Profiling and Debugging Tcl with DTrace

EuroTcl 08

6 June 2008

Daniel A. Steffen

`das@users.sf.net`

Outline

- Introduction to DTrace
- The D Language
- DTrace providers
- The Tcl provider
- Demos

What Is DTrace

- Comprehensive dynamic tracing framework created by Sun
- Available on Solaris 10, OpenSolaris & Mac OS X Leopard
    - Ports to *BSDs underway
- Zero disabled cost: trace production code
- Dynamic: instrumentation added to live code
- System-wide: instrument kernel & userland
- Programmable: ask arbitrary questions, follow your nose!

What DTrace Is Not

- Not intended to replace existing sampling/profiling tools
- Not magic, must understand problem in depth
    - Be able to generate sharp hypotheses
- Probe effect: small but not zero
- D language has no flow control: no loops or branches
    - only conditional expressions:

```
(expBoolX ? expTrue : expFalse)
```

How DTrace Works

- D language program from `dtrace` or other front-end
- Compiled into intermediate form (DIF) by `libdtrace`
- Bytecode interpreted on DTrace virtual machine in kernel
  - DIF validated in kernel, run-time errors handled safely
- Requested instrumentation dynamically added to kernel/ userland program text, removed again when tracing ends
- Tracing data captured in-kernel, passed out to userland for post-processing and output

The D Language

- Lightweight, small, simple
  - Interpreted in kernel, with interrupts disabled
  - No flow control: no loops or branches
  - No user-defined functions
  - Variable declarations are optional
- Structure of a D program:
  - Probe clauses
  - Declarations (variables, types, providers) and `#pragmas`

Probe Clauses

```
probe descriptions
/ predicate /
{
  action statements
}
```

- Basic unit of the D language
- Lazy: missing elements inferred
  - Default/empty action prints probe identifier
- Probe descriptions are , and action statements ; separated

Probe Descriptions

```
provider:module:function:name
  dtrace:::BEGIN, dtrace:::END, tick-1sec
  syscall::write*:entry
  pid*::printf:return
  tcl*:::proc-entry
```

- Describes events of interest
- Supports wildcards: `*`, blank field, missing field
- Only required part of a probe clause
- flowindent: indents matching `*entry` & `*return` output

Demo

DTrace CLI Exploration

```
# dtrace -F -n 'syscall:::'

# dtrace -F -n 'syscall::: {trace(execname)}'

# dtrace -F -n 'syscall::: /execname!="dtrace"
    && execname!="Terminal"/ {trace(execname)}'

# dtrace -n 'syscall:::entry
    {@[execname] = count()}'

# dtrace -n 'syscall:::entry
    /execname=="Safari"/
    {@[probefunc] = count()}'

# dtrace -n 'syscall::gettimeofday:entry
    /execname=="Safari"/
    {@[ustack(8)] = count()}'
```

Predicates

```
syscall::stat64:entry /execname=="Safari"/
syscall::open:entry /copyinstr(arg0)=="x"/
pid*::printf:return /self->tracing && --n/
tcl*:::cmd-entry /copyinstr(arg0)=="puts"/
```

- Expression evaluated at probe firing time
    - non-zero/no predicate: action statements are executed
    - zero: no statements executed, no trace data recorded
        - cheap, but more expensive than probe not firing at all
- Focus probe actions on data of interest

Action Statements

```
{trace(execname)}
{printf("(%s) 0x%p", probemod, arg0)}
{this->args = (Tcl_Obj**)(arg1?
    copyin(arg2,sizeof(Tcl_Obj*)*arg1):0);
    self->ts = vtimestamp}
```

- Arithmetic/relational/logical/bit operations like in C
- Assign to/read from variables (built-in and user defined)
- Call built-in actions and subroutines:
  - `trace(),printf(),ustack(),copyinstr()`

Variables

```
x, self->x, this->x, execname, `kmem_flags
x=123; a["xyz"]=456; b["w",9]=1; p[3]='?';
@[execname]=count(); @[tid,arg0]=count();
  @tot[k]=sum(arg1); @[k]=max(stackdepth);
  @time[k]=quantize(timestamp-self->ts);
```

- Global, thread-local, clause-local, built-in, external
- Scalars, associative arrays, scalar arrays
- Aggregations: no need to store entire data set
  - can only be assigned *aggregating functions*

Types

```
char, long, int32_t, uintptr_t, double
typedef struct s {char c[2]; long *p;} s;
    s *x = &y; x->c[1] = 'a'; *(x->p) = 1L
((struct proc *)p)->p_pid;
```

