Advanced DTrace
Tips, Tricks and Gotchas

Bryan Cantrill, Mike Shapiro and
Adam Leventhal
Team DTrace

Sun microsystems

Advanced DTrace

- Assumption that the basics of DTrace are understood – or at least familiar
- You need not have used DTrace to appreciate this presentation...
- ...but the more you have, the more you'll appreciate it
- In no particular order, we will be describing some *tips*, some *tricks* and some *gotchas*

DTrace Tips

- Tips are pointers to facilities that are (for the most part) fully documented
- But despite (usually) being well-documented, they might not be well-known...
- This presentation will present these facilities, but won't serve as a tutorial for them; see the documentation for details

DTrace Tricks

- There are a few useful DTrace techniques that are not obvious, and are not particularly documented
- Some of these “tricks” are actually workarounds to limitations in DTrace
- Some of these limitations are being (or will be) addressed, so some tricks will be obviated by future work

DTrace Gotchas

- Like any system, DTrace has some pitfalls that novices may run into – and a few that even experts may run into
- We've tried to minimize these, but many remain as endemic to the instrumentation problem
- Several of these are documented, but they aren't collected into a single place

Tip: Stable providers

- Allow meaningful instrumentation of the kernel without requiring knowledge of its implementation, covering:
  - CPU scheduling (`sched`)
  - Process management (`proc`)
  - I/O (`io`)
  - Some kernel statistics (`vminfo`, `sysinfo`, `fpuinfo`, `mib`)
  - More on the way…
- If nothing else, read the documentation chapters covering them!

Tip: Speculative tracing

- DTrace has a well-known *predicate* mechanism for conditional execution
- This works when one knows *at probe-firing* whether or not one is interested
- But in some cases, one only knows *after the fact*
- Speculative tracing is a mechanism for speculatively recording data, committing it or discarding it at a later time

Tip: Normalizing aggregations

- Often, one wishes to know not absolute numbers, but rather per-unit rates (e.g. system calls per second, I/O operations per transaction, etc.)
- In DTrace, aggregations can be turned into per-unit rates via *normalization*
- Format is "`normalize(@`*agg*`,` *n*`)`," where *agg* is an aggregation and *n* is an arbitrary D expression

Tip: `clear` and `tick` probes

- `clear` zeroes an aggregation's values
- With `tick` probes, `clear` can be used to build custom monitoring tools:

```
io:::start
{
        @[execname] = count();
}

tick-1sec
{
        printa("%40s %@d\n", @);
        clear(@);
}
```

Trick: Valueless `printa`

- `printa` takes a format string and an aggregation identifier
- "`%@`" in the format string denotes the aggregation value
- This is *not* required; you can print *only* the aggregation tuple
- Can be used as an implicit `uniq(1)`
- Can be used to effect a global ordering by specifying `max(timestamp)` as the aggregating action

Tip: `stop`

- One may wish to stop a process to allow subsequent investigation with a traditional debugger (e.g. DBX, MDB)
- Do this with the `stop` destructive action:

```
#pragma D option destructive

io:::start
/execname == “java”/
{
    printf(“stopping %d...”, pid);
    stop();
}
```

Trick: Conditional breakpoints

- Existing conditional breakpoint mechanisms are limited to pretty basic conditions
- The `stop` action and the `pid` provider allow for much richer conditional breakpoints
- For example, breakpoint based on:
  - Return value
  - Argument value
  - Latency
  - …

Gotcha: `stop` gone haywire

- Be very careful when using `stop` – it's a destructive action for a reason!
- If you somehow manage to stop every process in the system, the system will effectively be wedged
- If a `stop` script has gone haywire, try:
  - Setting `dtrace_destructive_disallow` to `1` via `kmdb(1)`/OBP
  - Waiting for deadman to abort DTrace enabling, then remotely logging in (hoping that `inetd` hasn't been stopped!)

