Dynamic Tracing in Solaris
Linuxforum 2005

Jørgen Olsen, Torben Hansen

Client Solutions

Sun microsystems

We make the net work.

Why Dynamic Tracing?

- Well-defined techniques for debugging *fatal*, *non-reproducible* failure:
  - Obtain core file or crash dump
  - Debug problem *postmortem* using mdb(1), dbx(1)
- Techniques for debugging *transient* failures are much more ad hoc
  - Typical techniques push traditional tools (e.g. truss (1), mdb(1)) beyond their design centers
  - Many transient problems cannot be debugged at all using extant techniques

Transient failure

- Any unacceptable behavior that does not result in fatal failure of the system
- May be a clear failure:
  - “read(2) is returning EIO on a device that isn't reporting any errors.”
  - “Our application occasionally doesn't receive its timer signal.”
  - “One of our threads is missing a condition variable wakeup.”

Invasive techniques

- If existing tools cannot root-cause transient failure, more invasive techniques must be used
- Typically, custom instrumentation is developed for the failing program and/or the kernel
- The customer reproduces the problem using the instrumented binaries

Invasive techniques, cont.

- Requires either:
  - running instrumented binaries in production

  or
  - reproducing a transient problem in a development environment
- Neither of these is desirable!
- Invasive techniques are slow, error prone, and often ineffective – we must develop a better way

Dynamic instrumentation

- Want to be able to *dynamically* modify the system to record *arbitrary* data
- Must be able to do this on a *production* system
- Must be completely *safe* – there should be no way to induce fatal failure

Introducing DTrace

- Available on stock systems – typical system has more than 25,000 probes
- Dynamically interpreted language allows for arbitrary actions and predicates
- Can instrument at both user-level and kernel-level
- Powerful data management primitives eliminate need for most postprocessing
- Unwanted data is pruned as close to the source as possible

Probes

- A *probe* is a point of instrumentation
- A probe is made available by a *provider*
- Each probe identifies the *module* and *function* that it instruments
- Each probe has a *name*
- These four attributes define a tuple that uniquely identifies each probe
- Each probe is assigned an integer identifier

Providers

- A provider represents a methodology for instrumenting the system
- Providers make probes available to the DTrace framework
- DTrace informs providers when a probe is to be enabled
- Providers transfer control to DTrace when an enabled probe is hit

Providers, cont.

- DTrace has quite a few providers, e.g.:
  - The *function boundary tracing (FBT)* provider can dynamically instrument every function entry and return in the kernel
  - The *syscall* provider can dynamically instrument the system call table
  - The *lockstat* provider can dynamically instrument the kernel synchronization primitives
  - The *profile* provider can add a configureable-rate profile interrupt of to the system
  - ...

Consumers

- A DTrace consumer is a process that interacts with DTrace
- No limit on concurrent consumers; DTrace handles the multiplexing
- Some programs are DTrace consumers only as an implementation detail
- dtrace(1M) is a DTrace consumer that acts as a generic front-end to the DTrace facility

Actions

- *Actions* are taken when a probe fires
- Actions are completely programmable
- Most actions *record* some specified state in the system
- Some actions *change* the state of the system system in a well-defined way
  - These are called *destructive actions*
  - Disabled by default
- Many actions take as parameters expressions in the *D language*

The D language

- D is a C-like language specific to DTrace, with some constructs similar to awk(1)
- Complete access to kernel C types
- Complete access to statics and globals
- Complete support for ANSI-C operators
- Support for strings as first-class citizen
- We'll introduce D features as we need them...

Built-in D variables

- For now, our D expressions will consist only of built-in variables
- Example of built-in variables:
  - `pid` is the current process ID
  - `execname` is the current executable name
  - `timestamp` is the time since boot, in nanoseconds
  - `curthread` is a pointer to the `kthread_t` structure that represents the current thread
  - `probemod`, `probefunc` and `probename` are the current probe's module, function and name

Actions: “`trace`”

- `trace()` records the result of a D expression to the trace buffer
- For example:
  - `trace(pid)` traces the current process ID
  - `trace(execname)` traces the name of the current executable
  - `trace(curthread->t_pri)` traces the `t_pri` field of the current thread
  - `trace(probefunc)` traces the function name of the probe

Actions, cont.

- Actions are indicated by following a probe specification with "`{` *action* `}`"
- For example:

```
dtrace -n 'readch{trace(pid)}'
dtrace -m 'ufs{trace(execname)}'
dtrace -n 'syscall:::entry {trace
  (probefunc)}'
```

- Multiple actions can be specified; they must be separated by semicolons:

```
dtrace -n 'xcalls{trace(pid); trace
  (execname)}'
```

D Scripts

- Complicated DTrace enablings become difficult to manage on the command line
- dtrace(1M) supports *scripts*, specified with the “-s” option
- Alternatively, executable DTrace interpreter files may be created
- Interpreter files always begin with:

```
#!/usr/sbin/dtrace -s
```

Predicates

- *Predicates* allow actions to only be taken when certain conditions are met
- A predicate is a D expression
- Actions will only be taken if the predicate expression evaluates to true
- A predicate takes the form "*/expr/*" and is placed between the probe description and the action

```
#!/usr/sbin/dtrace -s
syscall::open:entry
/execname == "date"/
{ trace(pid); }
```

