

DTrace Code Camp: A Deep Dive

Peter Karlsson
Technology Evangelist
Sun Microsystems
<http://opensolaris.org>

Agenda

- Why DTrace
- What is DTrace
- D - the language
- Solaris DTrace Providers
- DTrace and Java
- DTrace resources

Why DTrace?

- Transient problems are hard to debug.
- Example.
 - > Who sent a kill signal to my process
 - > Thread gets preempted when it should not
 - > In live production system my application does not scale above 30,000 user
 - > Why are there so many threads in run queue when the CPU is idle

Current Options

- Reproduce problem outside of production
 - > Not easy & Expensive
- Convert it into a fatal problem
 - > Causes more downtime than the transient problem
 - > Not easy to debug a transient problem with a snapshot
- Use tools like truss or pstack
 - > Per process tools – hard to debug systemic issues
 - > Too heavy weight for production

Current Options

- Custom instrumented application or kernel
 - > Too heavy weight for production
 - > Takes too many iteration to get to the root cause
 - > Huge QA cost
 - > Expensive production interruptions
 - > Very invasive

A Much Better Solution

- **A Dynamically Instrumentable System**
 - > have enough instrumentation to permit collecting any arbitrary data
 - > permit dynamically turning on/off instrumentation
 - > be performant to run in production
 - > ensures safety

DTrace

- Over 30K probes built into Solaris 10
- Can create more probes on the fly
- New powerful, dynamically interpreted language (D) to instantiate probes
- Probes are light weight and low overhead
- **No** overhead if probe not enabled
- Safe to use on “**live**” system