Gotcha: Running into limits

- If you try to enable very large D scripts (hundreds of enablings and/or thousands of actions), you may find that DTrace rejects it:

```
dtrace: failed to enable './biggie.d': DIF
  program exceeds maximum program size
```

- This can be worked around by tuning `dtrace_dof_maxsize` in `/etc/system` or via "`mdb -kw`"
- Default size is 256K

Tip: Verbose error messages

- For a more verbose error message when DOF is rejected by the kernel, set `dtrace_err_verbose` to `1`
- A more verbose message will appear on the console and in the system log:

```
# ./biggie.d
dtrace: failed to enable './biggie2.d': DIF
  program exceeds maximum program size
# tail -1 /var/adm/messages
Feb  9 17:55:57 pitkin dtrace: [ID 646358
  kern.warning] WARNING: failed to process DOF:
  load size exceeds maximum
```

Gotcha: Enabling `pid123:::`

- When using the pid provider, one usually wants to instrument function entry and return
- The pid provider *can* instrument every instruction
- If you specify "`pid123:::`" it will attempt to instrument every instruction in process 123!
- This *will* work – but you may be waiting a while...

Gotcha: Too many `pid` probes

- `pid` probes are created on-the-fly as they are enabled
- To avoid denial-of-service, there is a limit on the number of pid probes that can be created
- This limit (250,000 by default) is low enough that it can be hit for large processes:

```
dtrace: invalid probe specifier pid123:::: failed
  to create probe in process 123: Not enough space
```

Tip: Allowing more `pid` probes

- Increase `fasttrap-max-probes` in `/kernel/drv/fasttrap.conf`
- After updating value, either reboot or:
  - Make sure DTrace isn't running
  - Unload all modules ("`modunload -i 0`")
  - Confirm that `fasttrap` is not loaded ("`modinfo | grep fasttrap`")
  - Run "`update_drv fasttrap`"
  - New value will take effect upon subsequent DTrace use

Gotcha: Misuse of `copyin`

- `copyin` can copy in an arbitrary amount of memory; it returns a *pointer* to this memory, *not* the memory itself!
- This is the **incorrect** way to dereference a user-level pointer to a `char *`:

```
trace(copyinstr(copyin(arg0,
  curpsinfo->pr_dmodel == PR_MODEL_ILP32 ? 4 : 8))
```

- This is what was meant:

```
trace(copyinstr(*(uintptr_t *)copyin(arg0,
  curpsinfo->pr_dmodel == PR_MODEL_ILP32 ? 4 : 8)
```

Gotcha: Buffer drops

- There is always the possibility of running out of buffer space
- This is a consequence of instrumenting arbitrary contexts
- When a record is to be recorded and there isn't sufficient space available, the record will be *dropped*, e.g.:

```
dtrace: 978 drops on CPU 0
dtrace: 11 aggregation drops on CPU 0
```

Tip: Tuning away buffer drops

- Every buffer in DTrace can be tuned on a per-consumer basis via `-x` or `#pragma D option`
- Buffer sizes tuned via `bufsize` and `aggsize`
- May use size suffixes (e.g. `k`, `m`, `g`)
- Drops may also be reduced or eliminated by increasing `switchrate` and/or `aggrate`

Gotcha: Dynamic variable drops

- DTrace has a finite dynamic variable space for use by thread-local variables and associative array variables
- When exhausted, subsequent allocation will induce a *dynamic variable drop*, e.g.:

```
dtrace: 103 dynamic variable drops
```

- These drops are often caused by failure to zero dead dynamic variables
- *Must be eliminated for correct results!*

Tip: Tuning away dynamic drops

- If a program correctly zeroes dead dynamic variables, drops must be eliminated by tuning
- Size tuned via the `dynvarsize` option
- In some cases, “dirty” or “rinsing” dynamic variable drops may be seen:

```
dtrace: 73 dynamic variable drops with non-empty
  dirty list
```

