

# Solaris 10 Boot Camp Dtrace Overview

**Bob Netherton**

Solaris Adoption, US Client Solutions

Sun Microsystems

# Why Dynamic Tracing?

- Well-defined techniques 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 extant techniques

# 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 is missing a condition variable wakeup.”

# Transient failure, cont.

- May be a failure 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.”
- In these situations, it is up to *someone* to understand 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 suss out 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
- The customer reproduces the problem using the instrumented binaries

# Invasive techniques, cont.

- 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

# DTrace

## Solaris Dynamic Tracing – An Observability Revolution

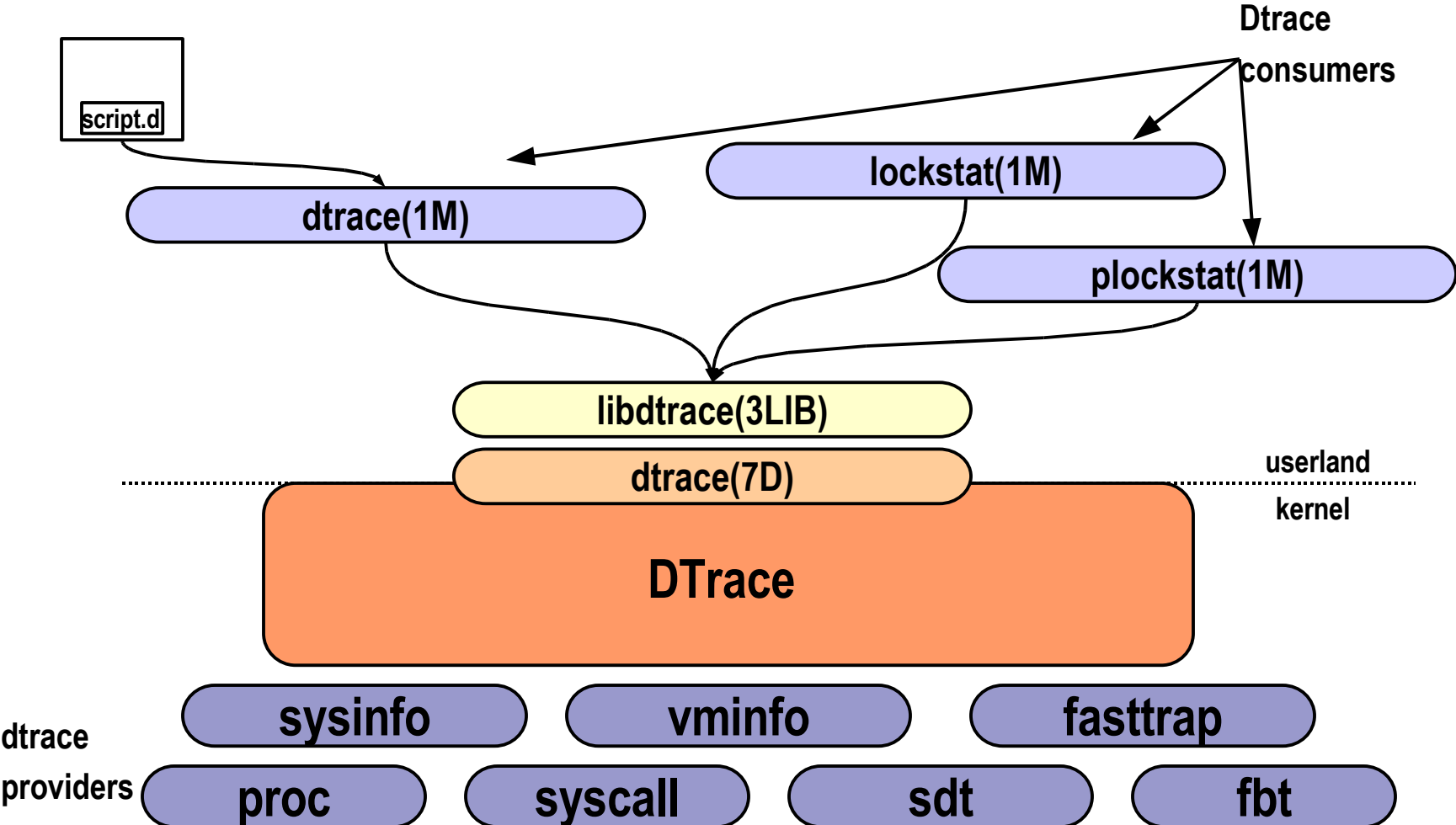
- Seamless, *global* view of the system from user-level thread to kernel
- Not reliant on pre-determined trace points, but *dynamic instrumentation*
- Data *aggregation* at source minimizes postprocessing requirements
- Built for live use on *production* systems
  - > Completely safe
  - > No loops – so no halting problem!!!!

