DTrace: Opening the Kimono

Bryan Cantrill
Solaris Kernel Development
Sun Microsystems
http://blogs.sun.com/bmc

The Problem

- As systems have grown more complex, performance problems are increasingly not seen in a system until it is deployed *in production*...
- ...but performance analysis tools are aimed at the *developer* in *development*
- Production environments left with crude, process-centric tools – of little use on *systemic* problems

Solution Constraints

- Constraints on performance analysis infrastructure in production:
  - Must have zero probe effect when not enabled
  - Must be absolutely safe – accidental misuse must not induce system failure!
- To have systemic scope:
  - *Entire* system must be instrumentable – kernel *and* applications!
  - Must be able to easily prune and coalesce data to highlight systemic trends

The DTrace Solution

- New facility in Solaris for dynamic instrumentation of production systems
- DTrace features:
  - **Dynamic instrumentation:** zero probe effect when disabled
  - **Unified instrumentation:** can instrument both kernel and running apps such that data and control flow can be followed across boundaries
  - **Arbitrary-context kernel instrumentation:** can instrument delicate in-kernel subsystems like synchronization, CPU scheduling

DTrace Features, cont.

- **Data integrity**: if data cannot be recorded for any reason, errors are always reported; absence of errors *guarantees* sound data
- **Arbitrary actions**: actions that can be taken at any point of instrumentation are not defined *a priori*; user can specify arbitrary action
- **Predicates**: predicate mechanism allows actions to only be taken when user-specified conditions are met
- **High-level control language**: predicates and actions are specified in a C-like language that supports all ANSI C operators, allows access to kernel variables and types

DTrace Features, cont.

- **User-defined variables:** support for global and thread-local variables, associative arrays
- **Data aggregation:** scalable mechanism for aggregating data based on an arbitrary tuple
- **Speculative tracing:** mechanism for speculatively record data, deferring the decision to commit or discard the data
- **Heterogeneous instrumentation:** separation of instrumentation methodology from data processing framework allows for disjoint instrumentation techniques

DTrace Features, cont.

- **Scalable architecture**: allows for tens of thousands of probes, provides primitives for efficiently specifying subsets of probes
- **Virtualized consumers**: everything virtualized on a per-consumer basis; no limit on concurrent DTrace consumers
- **Boot-time tracing**: instrumentation can be active during operating system boot
- **Scripting capacity**: DTrace may be used either on the command line via dtrace(1M) or in scripts with a leading "`#!/usr/sbin/dtrace`"

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 4-tuple that uniquely identifies each probe

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 in-kernel DTrace framework when an enabled probe is hit

Providers, cont.

- DTrace has over a dozen 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 *pid* provider can dynamically instrument *any* instruction in *any* running application

Actions and Predicates

- *Actions* are taken when a probe fires
- Actions often record data
- *Predicates* allow actions to only be taken when certain conditions are met
- Actions will only be taken if the predicate expression evaluates to true

The D Language

- Actions and predicates are specified in the D programming language
- D is a C-like language specific to DTrace
  - 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
  - Support for thread-local variables
  - Support for associative arrays
  - ...

D Program Structure

- Consists of one or more *clauses*
- Each clause has the form:

probe-descriptions
/predicate/
{
action-statements
}

- Probes are specified using the form:
  provider:*module*:*function*:*name*
- Omitted fields match any value

D Intermediate Form

- D is compiled at user-level into DIF
- DIF is a small RISC instruction set
- DIF is sent into the kernel, emulated when probe fires
- DIF emulation is completely safe:
  - No backwards branches
  - DIF emulator refuses to perform misaligned loads, divides-by-zero, etc.
  - Invalid loads detected post-load by kernel's fault handler, handled gracefully

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 *n*-tuple
- D syntax for using an aggregation:

  @*identifier*[*keys*] = *aggfunc*(*args*);

- Valid aggfunc:

`count`	`min`	`avg`	`quantize`
`sum`	`max`	`stddev`	`lquantize`

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

Semantic Instrumentation

- Through its various providers, DTrace allows the system to be instrumented nearly arbitrarily...
- ...but making the most use of this requires detailed knowledge of the system's implementation
- We want to instrument the system not in terms of its *implementation*, but in terms of its *semantics*

Execution Semantics

- DTrace allows providers to define the *interface stability* of their probes
- Using statically-defined probes, semantically meaningful points in subsystem execution can be bundled together as a *stable provider*
- Having stable execution semantics is not enough – one must also have stable *data* semantics!