- These drops can be eliminated by increasing `cleanrate`

Trick: `ftruncate` and `trunc`

- `ftruncate` truncates standard output if output has been redirected to a file
- Can be used to build a monitoring script that updates a file (e.g., webpage, RSS feed)
- Use with `trunc` on an aggregation with a `max(i++)` action and a valueless `printa` to have “last *n*” occurences in a single file

Trick: Tracking object lifetime

- Assign `timestamp` to an associative array indexed on memory address upon return from `malloc`
- In entry to `free`:
  - Predicate on non-zero associative array element
  - Aggregate on stack trace
  - `quantize` current time minus stored time
- Note: eventually, long-lived objects will consume all dynamic variable space

Trick: Rates over time

- For varying workloads, it can be useful to observe changes in rates over time
- This can be done using `printa` and `clear` out of a `tick` probe, but output will be by time – not by aggregated tuple
- Instead, aggregate with `lquantize` of current time minus start time (from `BEGIN` enabling) divided by unit time

Tip: Using `system`

- Use the `system` action to execute a command in response to a probe
- Takes `printf`-like format string and arguments:

```
#pragma D option quiet
#pragma D option destructive

io:::start
/args[2]->fi_pathname != “<none>” &&
  args[2]->fi_pathname != “<unknown>”/
{
    system(“file %s”, args[2]->fi_pathname);
}
```

Gotcha: Using `system`

- `system` is processed at *user-level* – there will be a delay between probe firing and command execution, bounded by the `switchrate`
- Be careful; it's easy to accidentally create a positive feedback loop:

```
dtrace -n 'proc:::exec
  {system("/usr/ccs/bin/size %s", args[0])}'
```

- To avoid this, add a predicate to above:

```
/!progenyof($pid)/
```

Trick: `system("dtrace")`

- In DTrace, actions cannot enable probes
- However, using the `system` action, one D script can launch another
- If instrumenting processes, steps can be taken to eliminate lossiness:
  - `stop` in parent
  - Pass the stopped process as an argument to the child script
  - Use `system` to `prun`(1) in a `BEGIN` clause in the child script

Tip: `-c` option

- To observe a program from start to finish, use "`-c` *cmd*"
- `$target` is set to target process ID
- `dtrace` exits when command exits

```
# dtrace -q -c date
  -n 'pid$target::malloc:entry{@ = sum(arg0)}'
  -n 'END{printa("allocated %@d bytes\n", @)}'
Fri Feb 11 09:09:30 PST 2005
allocated 10700 bytes

#
```

Gotcha: Stripped user stacks

- When using the `ustack` action, addresses are translated into symbols as a *postprocessing* step
- If the target process has exited, symbol translation is impossible
- Result is a stripped stack:

```
# dtrace -n syscall::::entry'{ustack()}'
CPU     ID                    FUNCTION:NAME
  0    363                  resolvepath:entry
              0xfeff34fc
              0xfefe4faf
              0x80474c0
```

Tip: Avoiding stripped stacks

- With the "`-p` *pid*" option, `dtrace` attaches to the specified process
- `dtrace` will hold the target process on exit, and perform all postprocessing before allowing the target to continue
- Limitation: you must know *a priori* which process you're interested in

Trick: Using `stop` and `ustack`

- If you don't know *a priori* which processes you're interested in, you can use a `stop`/`system` trick:
  - `stop` in `syscall::rexit:entry`
  - `system("prun %d", pid);`
- Any user stacks processed before processing the `system` action will be printed symbolically
- This only works if the application calls `exit`(2) explicitly!