# DTrace

## Solaris Dynamic Tracing – An Observability Revolution

- Ease-of-use and *instant gratification* engenders serious *hypothesis testing*
- Instrumentation directed by high-level control language (not unlike AWK or C) for easy scripting and command line use
- Comprehensive probe coverage and powerful data management allow for *concise* answers to *arbitrary* questions

# DTrace – The Big Picture



# DTrace Components

- Probes
  - > A point of instrumentation
  - > Has a name (string), and a unique probe ID (integer)
- Providers
  - > DTrace-specific facilities for managing probes, and the interaction of collected data with consumers
- Consumers
  - > A process that interacts with dtrace
  - > typically `dtrace(1)`
- Using dtrace
  - > Command line – `dtrace(1)`
  - > Scripts written in the 'D' language

# 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

*example:* syscall::open:entry

# Providers

- 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!

# Actions

- *Actions* are taken when a probe fires.
- Actions are completely programmable
  - > With constraints (no loops)
- 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.
- Rich set of built-in variables
- Anonymous arrays
- Complete access to statics and globals.
- Support for strings as first-class citizen.
- We'll introduce D features as we need them...

# 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);  
}
```

# D script template: state machine

```
#!/usr/sbin/dtrace -s
#pragma D option quiet

syscall::open:entry
{
    self->ts = timestamp;
}
syscall::open:return
/self->ts/
{
    printf("open took %d nanosecs\n",timestamp - self->ts);
    self->ts = 0;
}
```

# 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.

# Aggregations

- 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.
  - > Can be formatted with BEGIN/END blocks, just like `awk`

# 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); }'`

# Aggregations, continued

```
# dtrace -n 'syscall::write:entry { @[execname] = quantize(arg2); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
```

^C

gnome-terminal

value	----- Distribution -----	count
0		0
1	@@@@	1
2		0
4		0
8		0
16		0
32	@@@@@@@	2
64		0
128		0
256	@@@@@@@@@@@@@@@@@@@@	5
512		0
1024	@@@@@@@	2
2048	@@@@	1
4096		0

# General Approaches

- Start with existing tools
  - > `truss`
  - > `vmstat/mpstat/iostat/lockstat`
  - > `pfiles/pmap/pldd/pstack/ptree`
  - > `prstat -mL`
- Understand the new tools in Solaris 10:
  - > `intrstat`
  - > `plockstat` (DTrace consumer)
  - > `pstack` (for Java)
  - > `pfiles` (now shows file names)

# General Approaches

- You will type "dtrace" thousands of times
- Often you will hit a dead-end
- Think like Edison...
- Use the manual!
  - > Install it locally!
  - > And the examples in `/usr/demo/dtrace`

# General Approaches

- Use aggregations to make sense of large amounts of data
- look for "outliers" and "holes"
- Use `quantize()` / `lquantize()`
  - > `min()` / `max()` / `avg()` can hide important data
- User these providers
  - > `profile`, `sched`, `pid`, `proc`, and `io`, (less of `fbt`)
- Be aware of probe effect on `pid`

# General Approaches (cont.)

- A good starting place is `mpstat`
  - > User/system time ratios?
  - > context switching?
  - > lock contention?
  - > cross calls?
  - > faults?
  - > system call levels?
  - > Interrupts?

# Scenario: High User Time

## Code: multi.c (interest rate model)

```
# mpstat 1
```

CPU	minf	mjf	xcal	intr	ithr	csw	icsw	migr	smtx	srw	syscl	usr	sys	wt	idl
0	0	0	0	342	240	146	60	0	0	0	229	100	0	0	0
0	0	0	0	335	233	144	60	0	0	0	181	100	0	0	0

- Where is this utilization coming from?
- Use the profile provider. (small probe effect)
- Examples:

```
# dtrace -n 'profile-1001{ @[arg1] = count()}'
or
# dtrace -n 'profile-1001/pid == 1234/{ @[arg1] = count()}'
or
# dtrace -n 'profile-1001/execname == "multi"/{ @[arg1] = count()}'
or
# dtrace -n 'profile-1001{@[execname, ustack(1)] = count()}'

# dtrace -n 'profile-1001{@[execname] = count()}'
```

# Scenario: High User Time

Profile a user application: look at ustack.