Actions: More actions

- `tracemem()` records memory at a specified location for a specified length
- `stack()` records the current *kernel* stack trace
- `ustack()` records the current *user* stack trace
- `exit()` tells the DTrace consumer to exit with the specified status

Actions: Destructive actions

- Must specify “-w” option to DTrace
- `stop()` stops the current process
- `raise()` sends a specified signal to the current process
- `breakpoint()` triggers a kernel breakpoint
- `panic()` induces a kernel panic
- `chill()` spins for a specified number of nanoseconds

Output formatting

- The `printf()` function combines the `trace` action with the ability to precisely control output
- `printf` takes a printf(3C)-like format string as an argument, followed by corresponding arguments to print
- e.g.:

```
printf("%d was here", pid);
printf("I am %s", execname);
```

Output formatting, cont.

- Normally, dtrace(1M) provides details on the firing probe, plus any explicitly traced data
- Use the quiet option ("-q") to dtrace (1M) to supress the probe details
- The quiet option may also be set in a D script by embedding:

```
#pragma D option quiet
```

Global D variables

- D allows you to define your own variables that are global to your D program
- Like awk(1), D tries to infer variable type upon instantiation, obviating an explicit variable declaration
- Example:

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
sysinfo:::zfod
{ zfods++; }
profile:::tick-1sec
{ printf("%d zfods\n",  zfods); zfods = 0; }
```

Thread-local D variables

- D allows for *thread-local* variables
- A thread-local variable has the same name – but disjoint data storage – for each thread
- Denoted by prepending "`self->`" to the variable name
- Thread-local variables that have never been assigned in the current thread have the value zero (deallocated by assigning zero to it)

Thread-local D variables, cont.

- Example 1:

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

syscall::poll:entry
{
        self->ts = timestamp;
}

syscall::poll:return
/self->ts && timestamp - self->ts > 1000000000/
{
        printf("%s polled for %d seconds\n", execname,
            (timestamp - self->ts) / 1000000000);
        self->ts = 0;
}
```

Aggregations

- When trying to understand suboptimal performance, one often looks for *patterns* that point to bottlenecks
- When looking for patterns, one often doesn't want to study each datum – one wishes to *aggregate* the data and look for larger trends
- Traditionally, one has had to use conventional tools (e.g. awk(1), perl(1))

Aggregations, cont.

- DTrace supports the aggregation of data as a first class operation
- An *aggregating function* is a function f (x), where x is a set of data, such that:

$$f(f(x_0) \cup f(x_1) \cup \ldots \cup f(x_n)) = f(x_0 \cup x_1 \cup \ldots \cup x_n)$$

- E.g., COUNT, SUM, MAXIMUM, and MINIMUM are aggregating functions; MEDIAN, and MODE are not

Aggregations, cont.

- An *aggregation* is the result of an aggregating function keyed by an arbitrary tuple
- For example, to count all system calls on a system by system call name:

```
dtrace -n 'syscall::::entry \
  { @syscalls[probefunc] = count(); }'
```

- By default, aggregation results are printed when dtrace(1M) exits

Aggregations, cont.

- Aggregations need not be named
- Aggregations can be keyed by more than one expression
- For example, to count all ioctl system calls by both executable name and file descriptor:

```
dtrace -n 'syscall::ioctl:entry \
  { @[execname, arg0] = count(); }'
```

Aggregations, cont.

- Some other aggregating functions:
  - `avg()`: the average of specified expressions
  - `min()`: the minimum of specified expressions
  - `max()`: the maximum of specified expressions
  - `quantize()`: power-of-two distribution of specified expressions
- For example, distribution of write(2) sizes by executable name:

```
dtrace -n 'syscall::write:entry \
  { @[execname] = quantize(arg2); }'
```

Exploring DTrace

- This has been just an introduction to DTrace – there's much, much more:
  - BEGIN, END probes
  - Normalization
  - Associative arrays
  - User-level tracing
  - Speculative tracing
  - Postmortem tracing
  - Explicit versioning
  - Aggregation formatting
  - Provider specifics
  - Clause-local variables
  - Ring buffering
  - Anonymous tracing
  - Privilege model
  - Well-defined stability

Exploring DTrace, cont.

- *Much* more information available at:
  http://opensolaris.org
- Bryan Cantrill's blog:
  http://blogs.sun.com/roller/page/bmc
- Adam Lebenthal's blog:
  http://blogs.sun.com/ahl
- BigAdmin has a page and discussion forum dedicated to DTrace:
  http://www.sun.com/bigadmin/content/dtrace

The DTrace Revolution

- DTrace tightens the diagnosis loop: *hypothesis* → *instrumentation* → *data gathering* → *analysis* → *hypothesis*
- Tightened loop effects a revolution in the way we diagnose transient failure
- Focus can shift from *instrumentation* stage to *hypothesis* stage:
  - Much *less* labor intensive, less error prone
  - Much *more* brain intensive
  - *Much* more effective! (And a *lot* more fun)

Jorgen.Olsen@sun.com

Torben.Hansen@sun.com

Sun microsystems

We make the net work.