開
放
的
열린
مفتوح
libre
मुक्त
ಮುಕ್ತ
livre
libero
ముక్త
开放的
açık
open
nyílt
פתוח
オープン
livre
ανοικτό
offen
otevřený
öppen
открытый
வெளிப்படை

Observability Matters: How DTrace Helped Twitter

Adam Leventhal
Brendan Gregg

FishWorks
Sun Microsystems

This Talk

- Twitter performance meltdown
- Introduction to DTrace
- DTrace on the case
- Solutions and results
- DTrace for **your** application
- Q&A

What Is Twitter?

- Social networking/RSS/SMS
- Ruby on Rails application
- Horizontally scaled
- Centralized MySQL backend

Twitter's Problem

- Started with a few users
- Didn't scale with their success
- High latencies could make it unusable
- Many possible suspects:
  - The OS: kernel, libraries, etc.
  - Ruby, MySQL, Apache
  - The Twitter application itself

Enter DTrace

- Luckily: Twitter had DTrace
  - Solaris 10, Mac OS X 10.5, FreeBSD*
- Systemic observability
  - Ruby, Java, JavaScript, C/C++, kernel ...
- Concise answers to arbitrary questions
- Designed for **production** systems
  - Architected to always be safe
  - No overhead when not in use
- Also great for developers

DEMO

Trace all system calls:

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

Aggregate by executable name:

```
# dtrace -n 'syscall:::entry{ @[execname] = count(); }'
```

Examine a user-land process and generate a power-of-two histogram of allocations:

```
# dtrace -c date -n 'pid$target::malloc:entry{ @ = quantize(arg0); }'
```

Trace all I/O on the system:

```
# dtrace -n io:::start
```

In summary: DTrace has **systemic** scope.

The Investigation

- High latencies under load
- What could cause these latencies?
  - on-CPU time: slow/numerous functions, ...
  - off-CPU time: I/O, lock contention, ...

DEMO

Run our load generator program that attempts to mimic the salient problem that we saw when examining the Twitter application in production.

```
$ ./deepstack.rb
```

Use a familiar tool to start with; in this case we used mpstat(1):

```
# mpstat 1
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    0   0    0   450  296  138   64   36    1    0 12362   94   2   0   4
...
```

We're spending all our time in user-land (the usr column) so let's look at what application is taking up our time by using a probe that fires at 1234hz and seeing what application is running:

```
# dtrace -n 'profile-1234{ @[execname] = count(); }'
```

Now that we've see that it's Ruby, let's look at the stack trace in Ruby:

```
# dtrace -n 'profile-1234/execname == "ruby"/{ @[ustack()] = count(); }'
```

Investigation: Steps

1. Started with a broad look at the system using familiar tools
   - mpstat, ...
2. DTrace profiling to sample processes
3. DTrace profiling to sample user stacks

Investigation: So Far

- What we knew:
  - High application latency (from somewhere)
- What we learned:
  - Latency may be due to high CPU load
  - CPU load is due to some Ruby component:
    - Twitter application
    - Standard Ruby libraries
    - Interpreter itself

DEMO

We'll see what functions in the Ruby process are being called most frequently:

```
# dtrace -p `pgrep ruby` -n 'pid$target:::entry{ @[probefunc] = count(); }'
```

Now that we see that it's memcpy(3C), let's see the stack trace:

```
# dtrace -p `pgrep ruby` -n 'pid$target::memcpy:entry{ @[ustack()]=count(); }'
```

Looking at the stack trace a bit, we noticed backtrace() which seemed strange – why would the Twitter application spend so much time taking stack backtraces? We wanted to measure the amount of time spent in backtrace() to confirm our findings (be sure to look at the backtrace.d script):

```
#  ./backtrace.d -p `pgrep ruby`
        backtrace      665 ms
          TOTAL:      1005 ms
...
```

Investigation: Steps

4. Function call counts
5. ustack() for memcpy()
6. Time spent in backtrace()

- What we learned:
  - 40% of CPU time spent in backtrace()

DEMO

We noticed that backtrace() was many a bunch of calls to snprintf(3C) which is used to format strings. Our hope was that the snprintf(3C) calls might be used to format the Ruby stack backtrace for use in an exception object so we wrote a rather complicated script to test this theory (be sure to look at printstack.d):

```
# ./printstack.d -p `pgrep ruby`
<wait>
^C
  ./deepstack.rb:20:in `parserfunc2'
./deepstack.rb:6:in `parserfunc1'
./deepstack.rb:18:in `parserfunc2'
./deepstack.rb:6:in `parserfunc1'
...
./deepstack.rb:6:in `parserfunc1'
./deepstack.rb:33
              550
```

We saw something quite similar at Twitter: a very deep Ruby stack albeit with a more complex texture.

Investigation: Steps

7. Located source of backtrace() calls in Ruby code

- What we learned:
  - Several instances of code like this:

```
@string = (str.string rescue str)
```

Results: No Exceptions

- Rather than blithely calling the method:
```
@string = (str.string rescue str)
```
- ... check first:
```
@string = str.responds_to?(:string) ?
    str.string : str
```
- Result: 30% drop in CPU utilization

Ruby Provider

- Joyent built DTrace-enabled Ruby
- Trace function entry and return
- Probes for line execution, memory allocation, etc.

DEMO

You'll need the DTrace-enabled Ruby which you can find here: http://dtrace.joyent.com
Then run our other load generator program with that version of Ruby:

```
$ .../ruby hotfuncs.rb
```

List all Ruby probes in the DTrace framework:

```
# dtrace -l -n 'ruby*:::'
```

Use the function-entry probe to see what Ruby functions are being called most frequently:

```
# dtrace -n 'ruby*:::function-entry{ @[copyinstr(arg1)] = count(); }'
^C
  func_a                                                          1
  +                                                          120000
  <                                                          120003
```

Use the line probe to find other hotspots:

```
# dtrace -n 'ruby*:::line{ @[copyinstr(arg0), arg1] = count(); }' \
  -n 'END{ printa("%40s:%-6u %6@u\n", @); }'
```

Results: CRC32 in C

- Precision optimization
- Rewrite the CRC32 computation in C
- Estimated result: 15% drop in CPU utilization

DTrace Your Application

- Get your application on an OS with DTrace
- Start with the tools you know and dive deeper with DTrace
- If you can do it today or tomorrow, find Adam and Brendan: we'll help you investigate
- DTrace sees all

Q&A (and links)

- DTrace home page
  - http://www.opensolaris.org/os/community/dtrace
- DTrace-enabled Ruby
  - http://dtrace.joyent.com
- Getting started with DTrace
  - http://blogs.sun.com/ahl/entry/dtrace_boot_camp
- Exception problem:
  - http://dev.rubyonrails.org/ticket/8159
  - http://dev.rubyonrails.org/changeset/6571
- Joyent's blog post on this subject
  - http://joyeur.com/2007/04/24/solaris-dtrace-and-rails

開
放
的
열린
مفتوح
libre
मुक्त
ಮುಕ್ತ
livre
libero
ముక్త
开放的
açık
open
nyílt
פתוח
オープン
livre
ανοικτό
offen
otevřený
öppen
открытый
வெளிப்படை

Observability Matters: How DTrace Helped Twitter

Adam Leventhal
http://blogs.sun.com/ahl

Brendan Gregg
http://blogs.sun.com/brendan