



Dynamic Tracing Introduction

Jim Fiori
Technical Specialist
Sun Microsystems, Inc.



Agenda

- Learning Goals
- Background – Why Dynamic Tracing?
- Introducing DTrace
- DTrace concepts, with lab exercises
- Wrap-up
- DTrace Resources
- Q&A

Learning Goals

- Understand the motivation for DTrace
- Learn the basic concepts and terminology
- Apply these concepts by writing your first DTrace scripts through lab exercises
- As a result of this session, you should:
 - Be ready to dive deeper into DTrace by reading the documentation and writing your own DTrace scripts

Why Dynamic Tracing?

- Well-defined techniques exist 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 existing techniques

Definition of 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 misses a condition variable wakeup.

Transient failure, continued

- Failure may be based on customer's definition of unacceptable :
 - “We were expecting to accommodate 100 users per CPU and we're able to get no more than 60 users per CPU.”
 - “Every morning from about 9am to about 10:30am, the system is a dog.”
- We must identify the performance inhibitors - and either eliminate them or reset the customer's expectations

Debugging transient failure

- Historically, we have debugged transient failure using process-centric tools:
 - `truss(1)`, `pstack(1)`, `prstat(1)`, etc.
- These tools were not designed to debug *systemic* problems
- But the tools designed for systemic problems, (i.e., `mdb(1)`) are designed for *postmortem analysis*...

Postmortem techniques

- One technique is to use postmortem analysis to debug transient problems by inducing fatal failure during period of transient failure.
- Better than nothing, but not by much:
 - Requires inducing fatal failure, which nearly always results in more downtime than the transient failure.
 - Requires a keen intuition to be able to identify a dynamic problem from a static snapshot of state

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 (i.e. LD_PRELOAD)
- The customer reproduces the problem using the instrumented binaries

Invasive techniques, continued

- 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

- Dynamic tracing framework introduced in Build 43 of S10.
 - <http://www.sun.com/softwareexpress>
- Available on stock systems – Typical system has more than 30,000 probes.
- Dynamically interpreted language allows for arbitrary actions and predicates.
- Can instrument at both user- and kernel-level.

Introducing DTrace, continued

- Powerful data management primitives eliminate need for most postprocessing.
- Unwanted data is pruned as close to the source as possible.
- Mechanism to trace during boot.
- Mechanism to retrieve all data from a kernel crash dump.

This presentation

- An introduction to the basics of DTrace, with exercises.
- Not designed to be a comprehensive tutorial or to act as reference material - the *Solaris Dynamic Tracing Guide* serves these roles.
- After this introduction, you should feel comfortable diving straight into the *Solaris Dynamic Tracing Guide*.

Probes

- A *probe* is a point of instrumentation.
- A *probe*:
 - Is made available by a *provider*.
 - Identifies the *module* and *function* that it instruments.
 - Has a *name*.
 - Is assigned a integer identifier.
- A *probe* is uniquely identified by its *provider:module:function:name*

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, continued

- 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 configurable-rate profile interrupt of to the system

Providers, continued

- DTrace has quite a few providers, e.g.:
 - The *vminfo* provider can dynamically instrument the kernel “vm” statistics, used by commands such as `vmstat`
 - The *sysinfo* provider can dynamically instrument the kernel “sys” statistics, used by commands such as `mpstat`
 - The *pid* provider can dynamically instrument application code, such as any function entry and return point (actually any instruction!)
 - The *io* provider can dynamically instrument disk I/O events
 - And more!