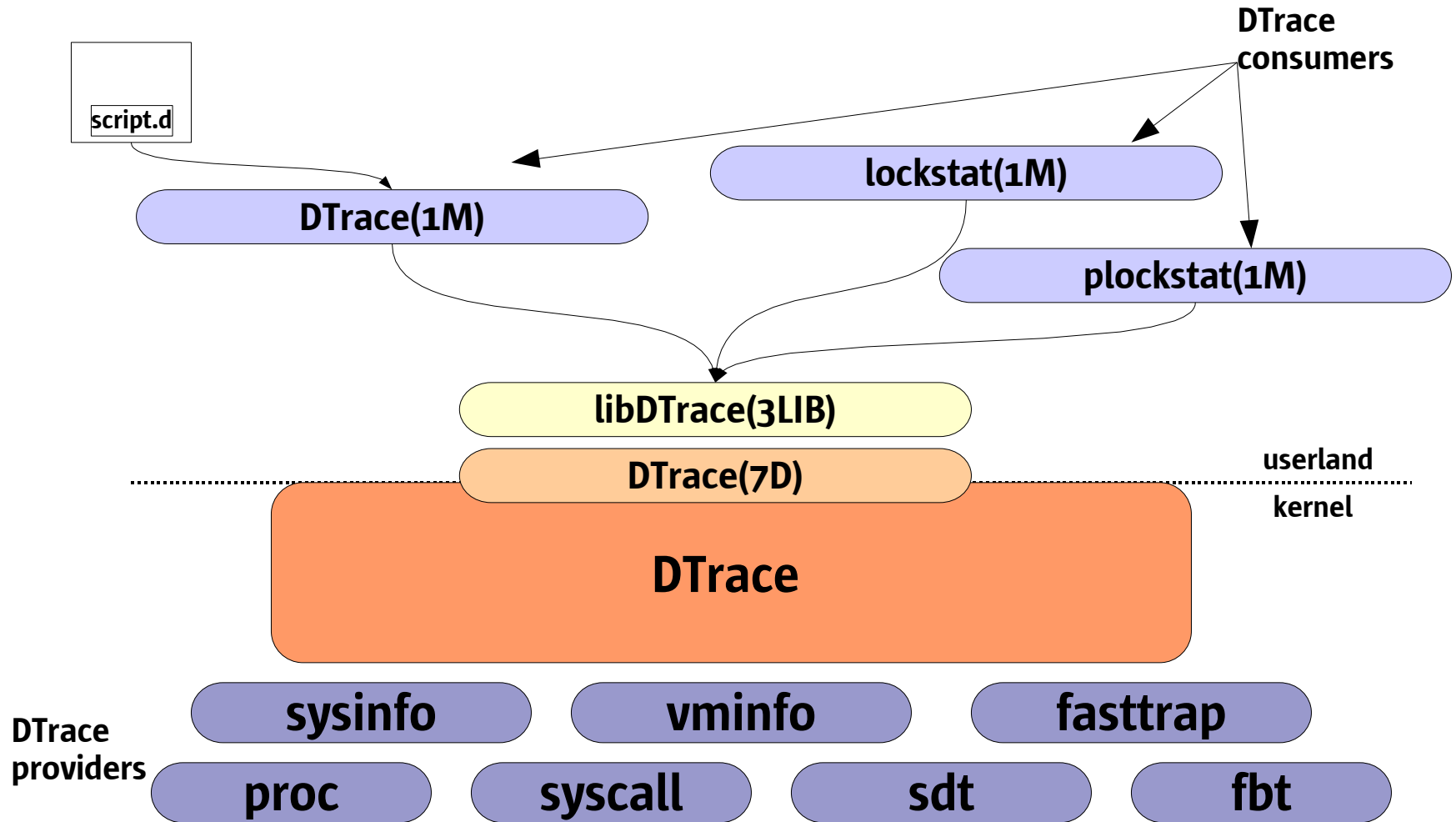


```
# dtrace -l |wc -l  
38485
```

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)

In a nutshell : DTrace architecture



How it works

- dtrace command compiles the D language Script
- Intermediate code checked for safety (like java)
- The compiled code is executed in the kernel by DTrace
- DTrace instructs the provider to enable the probes
- As soon as the D program exits all instrumentation removed
- No limit (except system resources) on number of D scripts that can be run simultaneously
- Different users can debug the system simultaneously without causing data corruption or collision issues

D- Language:

D-Language - Quick overview

The D Language

- DTrace uses new scripting language called “D”
 - > Dynamically interpreted language
- “C”-like language with some constructs like “awk”
- All ANSI C operators
- Global, thread local and probe local variables
- Built-in variables such as pid, cpu, execname and timestamp
- Pointer dereferencing
- Runtime safety - **no SEGV or core dump**
- Associative arrays

Data types & Operators

- D Language provides standard data types.
 - > Integer types(32 bit size, 64-bit size)
 - > char(1,1), short(2,2),int(4,4), long(4,8),long long(8,8)
 - > Float types(32-bit size, 64-bit size)
 - > float(4,4), double(8,8),long double(16,16)
- Integer float and string constants are present in D.
- add(+), sub(-), mul(*), div(/), mod(%) math operators
- less than (<), less than or equal(<=) etc. defined
- logical AND(&&), OR(||) & XOR(^^)
- bitwise operations and standard assignment ops present.

D Language - Format.

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

- When a probe fires then action is executed if predicate evaluates true
- Example, “Print all the system calls executed by ksh”

```
#!/usr/sbin/dtrace -s  
syscall:::entry  
/execname=="ksh"/  
{  
    printf(“%s called\n”,profunc);  
}
```

syscalls.d

DTrace – Example.

```
# dtrace -n BEGIN -n END
```

```
dtrace: description 'BEGIN' matched 1 probe
```

```
dtrace: description 'END' matched 1 probe
```

```
CPU          ID      FUNCTION:NAME
0            1      :BEGIN
^C
0            2      :END
#
```

- **dtrace -n BEGIN -n END**
 - > Enable the probes BEGIN & END
- **Output**
 - > the probes that were enabled
 - > CPU in which probe was executed
 - > ID of probe
 - > Function & name of the fired probe.

Hello World in D

```
#!/usr/sbin/dtrace -s hello.d  
BEGIN  
{  
    printf("Hello World\n");  
    exit(0);  
}  
  
END  
{  
    printf("Goodbye Cruel World\n");  
}
```

- First line similar to any shell script
- Note the -s : “Following lines are a script”
- The script in plain English
 - > “When the BEGIN probe is fired print hello world and exit.”

DEMO

Providers

- Providers represent 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 DTrace when an enabled probe is fired

- *Examples*

*syscall provider provides probes in every system call
fbt provider provides probe into every function in the kernel*

Probe

- Probes are points of **instrumentation**
- Probes are made available by **providers**
- Probes identify the **module** and **function** that they instruments
- Each probe has a name
- These four attributes define a tuple that uniquely identifies each probe

provider:module:function:name

- Example

syscall::open:entry

Listing Probes.