```
# dtrace -n 'profile-1001/execname == "multi"/{@[ustack(1)] = count()} \
          END{trunc(@,10)}'
```

```
^C
```

```
CPU      ID          FUNCTION:NAME
  0       2                :END
```

```
libm.so.2`exp+0x4
```

```
94
```

```
libm.so.2`exp+0x54
```

```
99
```

```
libm.so.2`exp+0x67
```

```
111
```

```
libm.so.2`exp+0x98
```

```
116
```

```
multi`intio_calc_n_month_rate+0x8f
```

```
117
```

```
multi`intio_calc_n_month_rate+0x186
```

```
138
```

```
libm.so.2`exp+0x10d
```

```
163
```

```
libm.so.2`exp+0x90
```

```
184
```

```
libm.so.2`exp+0x61
```

```
617
```

```
libm.so.2`log+0x1a
```

```
854
```

**ustack(1) does not add much overhead**

# Scenario: High User Time

- Clearly, this application is sitting in math libraries
- Non-Production Probing:
  - > We'd like a function count, and the time spent in each function
  - > (Warning - can be a significant probe effect!!!)

```
# cat quantify.d
#!/usr/sbin/dtrace -s
pid$1:::entry
{
    self->ts[self->stack++] = timestamp;
}

pid$1:::return
/ self->ts[self->stack - 1] /
{
    this->elapsed = timestamp - self->ts[--self->stack];
    @[probefunc] = count();
    @a[probefunc] = quantize(this->elapsed);
    self->ts[self->stack] = 0;
}
```

# Scenario: High User Time

```
# ./quantify.d
intio_calc_pmms_rate          16
intio_calc_adj_rate          81
intio_calc_adjustment        82
intio_calc_n_month_rate      34655
log                           39380
exp                           78763
intio_calc_adjustment
```

```
value ----- Distribution ----- count
2048 | 0
4096 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 82
8192 | 0
```

log

```
value ----- Distribution ----- count
2048 | 0
4096 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 39337
8192 | 19
16384 | 12
32768 | 9
65536 | 0
131072 | 2
262144 | 0
524288 | 1
1048576 | 0
```

# Scenario: High User Time

exp

value	Distribution	count
2048		0
4096	@@@	78685
8192		37
16384		26
32768		9
65536		2
131072		3
262144		0
524288		1
1048576		0

intio\_calc\_pmms\_rate

value	Distribution	count
16777216		0
33554432	@@@	16
67108864		0

intio\_calc\_adj\_rate

value	Distribution	count
2097152		0
4194304	@@@@@@@@@@@@@@@@@@@@@@@@@@@@@	33
8388608		0
16777216		0
33554432	@@@	48
67108864		0

# Scenario: High User Time

- Another view of the application....

```
# cat watchpid.d
#!/usr/sbin/dtrace -Fs
pid$1:::entry{}
pid$1:::return{}

# ./watchpid.d `pgrep multi`
0      -> intio_calc_n_month_rate
0      -> log
0      <- log
0      -> log
0      <- log
0      -> exp
0      <- exp
0      -> exp
0      <- exp
0      -> exp
0      <- exp
0      -> exp
0      <- exp
0      -> exp
0      <- exp
0      <- intio_calc_n_month_rate
0      -> intio_calc_n_month_rate
0      -> log
0      <- log
...
```

Look for best `exp()`, `log()`, Possibly inline

# Scenario: High User Time

- Other things to watch for:
  - > Calls to `.mul()`, `.div()`, etc for SPARC
    - > compiled for SPARC V7 - recompile!
  - > Excessive calls to libraries such as:
    - > `getenv(3C)`, `getrusage(3C)`, `getrlimit(2)`
    - > `time(2)`, `gettimeofday(3C)`
      - possibly replace with `gethrtime()`
      - `time(2)` on SPARC is very expensive! (7X)
      - `time(2)` on Intel is somewhat expensive! (1.5X)
      - `gettimeofday(3C)` slightly more expensive than `gethrtime(3C)` (10%)
  - > Watch for `memmove(3C)` vs `memcpy(3C)`
    - > Use `memcpy()` if regions do not overlap

# Scenario: System time being consumed (>10%, or when usr/sys ratio approaches 2-to-1)

```
# vmstat 1
kthr      memory          page        disk        faults      cpu
r  b  w    swap  free  re  mf  pi  po  fr  de  sr  cd  s0  s1  --  in   sy   cs   us  sy  id
0  0  0  1157600  751464  3188  24781  3937  0  0  0  0  433  0  0  0  1236  15801  1543  32  41  28
0  0  0  1157292  752688  2738  17618  4578  0  0  0  0  309  0  0  0  995  11657  1289  29  29  43
0  0  0  1157156  752476  4450  40268  2116  0  0  0  0  306  0  0  0  987  24698  1608  26  63  11
```

- Profile the system, see who is asking for resources:

```
# dtrace -n 'profile-1001 {@[execname,uid,pid,tid] = count()}'
grep          0      11239      1          2
grep          0      11101      1          2
grep          0      12155      1          2
grep          0      11617      1          2
grep          0      12457      1          3
grep          0      11916      1          3
dtrace        0      10453      1          3
grep          0      11506      1          4
grep          0      11488      1          4
Xorg          5667      408        1          5
gnome-terminal 5667      660        1          7
sched         0         0          1         19
find          0         804        1         509
sched         0         0          0        1573
```

# Scenario: System time being consumed

- Profile the kernel, see what kernel functions are hit:

```
# dtrace -n 'profile-1001 /arg0/{@[arg0] = count()}END \
      {trunc(@,10);printa("%a %@u\n", @)}'
^C
CPU      ID          FUNCTION:NAME
  0       2          :END unix`hwblkpagecopy+0xca 27
unix`page_numtopp_nolock+0x29 29
unix`ddi_get8+0x13 36
unix`cas64+0x1a 46
unix`fakesoftint_return+0x2 59
unix`page_exists+0x3f 64
unix`page_lookup_create+0x5a 66
unix`page_lookup_nowait+0x45 81
unix`mutex_enter+0x11 241
unix`cpu_halt+0x9d 660
```

# Scenario: System time being consumed

- What are the most frequently called kernel functions?

```
# dtrace -n 'fbt:::entry {@[probefunc] = count()}END{trunc(@,10)}'
      (can be expensive)
```

CPU	ID	FUNCTION:NAME	
0	2	:END	
		htable_e2va	30397
		page_pptonum	40784
		x86_hm_exit	50531
		x86_hm_enter	50531
		htable_va2entry	53327
		x86pte_access_pagetable	53706
		x86pte_release_pagetable	53706
		apic_setspl	54169
		psm_get_cpu_id	55707
		page_next_scan_large	67220

- Demand paging related work.

# Scenario: System calls (>100 per second)

```
% vmstat 1
```

kthr			memory		page				disk				faults		cpu						
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	cd	s0	s1	--	in	sy	cs	us	sy	id
0	0	0	1158040	759184	950	3939	1480	0	1	0	55	84	0	0	0	549	3143	577	6	9	85
0	0	0	1102740	720324	7	42	25	0	0	0	0	5	0	0	0	403	327	244	2	0	98
0	0	0	1102740	720324	0	0	0	0	0	0	0	0	0	0	0	370	261	202	1	1	98

- System calls show interface between application and OS

```
# dtrace -n 'syscall:::entry/pid != $pid/{@[execname,probefunc] = count()}'
```

```
...
```

gnome-terminal	pollsys	44
Xorg	read	46
mixer_applet2	ioctl	54
acoread	pollsys	423
acoread	ioctl	840

# Scenario: System calls

- Aggregate on syscall

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

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

```
dtrace -n 'syscall:::entry { @[probefunc,pid] = count()}'
```

- If top syscalls are `read()` / `write()` / `poll()`
  - > aggregate on `arg0` (fd #)
  - > look up file descriptor using `pfiles`

# Scenario: System calls

- Look for system call errors:


```
# dtrace -n 'syscall::return /errno/ {trace(execname);trace(pid);trace(errno)}'
0    318    pollsys:return    Xorg                408                4
0    12     read:return      gnome-terminal      660                11
0    12     read:return      Xorg                408                11
0    12     read:return      dsdm                570                11
0    12     read:return      nautilus            650                11
0    12     read:return      Xorg                408                11
```

- Why not `truss`?
  - > `truss` is much easier to use, provides better information
  - > But watch production use!
  - > `truss` only looks at one process
  - > DTrace is better at looking for system-wide events

# 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)

# For more information on Dtrace

- Solaris 10 Dynamic Tracing Guide  
<http://docs.sun.com/app/docs/doc/817-6223>
-  OpenSolaris Dtrace community  
> <http://opensolaris.org/os/community/dtrace>
- Sample scripts  
`/usr/demo/dtrace`
- Brendan Gregg's Dtrace toolkit  
<http://users.tpg.com.au/adsl4yb/dtrace.html>
- <http://blogs.sun.com>

**Questions ?**

**Thank you!**

# Solaris 10 Boot Camp Dtrace Overview

**Bob Netherton**

[bob.netherton@sun.com](mailto:bob.netherton@sun.com)

<http://blogs.sun.com/bobn>