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.
- `lockstat(1M)` and `plockstat(1M)` are consumers

Listing probes

- Probes can be listed with the “-l” option to `dtrace(1M)`
- Can list probes
 - from a specific provider with “-P *provider*”
 - in a specific module with “-m *module*”
 - in a specific function with “-f *function*”
 - with a specific name with “-n *name*”
- A probe is defined as follows:
provider:module:function:name

Exercise: Listing probes

- Use `dtrace(1M)` to list all available probes
 - How many probes are available?
 - What are the different providers?
 - Which provider provides the greatest number of probes?
- List probes:
 - in the “read” function.
 - in the “ufs” module.
 - with the “xcalls” name
 - List probes from the “sysinfo” provider

Fully specifying probes

- To specify multiple components of a probe, separate the components with a colon.
- Empty components match anything.
- For example, “syscall::open:entry” specifies a probe:
 - from the “syscall” provider.
 - in any module.
 - In the “open” function.
 - named “entry”.

Enabling probes

- Enabled a probe by specifying it without the “-l” option.
- When enabled in this way, probes are enabled with the *default* action.
 - The default action will indicate only that the probe fired; no other data will be recorded.
- For example, “dtrace -m nfs” enables every probe in the “nfs” module

Exercise: enabling probes

- Enable probes in the “random” module.
- Enable probes provided by the “syscall” provider.
- Enable probes named “zfod”.
- Enable probes provided by the “syscall” provider in the “open” function.
- Enable the entry probe in the “clock” function.

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 in a well-defined way.
 - These are called destructive actions.
 - Destructive actions are disabled by default.

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 support for ANSI-C operators.
- Complete access to statics and globals.
- 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.
 - *tid* is the current thread ID.
 - *execname* is the current executable name.
 - *timestamp* is the time since boot, in nanoseconds.
 - *probeprov*, *probemod*, *probefunc* and *probename* are the current probe's provider, 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(probefunc)* traces the function name of the probe.

Actions: continued

- 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)}'*

Exercise: Actions

- Trace the executable name in every poll(2) system call.
- Trace the PID in every entry to the pagefault function.
- Trace the timestamp in every entry to the clock function.

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`

D scripts, continued

- Basic structure of a D script:

```
probe description (provider:module:function:name)  
/ predicate /  
{  
    action statements  
}
```

- For example, a script to trace the executable name upon entry of each system call:

```
#!/usr/sbin/dtrace -s  
syscall:::entry  
{  
    trace(execname);  
}
```

Predicates – (AKA Conditionals)

- *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 “*/expression/*” and is placed between the probe description and the action.

Predicates, continued

- For example, tracing the pid of every process named “date” that performs an open(2):

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

Exercise: Predicates

- Trace the timestamp in every `ioctl(2)` from processes named `dtrace`.
- Use the `arg0` variable to trace the executable name of every process reading from file descriptor 0.

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- Use the `arg2` variable to trace the executable name of every process writing more than 100 bytes

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

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.
 - Java stack displayed when process is a 1.5+ jvm.
- *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.

Exercises: more Actions

- Use `dtrace(1M)` to record a kernel stack in the “zfod” probe.
- Modify the above to record a user-stack.
- Write a D script to stop any process named “xcalc” that performs an `ioctl(2)`.
- Modify the above to record the process ID of the process being stopped.

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\n”, pid);
 - *printf* (“I am %s\n”, execname);

Output formatting, continued

- Normally, `dtrace(1M)` provides details on the firing probe, plus any explicitly traced data.
- Use the quiet option (“-q”) to `dtrace(1M)` to suppress 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.

Global D variables, continued

- 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.
- By definition, *thread-local* variables eliminate the race conditions associated with global variables.
- Denoted by prepending “*self->*” to the variable name

Thread-local D variables, cont.

- Thread-local variables that have never been assigned in the current thread have the value zero.
- Underlying thread-local storage for a thread-local variable is 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;
}
```

Thread-local D variables, cont.

- Example 2:

```
syscall::ioctl:entry  
/execname == date /  
{  
    self->follow = 1;  
}  
fbt::  
/self->follow/  
{}
```