- Probes can be listed with the “-l” option to `dtrace(1M)`
 - > in a specific function with “-f function”
 - > in a specific module with “-m module”
 - > with a specific name with “-n name”
 - > from a specific provider with “-P provider”
- Empty components match all possible probes
- Wild card can be used in naming probe

Predicates

- A predicate is a D expression
- Actions will only be executed if the predicate expression evaluates to true
- A predicate takes the form “/expression/” and is placed between the probe description and the action

- *Examples*

- > Print the pid of every “ls” process that is started

```
#!/usr/sbin/dtrace -s  
proc:::exec-success  
/execname == "ls"/  
{  
}
```

pred.d

Actions

- Actions are executed when a probe fires
- Actions are completely programmable
- Most actions record some specified state in the system
- Some actions change the state of the system in a well-defined manner
 - > These are called destructive actions and are disabled by default.
- Probes may provide parameters that can be used in the actions.

Destructive actions

- We saw that DTrace is safe to use on a live system because there are checks to make sure it does not modify what it observes.
- There are some cases where you want to change the state of the system.
- Example
 - > stop a process to better analyze it.
 - > kill a runaway process
 - > run a process using system
- These actions are disabled by default.
- Use '-w' option to enable it.
- We will see some of these destructive actions.

Destructive actions

- `stop()`
 - > stop the process that fired the probe. Use `prun` to make the process continue. Use `gcore` to capture a core of the process.
- `raise(int signal)`
 - > Send signal to the process that fired the probe.
- `copyout(void *buf, uintptr_t addr, size_t nbytes)`
 - > copy `nbytes` from address pointed by `buf` to `addr` in the current proc's user address space.
- `copyoutstr(string str, uintptr_t addr, size_t maxlen)`
 - > copy string (null terminated) from `str` to `addr` in the current proc's user address space. Max length is `maxlen`.
- `system(string program)`
 - > Similar to system call in C. Run the program. `system` also allows you to use `printf` formats in the string.

Aggregation

- Think of a case when you want to know the “**total**” time the system spends in a function.
 - > We can save the amount of time spent by the function every time it is called and then add the total.
 - > If the function was called 1000 times that is 1000 bits of info stored in the buffer just for us to finally add to get the total.
 - > Instead if we just keep a running **total** then it is just one piece of info that is stored in the buffer.
 - > We can use the same concept when we want **averages, count, min or max.**
- **Aggregation** is a D construct for this purpose.

Aggregates

- Often the patterns are more interesting than each individual sample
- Want to aggregate data to look for trends
- Aggregates as first class operations
- Aggregation is the result of an aggregating function
- Examples:
 - > count()
 - > max(), min(), avg()
 - > quantize()
- May be keyed by an arbitrary tuple

Aggregation - Format

- **@name[keys] = aggfunc(args);**
- '@' - key to show that name is an aggregation.
- keys – comma separated list of D expressions.
- **aggfunc** could be one of...
 - > sum(expr) – total value of specified expression
 - > count() – number of times called.
 - > avg(expr) – average of expression
 - > min(expr)/max(expr) – min and max of expressions
 - > quantize()/lquantize() - power of two & linear distribution

Aggregation Example 1.

aggr.d

```
#!/usr/bin/dtrace -s  
sysinfo:::pswitch  
{  
    @[execname] = count();  
}
```

```
bash-3.00$ ./aggr.d  
dtrace: script './aggr.d' matched 3 probes  
^C
```

soffice.bin	1
dtrace	2
java	4
sched	9

More about aggregations

- `printa(@name)` – print aggregation “name”
 - > `printa` also takes a format string
- `normalize()` - normalize aggregation over time. (ie. average).
 - > data is not lost just a view created
- `denormalize()` - opposite of `normalize`
 - > remove the normalized view
- `clear(@name)` – clear and allow DTrace to reclaim aggregation mem
- `trunc(@name, num)` - truncate value and key after top num entries
 - > A negative num will truncate after bottom num entries
 - > Typically we are only interested in the top /bottom functions

Aggregation example

```
#!/usr/sbin/dtrace -s
syscall::mmap:entry
{
    @a["number of mmaps"] = count();
    @b["average size of mmaps"] = avg(arg1);
    @c["size distribution"] = quantize(arg1);
}
```

```
profile:::tick-10sec
{
    printa(@a);
    printa(@b);
    printa(@c);

    clear(@a);
    clear(@b);
    clear(@b);
}
```

Calculating time spent

- One of the most common request is to find time spent in a given function
- Here is how this can be done

```
#!/usr/sbin/dtrace -s
syscall::open*:entry,
syscall::close*:entry
{
    ts=timestamp;
}
```

```
syscall::open*:return,
syscall::close*:return
{
    timespent = timestamp - ts;
    printf("ThreadID %d spent %d nsecs in %s", tid, timespent, probefunc);
    ts=0; /*allow DTrace to reclaim the storage */
    timespent = 0;
}
```

Whats wrong with this??