- Fundamental types like in C, plus `string`
- `struct`, `union`, `typedef`, `enum` like in C
- Pointer, structure and array access like in C
  - Protection from invalid pointer access
- Kernel types known, for userland `#include` std C headers

DTrace Providers

dtrace	BEGIN, END, ERROR
profile	profile-100hz, profile-10s, tick-1s
fbt	sock_connect:entry, copystr:return
vminfo	vm_fault:cow_fault, vm_pageout:swapout
syscall	readlink:entry, mkdir:return
lockstat	lck_mtx_lock:adaptive-block
proc	fork:create, sendsig:signal-handle
io	buf_biowait:wait-start, buf_biodone:done
pid	pid123:libc:fprintf:entry, pid*::myfn:9f
USDT	tcl*:::cmd-entry, ruby*:::function-entry

Tcl DTrace Provider

- Added in 8.4.16 and 8.5b1
  - `http://wiki.tcl.tk/DTrace`
- Uses USDT and *is-enabled* probes: disabled probe-sites cost a branch and a few noops
- Similar information available for tracing as with TIP280, `tcl_traceCompile` and `tcl_traceExec`
  - But support can be enabled in production Tcl builds
  - Plus have system-wide tracing & other DTrace advantages
- Configure with `--enable-dtrace`

Tcl Provider Probes

Probes		arg0	arg1	arg2	arg3	argN
`proc-entry`	`cmd-entry`	`name`	`objc`	`objv`		
`proc-return`	`cmd-return`	`name`	`code`			
`proc-args`	`cmd-args`	`name`	`arg`	`arg`	`arg`	`arg`
`proc-result`	`cmd-result`	`name`	`code`	`res`	`resO`	
`proc-info`	`cmd-info`	`cmd`	`type`	`proc`	`file`	`line`
`inst-start`	`inst-done`	`name`	`stkN`	`stkT`		
`obj-create`	`obj-free`	`obj`				
`tcl-probe`	*`[tcl::dtrace]`*	`arg`	`arg`	`arg`	`arg`	`arg`

Retrieving Tcl_Obj Args

```
tcl*:::proc-entry, tcl*:::cmd-entry {
  this->args = arg1 ? (Tcl_Obj**)copyin(arg2,
      sizeof(Tcl_Obj*) * arg1) : NULL;
  this->i = 0;
  this->o = arg1 > this->i &&
      *(this->args + this->i) ? (Tcl_Obj*)
      copyin((user_addr_t)*(this->args +
      this->i), sizeof(Tcl_Obj)) : NULL;
  this->s0 = this->o ? (this->o->bytes ?
      copyinstr((user_addr_t)this->o->bytes,
      maxstrlen) : lltostr(this->o->
      internalRep.longValue)) : ""; }
```

TclDTrace

- Google Summer of Code project
  - Student: Remigiusz Modrzejewski
  - Mentor: DAS, Backup Mentors: Jeff Hobbs, Tomasz Kosiak
- Implement a Tcl binding to `libdtrace`
  - Run D scripts and process results directly from Tcl
  - Tk visualization of tracing output
- `http://dev.lrem.net/tcldtrace/`

Future Ideas

- `proc-return`, `cmd-return`: pass arg(s) with info from `Tcl_GetReturnOptions()` dict
- `obj-settype`: intrep mutation, shimmering
- `event-entry`, `event-return`
- `Tcl_Obj*` translator
- Tracing of commands executed via `TclCompEvalObj()`
- TclOO tracing
- Other suggestions? File a bug on SF

Further Reading & Tools

- `http://wiki.tcl.tk/DTrace`
- `http://www.opensolaris.org/os/community/dtrace/`
- Solaris Dynamic Tracing Guide
  - `http://dlc.sun.com/osol/docs/content/DYNMCTRCGGD/dynmctrcggd.html`
- `http://opensolaris.org/os/community/dtrace/dtracetoolkit/`
- `http://developer.apple.com/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide`

No OS With DTrace ?

1. Download VirtualBox VM: free & open-source
   - `http://www.virtualbox.org/`
   - x86 hardware running Windows, Linux, Mac OS X
2. Download & Install OpenSolaris 2008.05:
   - `http://www.opensolaris.com/`
   - 680MB LiveCD download, free & open-source
3. Profit

Demos

Thanks

`http://categorifiedcoder.info/`