability.

Given the success of DTrace, we anticipate other solutions to the system observability problem. To assure success, future solutions should abide by the same constraint: absolute safety on production systems above all else. Abiding by this constraint—and adding mechanisms to instrument the entire stack of abstraction in ways that highlight patterns—has allowed for new observability where it is most badly needed: in production systems, and high in the stack of abstraction. With the ability to observe the entire stack of running, production software, cow season is open, and there's no bag limit. Q

REFERENCES

1. FreeBSDTrace project; http://www.sitetronics.com/wordpress/
2. Cantrill, B., Shapiro, M., and Leventhal, A. 2004. Dynamic instrumentation of production systems. Proceedings of the 2004 Usenix Annual Technical Conference.
3. Sun Microsystems. 2005. Solaris Dynamic Tracing Guide.
4. Kimelman, D., Rosenburg, B., and Roth, T. 1997. Visualization of dynamics in real-world software systems. In Software Visualization: Programming as a Multi-Media Experience, MIT Press.
5. Price, D., and Tucker, A. 2004. Solaris zones: Operating system support for consolidating commercial workloads. Proceedings of the 18th Usenix LISA Conference.
6. Pike, R., Dorward, S., Griesemet, R., and Quinlan, S. Forthcoming. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming Journal.

BRYAN CANTRILL is a senior staff engineer in the Solaris Kernel Development Group at Sun Microsystems. His interests include dynamic software instrumentation, postmortem diagnosability, and realtime kernel and microprocessor architecture. Most recently, he (and two colleagues) designed,

implemented, and shipped DTrace, a facility for systemic dynamic instrumentation of Solaris. Cantrill received a Sc.B. in computer science from Brown University.

Originally published in Queue vol. 4, no. 1—
see this item in the ACM Digital Library

About the Author

Bryan Cantrill is a Distinguished Engineer at Sun Microsystems, where he has spent over a decade working on system software, from the guts of the kernel to client-code on the browser and much in between. Along with colleagues Mike Shapiro and Adam Leventhal, Bryan designed and implemented DTrace, a facility for dynamic instrumentation of production systems that won the Wall Street Journal's top Technology Innovation Award in 2006. In 2005, Bryan was named by MIT's Technology Review as one of the top thirty-five technologists under the age of thirty-five, and by InfoWorld as one of their Innovators of the Year. Bryan received the ScB magna cum laude with honors in Computer Science from Brown University. For additional information see the ACM Digital Library Author Page for: Bryan Cantrill

COMMENTS