Thread Local Variable

- self->variable = expression;
 - > self – keyword to indicate that the variable is thread local
 - > A boon to multi-threaded debugging
 - > As name indicates this is specific to the thread.
 - > See code re-written

```
#!/usr/sbin/dtrace -s
syscall::open*:entry,
syscall::close*:entry
{
    self->ts=timestamp;
}

syscall::open*:return,
syscall::close*:return
{
    timespent = timestamp - self->ts;
    printf("ThreadID %d spent %d nsecs in %s", tid, timespent, probefunc);
    self->ts=0; /*allow DTrace to reclaim the storage */
    timespent = 0;
}
```

Built-in Variable

- Here are a few built-in variables.

arg0 ... arg9 – Arguments represented in int64_t format

args[] - Arguments represented in correct type based on function

cpu – current cpu id

cwd – current working directory

errno – error code from last system call

gid, uid – real group id, user id

pid, ppid, tid – process id, parent proc id & thread id

probeprov, probemod, probefunc, probename - probe info

timestamp, walltimestamp, vtimestamp – time stamp nano sec from an arbitrary point and nano sec from epoc

External Variable

- DTrace provides access to kernel & external variables.
- To access value of external variable use `

```
#!/usr/sbin/dtrace -qs
dtrace:::BEGIN
{
    printf("physmem is %d\n", `physmem);
    printf("maxusers is %d\n", `maxusers);
    printf("ufs:freebehind is %d\n", ufs`freebehind);
    exit(0);
}
```

ext.d

- Note: ufs`freebehind indicates kernel variable freebehind in the ufs module
- These variables cannot be lvalue. They cannot be modified from within a D Script

DTrace Providers

Providers

- Here is the list of providers
 - > **dtrace Provider** – provider probes related to DTrace itself
 - > **lockstat Provider** – lock contention probes
 - > **profile Provider** – probe for firing at fixed intervals
 - > **fbt Provider** – function boundary tracing provider
 - > **syscall Provider** – probes at entry/return of every syscall
 - > **sdt Provider** – “statically defined probes” user definable probe
 - > **sysinfo Provider** – probe kernel stats for mpstat and sysinfo tools
 - > **vminfo Provider** – probe for vm kernel stats
 - > **proc Provider** – process/LWP creation and termination probes
 - > **sched Provider** – probes for CPU scheduling

Providers - cont.

- We will now see some more details on a few Solaris Providers
 - > **io Provider** – provider probes related to disk IO
 - > **mib Provider** – probes in network layer in kernel
 - > **fpuinfo Provider** – probe into kernel software FP processing
 - > **pid Provider** – probe into any function or instruction in user code.
 - > **plockstat Provider** – probes user level sync and lock code
 - > **fasttrap Provider** – *non-exposed probe*.
 - > Fires when DTrace probes are called

fbt Provider

- The *fbt* – *Function Boundary Tracing* provider has probe into most functions in the kernel.
- Using fbt probe you can track entry and return from almost every function in the kernel.
- There are over 20,000 fbt probe in even the smallest Solaris systems
- You'd need Solaris internal knowledge to be able to use this effectively
- Once opensolaris.org has entire Solaris code you will be able to use these probes more effectively.
- Very useful if you develop your own kernel module.
- We will see a few examples.

fbt probe example.

The following example prints all the kernel functions called by `ioctl` syscall from a “bash” shell.

```
#!/usr/sbin/dtrace -Fs  
/*-F provides nice indented printing */
```

```
syscall::ioctl:entry  
/execname == "bash"/  
{  
    self->traceme = 1;  
    printf("fd: %d", arg0);  
}
```

```
fbt::  
/self->traceme/  
{}
```

```
syscall::ioctl:return  
/self->traceme/  
{  
    self->traceme = 0;  
    exit(0);  
}
```

fbt1.d

profile Provider

- Profile providers has probes that will fire at regular intervals.
- These probes are not associated with any kernel or user code execution
- profile provider has two probes. profile probe and tick probe.
- format for profile probe: profile-n
 - > The probe will fire n times a second on every CPU.
 - > An optional ns or nsec (nano sec), us or usec (microsec), msec or ms (milli sec), sec or s (seconds), min or m (minutes), hour or h (hours), day or d (days) can be added to change the meaning of 'n'.

