



Dtrace: Dynamic Tracing for Solaris

Seán McGrath
Performance Engineer
Sun Microsystems



Traditional Tracing Methods

- Ship optimized binaries
 - Virtually no debugging/tracing support
- Need special versions for tracing
 - Slower, unoptimized
- Reproducing problems
 - Experimental, prone to mis-diagnosis
- Services need to be restarted
 - Can hide the problem

What Is Dtrace?

- **Dynamic tracing system**
 - Code being traced needs no preparation
- **Requires no access to source**
 - Can leverage debugging information
- **Operates on the fly**
 - Probes are inserted without interruption
- **Experiment oriented**
 - Facilitates rapid check, refine, check cycles

How does this help me?

- Safe to use on production systems
- Zero effect when not in use
- Operates outside the application
 - All processes are accessible
 - All kernel space is accessible
- Unprecedented access to system
 - No more black boxes (e.g. Oracle)
- No process can shield itself

How does it work?

- Novel Approach to tracing
 - Probing and processing no longer intertwined like traditional tools
- Providers
 - Suppliers of the probe points
- Processing
 - Performing activities based on provider data

Providers

- Operate independently
 - Allows for differences in perspective
- Many separate providers
 - System calls (application/kernel interface)
 - Function boundary tracing (entry, exit)
 - I/O events (started; done)
 - Profiling
- Solaris kernel has ~30,000 probe points
 - To start with!

Processing Infrastructure

- **Predicates**
 - Filter data based on conditional statements
- **Actions**
 - Store the data
 - Aggregate the data
- **Speculative Tracing**
 - Gather data and decide to keep or drop it based on conditions

Predicates and Actions

- Translated into an intermediate format
 - From the D language
 - DIF instruction set
 - Designed for simplicity of emulation
- Safety Guaranteed
 - Validity is checked upon load
 - No loops (forward branches only)
 - Run-time errors are caught and flagged

The D language

- How most people interact with Dtrace
 - Consists of one or more clauses:
probe-descriptions
/predicate/
{
 action statements
}
- Translated into DIF
 - checked too

The D Language

Example

- `syscall::entry`
/ `execname == "Xorg" /`
{
 `trace(probefunc);`
}
- Compare this with `truss / strace`
 - This would hang the X server if you run it in a terminal window

Future Goals

- Helper providers and actions
 - Middleware related specific information
- Complete Java integration
 - Stack tracing is available; `jstack()`; more to come
- Third party provider support
 - API needs external stability
- on-the-fly debug information access
 - Will to be able to read stabs, dwarf info

Jesus Christ
almighty, it's like they
saw inside my head
and gave me The
One True Tool!

A Slashdotter

Places to go

- OpenSolaris
 - The Dtrace source code for download
- Bryan Cantrill's Blog
 - <http://blogs.sun.com/bmc/>
- Adam Leventhal's Blog
 - <http://blogs.sun.com/ahl/>
- Brendan Gregg's Dtrace Toolkit
 - <http://brendangregg.com/dtrace.html>
- Google for 'Dtrace'



Dtrace: Dynamic tracing for Solaris

sean.mcgrath@sun.com



Data types, operators and variables

- Most data types are accessible
 - Provided by subset of debugging information
 - Currently does not read stabs, dwarf on the fly
- Most kernel (and module) variables
 - Backtick operator (`) makes them external references
- User variables are available
 - Global variables
 - Clause local (this->) and thread local (self->)

Aggregations of data

- Don't care about instances of calls
 - You want to see the shape of the data
- Aggregations are the solution
 - $f(f(x_0) \cup f(x_1) \cup \dots \cup f(x_n)) == f(x_0 \cup x_1 \cup \dots \cup x_n)$
 - Collects the information in situ
 - Uses per-cpu buffers (scalability)
- Certain operations are not suitable
 - Median, mode

Data types, operators and variables

Example

- `syscall::read:entry`
{
 `self->ts = timestamp;`
}
- `syscall::read:return`
{
 `trace(timestamp - self->ts);`
 `self->ts = 0;`
}

Using Aggregations

Example

- `syscall:::entry`
 {
 @[execname, probefunc] = count()
 }
- `syscall::read*:entry`,
 `syscall::write*:entry`
 { @[execname, probefunc] =
 quantize(arg2); }