An Overview of DTrace

Sam Falkner

Solaris Engineering
Sun Microsystems
FROSUG / FRUUG Joint Meeting
November 17, 2005

Agenda

- What is/isn’t DTrace
- The D Language
- Use Cases
  - Kernel Developer
  - Application Developer
  - Programming Language Implementor
  - Systems Administrator
  - End User
- Other Platforms

What is DTrace?

- A Dynamic Tracing facility
  - > profile, debug, learn
- For Application, Library, and Kernel
  - > And all three at once!
- No need to recompile, restart, reboot
- Safe -- no fear of crashes or panics

DTrace is not...

- Not able to modify outcomes
  - DTrace cannot intercept and change the value returned from a function
  - DTrace cannot call arbitrary routines
  - DTrace cannot modify memory
  - DTrace can however affect timing, sometimes purposefully
- Not a general programming language
  - No function calls, no branching or iteration
- Most limitations are to provide absolute safety

The D Programming Language

- Somewhat AWK-like
  - Conditions and actions
  - Their order is important
- “Knows” about C data types
  - Can dereference members of structures

An Easy D script

```
syscall::mount:entry
{
        self->traceme = 1;
}
```

When the kernel begins to handle a mount system call, trace this thread

```
fbt:::return
/self->traceme && arg1 == 28/
{
}
```

If we’re tracing, and we return 28, do the default action

```
syscall::mount:return
{
        self->traceme = 0;
}
```

When the mount is finished, unmark this thread

An Easy D Script

- D Script is one or more clauses
- Each clause is
  - Probe
  - Predicate (optional)
  - Action

```
fbt:::return
/self->traceme && arg1 == 28/
{
}
```

Probes

- Probe is provider:module:function:name
  - `fbt:nfs:nfs4_getsecattr:entry`
- Provider is a DTrace subsystem specializing in one particular thing
- Module is “which kernel module or library”
- Function is usually (not always) a function name
  - Exception: syscall provider uses system call name
- Name can be anything at provider’s discretion
  - Examples: entry, return

Predicates

- These are the only conditionals
- They are optional
  - If no predicate, the action is always taken
- They are in the same context as the action
  - They have access to the same variables, etc.

Actions

- Can do many things! :-)
  - Store data in globals, locals, etc.
  - Print information
  - Commit or discard “speculative” data
- Default action is to print certain data about the probe that is firing
- Remember, order of the clauses is important...

Order Dependency

```
fbt:nfs:nfs4_getsecattr:entry
{
	self->traceme = 1;
}

fbt:nfs:nfs4_getsecattr:return
{
	self->traceme = 0;
}

fbt:nfs::return
/self->traceme/
{
	trace(arg1);
}
```

Oops! We won’t trace the return value from nfs4_getsecattr()

Destructive Actions

- Must be enabled explicitly
  - > -w option to command line
  - > #pragma D option destructive
- Application level destructive actions
  - > dtrace_proc or dtrace_user privilege
  - > stop(), raise(), copyout(), system(), ...
- Kernel level destructive actions
  - > Only run by superuser
  - > breakpoint(), panic(), chill()

Speculative Data

- Sometimes, you don’t know if data will be interesting until it’s gone
- Speculation gives us a place to record data, for now...
- Once we know whether or not we want the data, we can commit or discard
- Example: If a certain function is returning a failure, show me everything that happened leading up to that function

Speculative Data Example

```
syscall::mount:entry
{
        self->spec = speculation();
}

fbt:::return
/self->spec/
{
        speculate(self->spec);
        printf(“returning %d\n”, arg1);
}

/* continued... */
```

Speculative Data Example (cont.)

```
syscall::mount:return
/self->spec && errno != 0/
{
        commit(self->spec);
}
syscall::mount:return
/self->spec && errno == 0/
{
        discard(self->spec);
}
syscall::mount:return
/self->spec/
{ self->spec = 0; }
```

Aggregations

- How many times was this function called?
- What is the average time taken for this system call?
- What is the maximum number of bytes given to the write() system call
- Give me a quantization bar graph showing the breakdown of how much memory is being request via malloc()

Quantization Example

```
pid$target::malloc:entry
{
        @[0] = quantize(arg0);
}
```

```
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@                                        33
               2 |@@@@@                                    204
               4 |@@@@                                     175
               8 |@@@@                                     179
              16 |@@@@@@@@@@@@@@@                          606
              32 |@@@@@@@@                                 313
              64 |@                                        55
             128 |                                         10
```

Kernel Developer

- No need to reboot to debug
- Seldom need to add debugging code
  - > And when you do, you can use static DTrace probes (the “sdt” provider)

Application Developer

- pid provider traces user level processes
- Easy to write custom providers for an application
  - > Create a .d file in your application directory, describing probes
  - > Postprocess object files with "dtrace -G ..."
  - > Documented in the DTrace guide in http://docs.sun.com/ under "Statically Defined Tracing for User Applications"

Programming Language Implementor

- Easy to add a provider for your programming language, just like adding a provider for an application (see previous slide)
- Trace function calls and returns, garbage collection, etc.
- Many examples out there: Java, Python, PHP

Systems Administrator

- Scripts provided by others (developers, Sun) may be run with confidence
  - > Just need to be mindful of "-w" flag or "#pragma D option destructive"
  - > Example: DExplorer (http://opensolaris.org/os/community/dtrace/dexplorer/)
  - > Example: http://tinyurl.com/afak2
- DTrace requires privelages (dtrace_proc, dtrace_user, dtrace_kernel) which may be given to ordinary (non-root) users, e.g. application experts
  - > /etc/user_attr or the ppriv command

End User

- Debug troublesome applications
  - save as open.d and run “`dtrace -s o.d -q -c mozilla`” to look for failed opens:

```
syscall::open:entry
/progenyof($target)/
{
    self->n=stringof(arg0);
}
syscall::open:return
/self->n!=0 && errno!=0/
{
    printf("%d %s\n", errno, self->n);
    self->n = 0;
}
```

Other Platforms

- Linux: SystemTap/KProbes
  - Still a work in progress
  - Sends machine code to the kernel, rather than byte code
    - Can modify memory, call other routines, loop...
- FreeBSD: DTrace port
  - Being assisted by Solaris DTrace team
  - Might draw more users to OpenSolaris
  - Helps people who won't use OpenSolaris anyway

References

- http://docs.sun.com/app/docs/doc/817-6223
- http://opensolaris.org/os/community/dtrace/

Questions?