Data Semantics

- Providers can define *translators* that describe the translation from an implementation-dependent structure to an implementation-neutral one
- Probes can have *translated arguments*, allowing for stable data semantics
- Allows providers to not merely reflect the implementation, but to present a semantically stable abstraction above it

Stable Providers

- We have built several stable providers in the kernel:
  - `sched` provider for CPU scheduling
  - `proc` provider for process management
  - `io` provider for I/O
  - `sysinfo` provider for system statistics
  - `vminfo` provider for VM statistics
  - ...

User-level Stable Providers

- The system is *not* merely the kernel!
- Want the *entire system* to be instrumented in ways that have stable, meaningful semantics
- We have infrastructure for user-level system components to define their own stable providers
- Stable providers can be implemented in terms of the user-level statically defined tracing (USDT) provider

Stable Providers

- Many open source projects can benefit from the addition of stable providers
- A stable user-level provider allows this:

```
pid$target::__1cLmysql_parse6FpnDTHD_pcI_v_:entry
{
    @[copyinstr(arg1)] = count();
}
```

To become this:

```
mysql:::query-start
{
    @[args[0]] = count();
}
```

Provider Example: PHP

- Recently (as in, last night) Wez Furlong from the PHP team developed an experimental DTrace provider for PHP
- Exports two probes:

  `function-entry` upon entry to a PHP function

  `function-return` upon return from a PHP function
- Each probe has three arguments:
  - The name of the function
  - The name of the file
  - The line number of the call site

PHP, cont.

- For somone who understands PHP internals, implementing the provider was relatively easy...
- ...and it allows entirely new dimensions of observability into PHP:
  - Allows for the *entire stack* to be understood – from PHP through native library code and into the operating system kernel
  - Allows *systemic* analysis; one can aggregate across multiple PHP processes!
  - Allows use *in production*!

Working on DTrace Itself

- DTrace itself is open source – and there's lots of work still to do...
- Some small-to-medium sized projects:
  - DTrace providers for Perl, Python
  - libdtrace binding for Perl and/or Python
  - libdtrace binding/interface for mdb(1)
  - Improve fault messages to indicate line number of faulting D statement (instead of just DIF offset)
  - `print` action equivalent to mdb's `::print`
  - Floating point support in D
  - Many more – just ask!

Porting DTrace

- DTrace – like all of OpenSolaris – is licensed under the CDDL
- CDDL is a cleaned-up MPL, allowing it to mix with a wide variety of both open source and proprietary systems...
- ...but according to the FSF, restrictions in the GPL prevent mixing CDDL and GPL
- We welcome porting DTrace to other systems – and we're happy to help out

Porting DTrace, cont.

- Porting to a new system would be non-trivial – but by no means impossible
- Necessary expertise:
  - Kernel runtime linker
  - Low-level kernel implementation details (fault handling, cross calls, atomics, etc.)
  - Application debugger infrastructure (process control, symbol lookup, etc.)
  - Encoding for kernel type information
- Porting stable providers would require some additional subsystem expertise

Conclusions

- DTrace is a powerful new facility for *systemic* diagnosis *in production*
- If you're a developer, DTrace will change the way you debug software...
- And by defining your own stable provider, DTrace can become much more useful to *your* users
- There is much work to be done on DTrace itself – contributors welcome!

DTrace Availability

- DTrace is a part of OpenSolaris; source, binaries available at opensolaris.org
- opensolaris.org has a community site dedicated to DTrace:

  http://opensolaris.org/os/community/dtrace

  (Or google “dtrace” + “I'm feeling lucky”)
- Community is quite active; DTrace discussion list has over 500 subscribers!
- Documentation (400+ pages!) available at docs.sun.com

DTrace: Opening the Kimono

Bryan Cantrill
Solaris Kernel Development
Sun Microsystems
http://blogs.sun.com/bmc