Gotcha: Slow user stacks

- If neither `-p` or `-c` is specified, process handles for strack symbol translation are maintained in an LRU *grab cache*
- If more processes are being `ustack`'d than handles are cached, user stack postprocessing can be slowed
- Default size of grab cache is eight process handles; can be tuned via `pgmax` option

Tip: Ring buffering and `-c/-p`

- Problem: program repeatedly crashes, but for unknown reasons
- Use ring buffering by setting `bufpolicy` to `ring`
- Ring buffering allows use on long-running processes
- For example, to capture all functions called up to the point of failure:

```
dtrace -n 'pid$target:::entry'
  -x bufpolicy=ring -c cmd
```

Gotcha: Deadman

- DTrace protects against inducing too much load with a *deadman* that aborts enablings if the system becomes unresponsive:

```
dtrace: processing aborted: Abort due to systemic
  unresponsiveness
```

- Criteria for responsiveness:
  - Interrupt can fire once a second
  - Consumer can run once every thirty seconds
- On a heavily loaded system, a deadman timeout may *not* be due to DTrace!

Tip: Tuning the deadman

- If the deadman is due to residual load, the deadman may simply be disabled by enabling destructive actions
- Alternatively, the parameters for the deadman can be explicitly tuned:
  - `dtrace_deadman_user` is user-level reponsiveness expectation (in nanoseconds)
  - `dtrace_deadman_interval` is interrupt responsiveness expectation (in nanoseconds)
  - `dtrace_deadman_timeout` is the permitted length of unresponsiveness (in nanoseconds)

Trick: Stack filtering

- Often, one is interested in a probe only if a certain function is on the stack
- DTrace doesn't (yet) have a way to filter based on stack contents
- You can effect this by using thread-local variables:
  - Set the variable to “1” when entering the function of interest
  - Predicate the probe of interest with the thread-
  - Don't forget to clear the thread-local variable!

Trick: Watchpoints via `pid`

- Problem: you know which data is being corrupted, but you don't know by whom
- Potential solution: instrument every instruction, with `stop` action and predicate that data is incorrect value
- Once data becomes corrupt, process will stop; attach a debugger (or use `gcore`(1)) to progress towards the root-cause...

Trick: Measuring DTrace

- Can exploit two properties of DTrace:
  - Clause-local variables retain their values across multiple enablings of the same probe in the same program
  - The `timestamp` variable is cached for the duration of a clause, but not across clauses
- Requires three clauses:
  - Assign `timestamp` to clause-local in 1st clause
  - Perform operation to be measured in 2nd clause
  - Aggregate on difference between `timestamp` and clause-local in 3rd clause

Trick: Iterating over structures

- To meet safety criteria, DTrace doesn't allow programmer-specified iteration
- If you find yourself wanting iteration, you probably want to use aggregations
- In some cases, this may not suffice...
- In some of these cases, you may be able to effect iteration by using a `tick-`*n* probe to increment an indexing variable...

Gotcha: Unsporting libraries

- Regrettably, on x86 there are compiler options that cause the compiler to not store a frame pointer
- This is regrettable because these libraries become undebuggable: stack traces are impossible
- Library writers: *don't do this*!
  - gcc: Don't use `-fomit-frame-pointer`!
  - Sun compilers: avoid `-xO4`; it does this by default!

Gotcha: Unsporting functions

- Some compilers put jump tables in-line in program text
- This is a problem because data intermingled in program text confuses text processing tools like DTrace
- DTrace always errs on the side of caution: if it becomes confused, it will refuse to instrument a function
- Most likely to encounter this on x86
- Solution to this under development…

Gotcha: Unsporting apps

- Some applications have stripped symbol tables and/or static functions
- Makes using the `pid` provider arduous
- Can still use the `pid` provider to instrument instructions in stripped functions by using "–" as the probe function and the address of the instruction as the name:

```
# dtrace -n pid123::-:80704e3
dtrace: description 'pid123::-:80704e3' matched 1
  probe
```

Trick: `sizeof` and profiling

- `sizeof` historically works with types and variables
- In DTrace, `sizeof` (*function*) yields the number of bytes in the function
- When used with `profile` provider, allows function profiling:

```
profile-1234hz
/arg0 >= `clock &&
    arg0 <= `clock + sizeof (`clock)/
{
    ...
}
```

