Optimizing Applications Using Oracle Solaris 11 - DTrace

Angelo Rajadurai, Principal Software Engineer
Agenda

- Introduction to Dynamic Tracing
- Nuts and Bolts of DTrace
  - Developing D-scripts
  - D-script Design Patterns
- Performance tuning with DTrace
- Case study
  - Observing and solving locking issues
  - Pin pointing root cause in a Cloud World
  - IO bottlenecks
  - Quick and easy kernel and Userland Profiling
- Other Resources
Oracle Corporation

Native Apps

Apps in VMs

Presentation Layer
JSP/JavaScript/JavaFX

Framework

Application Container
Weblogic

Database
Oracle DB/ MySQL

Operating Systems
Solaris/OEL/OS X/Windows

ruby
python
php
perl
shell

Application Logic

Framework

Native Apps

Application Container
Weblogic

Database
Oracle DB/ MySQL

Operating Systems
Solaris/OEL/OS X/Windows
The Deployment Dilemma

Developer: “I'd ask the database admin”

Sysadmin: “I don't know! Ask the Developer”

Middleware Expert: “Let's ask the Sysadmin”

“May be the middleware guys will know”

Oracle Corporation
Traditional Tools

- **Development Debugging tools** - Tightly bound to programming language, Framework or Application
- **System Tools** (vmstat, mpstat, iostat) - Get system wide view but no ability to drill down to workload
- **truss pstack** - not great for going beyond single process views
- **mdb(1)** - Typically used as postmortem rather than live analysis
The Vicious Cycle

Prebuilt Instruments               Custom Instruments

Add Code ➔ Compile ➔ Custom Instruments ➔ Prebuilt Instruments ➔ Stop App ➔ Restart w/ flags ➔
Solution – Dynamic Tracing

- Dynamically instrument live running system
- Ability to dynamically enable and disable points of instrumentation
- Ability to instrument arbitrary locations
- Low overhead for instrumentation
- No overhead when instrumentation is turned off
- Safe to use in production
Dynamic Tracing in Solaris

DTrace

Oracle Corporation
Using DTrace is Simple
D - Scripts

probe

provider : module : probe : name

/ / predicate /

{ }

actions

→ where

→ when

→ do what
DTrace command line

```
# dtrace -n probe'/predicate/{action}'
```
DTrace providers

- Oracle Corporation

- DTrace core provider
  - ip
  - tcp
  - udp
  - proc
  - fbt
  - iSCSI
  - IO
  - syscall
  - sysinfo
  - profile
  - lockstat
  - kerberose
  - mysql
  - postgres
  - sh
  - FC
  - nfs
  - cpc
  - FC
  - sh
  - nfs
  - cpc

- Other Providers
  - java
  - perl
  - ruby
  - python
  - javascript
  - mysql
  - postgres
  - cpc
  - ip
  - tcp
  - udp
  - proc
  - fbt
  - iSCSI
  - IO
  - syscall
  - sysinfo
  - profile
  - lockstat
  - kerberose
  - nfs
  - sh
  - FC
  - cpc
  - ip
  - tcp
  - udp
  - proc
  - fbt
  - iSCSI
  - IO
  - syscall
  - sysinfo
  - profile
  - lockstat
  - kerberose
  - nfs
  - sh
  - FC
  - cpc
Why DTrace?

• DTrace addresses the following pain points
  • Observing applications in production is hard
  • Debug flags are not Dynamic (requires app restart)
  • Expensive reproduce problems out of production
  • Error induction and code coverage is expensive
DTrace Demo
Getting Started with DTrace

• Excellent collection of D-script
  • DTraceToolKit
  • Docs/Examples - this directory has examples for each script.
  • `pkg install dtrace-toolkit`
  • scripts in `/usr/dtrace/DTT`

• DTrace and IDE integration
  • Project D-Light

• DTrace visualization
  • Chime & DTrace Java API
D-Script Design Patterns

- Structuring your D-Scripts
- D-script Design Patterns
  - The event trace pattern
  - The count pattern
  - What’s in between pattern
  - Time spent pattern
  - Profile Pattern
Event Tracing Pattern

• Good for rare events

Samples

```
dtrace -n swapin’{trace(execname)}’
```

```
syscall::read::entry
/strstr(fds[arg0].fi_pathname, $$1)! = 0/
{
    printf(“%s read from %s\n”, execname, $$1);
}
```
The Count Pattern

Good for collecting summary of events

Samples

dtrace -n on-cpu’{@[execname]=count()’}

syscall:::entry
/execname==$$1/
{
    @[probefunc]=count();
}

Oracle Corporation
What is In Between Pattern

Good for drilling down into details

```c
pid$1::printf:entry
{ 
    self->traceme=1;
}

pid$1:::entry,
pid$1:::return,
fbt:::
/self->traceme/
{ 
    @[probefunc]=count();
}

pid$1::printf:return
{ 
    self->traceme=0;
}
```
Time Spent Pattern

Find Time Spent in various methods

```c
pid$1:libc::method-entry
{
    self->ts[probefunc]=timestamp;
}

pid$1:::method-return
/self->ts[probefunc]/
{
    @[probefunc]=sum(timestamp-self->ts[probefunc]);
    self->ts[probefunc]=0;
}
```
Profile Pattern

• An inexpensive way to find offending method
• Examples

• See **hotuser & hotkernel** scripts in DTraceToolkit

  \[ \text{dtrace -n profile-101'/(arg1/[@ufunc(arg1),caller]=count())'}} \]
Performance Tuning
With DTrace
Case Studies
Demos
Q&A
Hardware and Software
Engineered to Work Together