```
syscall::ioctl:return  
/self->follow/  
{  
    self->follow = 0;  
}
```

Exercise: D variables

- Write a D script to trace the executable name and amount of time spent in every `open(2)`.
- Write a D script to follow a `brk(2)` system call through the kernel when called by a `date(1)` command.
- Add “*#pragma D option flowindent*” to the above and observe the change in output.

Aggregations

- When trying to understand suboptimal performance, one often looks for patterns that point to bottleneck.
- 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, continued

- 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:
 - $f(f(x_0) \cup f(x_1) \cup \dots \cup f(x_n)) = f(x_0 \cup x_1 \cup \dots \cup x_n)$
- e.g. COUNT, SUM, MAXIMUM, and MINIMUM are aggregating functions; MEDIAN and MODE are not

Aggregations, continued

- 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, continued

- 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, continued

- 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); }'
```

Exercise: Aggregations

- Count the number of system calls by executable name.
- Count the number of write system calls by process ID.
- Get a distribution of read(2) sizes by executable name and file descriptor.
- Get a distribution of time spent in read(2) by executable name

PID provider

- Allows dynamic instrumentation of applications, regardless of the compiler used or compile flags
- Example, list all function entry probes in 'xclock'

```
# xclock -update 1&  
[2] 778  
# dtrace -l -n pid778:::entry |wc -l  
8396
```

- Example – print the arguments to the 'DrawSecond' function within 'xclock'

```
#!/usr/sbin/dtrace -s  
#pragma D option quiet  
pid$1::DrawSecond:entry  
{  
    printf("len is %d, width = %d, offset = %d, tick_units = %d\n",  
    arg1, arg2, arg3, arg4);  
}  
# ./d.d 778  
len is 64, width = 3, offset = 53, tick_units = 252  
...
```

IO provider

- Allows dynamic instrumentation of physical disk I/O's. Can Show how file system buffering and read-ahead/write-behind is working
- Example, show the device, program, I/O size, I/O type, and file name of all disk I/O, and a summary

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
BEGIN{printf("%-10s %10s %10s %3s %s\n", "Device", "Program", "I/O Size", "R/W", "Path");}
io:::start {
    printf("%-10s %10s %10d %3s %s\n", args[1]->dev_statname, execname,
        args[0]->b_bcount, args[0]->b_flags & B_READ? "R" : "W" , args[2]->fi_pathname);
    @[execname, pid, args[2]->fi_pathname] = sum(args[0]->b_bcount);
}
END { printf("%-10s %8s %10s %s\n", "Program", "PID", "Total", "Path");
    printa("%-10s %8d %10@d %s\n", @);
}
```

Device	Program	I/O Size	R/W	Path
cmdko	mkfile	8192	W	/export/home/foo
cmdko	mkfile	49152	W	/export/home/foo

...

Program	PID	Total	Path
mkfile	813	10493952	/export/home/foo

MIB provider

- Allows dynamic instrumentation of all MIB counters such as those reported by 'netstat -s' and 'kstat'
- Example, show TCP connections per second

```
# !/usr/sbin/dtrace -s
#pragma D option quiet
mib:::tcpActiveOpens,mib:::tcpPassiveOpens
{
    @[execname, probefunc, probename] = sum(arg0);
    event = 1;
}
tick-1sec
/event/
{
    printa("%20s %20s %20s %@ d\n",@);
    trunc(@);
    event = 0;
}
```

- Warning – user context not available on many probes

Exercise: PID Provider

- Start up an 'xclock' process:
 `% xclock -update 1 &`
- Count the number of “entry” probes in xclock
- Write a script to probe on the entry point of the “DrawSecond()” function
- Print the 2nd, 3rd, 4th, and 5th arguments to “DrawSecond()”. These are integer arguments:
 `length, width, offset, tick_units.`
- Resize xclock and watch the arguments change

Exploring DTrace

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

Wrap-up - 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)

Exploring Dtrace

- BigAdmin has a page and discussion forum dedicated to DTrace:
<http://www.sun.com/bigadmin/content/dtrace>
- DTrace Demos
 - /usr/demo/dtrace



Jim Fiori

jim.fiori@sun.com

<http://www.sun.com/softwareexpress>