Trick: Using GCC's preprocessor

- `-C` option uses /usr/ccs/lib/cpp by default, a cpp from Medieval Times
- Solaris 10 ships gcc in /usr/sfw/bin so a modern, ANSI cpp is available with some limitations (#line nesting broken)
- To use GCC's cpp:

```
# dtrace -C -xcpppath=/usr/sfw/bin/cpp -Xs -s a.d
```

- Needed when .h uses ANSI-isms like ##
- Also useful for M4 propeller-heads

Gotcha: `$target` evaluation

- When using the `-c` option, the child process is created and stopped, the D program is compiled with `$target` set appropriately, and the child is resumed
- By default, the child process is stopped immediately before the `.init` sections are executed
- If instrumenting the linker or a library, this may be too late – or too early

Tip: Tuning `$target` evaluation

- Exact "time" of D program evaluation can be tuned via the `evaltime` option
- `evaltime` option may be set to one of the following:
  - `exec:` upon return from exec(2) (first instruction)
  - `preinit:` before `.init` sections run (default)
  - `postinit:` after `.init` sections run
  - `main:` before first instruction of `main()` function

Gotcha: Data model mismatch

- By default, D compiler uses the data model of the **kernel** (ILP32 or LP64)
- This may cause problems if including header files in instrumenting 32-bit applications on a 64-bit kernel
- Alternate data model can be selected using `-32` or `-64` options
- If alternate model is specified, kernel instrumentation won't be allowed

Gotcha: Enabled probe effect

- When enabled, DTrace (obviously) has a non-zero probe effect
- In general, this effect is sufficiently small as to not distort conclusions...
- However, if the time spent in DTrace overwhelms time spent in underlying work, time data will be distorted!
- For example, enabling both `entry` and `return` probes in a short, hot function

Tip: Sample with `profile`

- When honing in on CPU time, use the `profile` provider to switch to a sample-based methodology
- Running with high interrupt rates and/or for long periods allows for *much* more accurate inference of cycle time
- Aggregations allow for easy profiling:
  - Aggregate on sampled PC (`arg0` or `arg1`)
  - Use "%a" to format kernel addresses
  - Use "`%A`" (and `-p`/`-c`) for user-level addresses

Trick: Higher-level profiling

- In interrupt-driven probes, `self->` denotes variables in the *interrupt* thread, not in the *underlying* thread
- Can't use interrupt-driven probes and predicate based on thread-local variables in the underlying thread
- Do this using an associative array keyed on `curlwpsinfo->pr_addr`
- Can use this to profile based on higher-level units (e.g. transaction ID)

Gotcha: `vtimestamp`

- `vtimestamp` represents the number of nanoseconds that the current thread has spent on CPU since some arbitrary time in the past
- `vtimestamp` factors out time spent in DTrace – the *explicit* probe effect
- There is no way to factor out the *implicit* probe effect: cache effects, TLB effects, etc. due to DTrace
- Use the absolute numbers carefully!

Gotcha: Fixed-length strings

- D string type behaves like this C type:

```
typedef struct {
        char s[n]; /* -xstrsize=n, default=256 */
} string;
```

- Implications:
  - You always allocate the maximum size
  - You always copy by value, not by reference
  - String assignment silently truncates at size limit
- Using strings as an array key or in an aggregation tuple is suboptimal if other types of data are available

Tip: Demo DTrace scripts

- `/usr/demo/dtrace` contains all of the example scripts from the documentation
- `index.html` in that directory has a link to every script, along with the chapter that contains it
- DTrace demo directory is installed by default on all Solaris 10 systems

Team DTrace

Bryan Cantrill
Mike Shapiro
Adam Leventhal
dtrace-core@kiowa.eng.sun.com

Sun microsystems