profile probe - examples

- The following example prints out frequency at which proc execute on a processor.

```
#!/usr/sbin/dtrace -qs
```

```
profile-100
```

```
{
    @procs[pid, execname] = count();
}
```

prof.d

- This one tracks how the priority of process changes over time.

```
#!/usr/sbin/dtrace -qs
```

```
profile-1001
```

```
/pid == $1/
```

```
{
    @proc[execname]=lquantize(curlwpsinfo->pr_pri,0,100,10);
}
```

prio.d

- try this with a shell that is running...

```
$ while true ; do i=0; done
```

tick-n probe

- Very similar to profile-n probe
- Only difference is that the probe only fires on one CPU.
- The meaning of “n” is similar to the profile-n probe.

proc Provider

- The proc Provider has probes for **process/lwp lifecycle**
 - > **create** – fires when a proc is created using fork and its variants
 - > **exec** – fires when exec and its variants are called
 - > **exec-failure** & **exec-success** – when exec fails or succeeds
 - > **lwp-create**, **lwp-start**, **lwp-exit** – lwp life cycle probes
 - > **signal-send**, **signal-handle**, **signal-clear** – probes for various signal states
 - > **start** – fires when a process starts before the first instruction is executed.

Examples

- The following script prints all the processes that are created. It also prints who created these process as well.

```
#!/usr/sbin/dtrace -qs
proc:::exec
{
    self->parent = execname;
}

proc:::exec-success
/self->parent != NULL/
{
    @[self->parent, execname] = count();
    self->parent = NULL;
}

proc:::exec-failure
/self->parent != NULL/
{
    self->parent = NULL;
}

END
{
    printf("%-20s %-20s %s\n", "WHO", "WHAT", "COUNT");
    printa("%-20s %-20s %>@d\n", @);
}
```

proc1.d

More Examples

- The following script prints all the signals that are sent in the system. It also prints who sent the signal to whom.

```
#!/usr/sbin/dtrace -qs
proc:::signal-send
{
    @[execname, stringof(args[1]->pr_fname),args[2]] = count();
}

END
{
    printf("%20s %20s %12s %s\n", "SENDER", "RECIPIENT", "SIG", "COUNT");
    printa("%20s %20s %12d %@ d\n", @);
}

```

proc2.d

```
$ ./proc2.d
^C
```

SENDER	RECIPIENT	SIG	COUNT
sched	dtrace	2	1
sched	ls	2	1
sched	ksh	18	4
sched	ksh	2	5
ksh	ksh	2	5
ksh	ksh	20	12

DTrace and User Process

- DTrace provides a lot of features to probe into the user process
- We will look at features in DTrace that is useful when we examine user process
- Some examples of using DTrace in user code will be discussed

The pid Provider

- The **pid** Provider is extremely flexible and allowing you to instrument any instruction in **user land** including entry and exit
- pid provider creates probes on the fly when they are needed
- This is why they **do not** appear in the dtrace -l listing
- We will see how to use the pid provider to trace
 - > Function Boundaries
 - > Any arbitrary instruction in a given function

pid – Function Boundary probes

- The probe is constructed using the following format
`pid<processid>:<library or executable>:<function>:<entry or return>`
- Examples:
`pid1234:date:main:entry`
`pid1122:libc:open:return`
- Following code counts all libc calls made by a program

```
#!/usr/sbin/dtrace -s  
pid$target:libc::entry  
{  
    @[probfunc]=count()  
}
```

pid1.d

pid – Function Offset probes

- The probe is constructed using the following format
`pid<processid>:<library or executable>:<function>:<offset>`
- Examples:
`pid1234:date:main:16`
`pid1122:libc:open:4`
- Following code prints all instructions executed in the programs main

```
#!/usr/sbin/dtrace -s  
pid$target:a.out:main:  
{  
}
```

offs.d

Action & Subroutines

- There are a few actions and subroutines in DTrace that helps us examine user land applications
 - > `ustack(<int nframes>, <int strsize>)` - records user process stack
 - > `nframes` – specifies the number of frames to record
 - > `strsize` – if this is specified and non 0 then the address to name
 - translation is done when the stack is recorded into a buffer of `strsize`. This will avoid problem with address to name translation in user land when the process may have exited
 - For java code analysis you'd need Java 1.5 to use this `ustack()` functionality (see example)

DTrace and Java

DTrace and Java

- Ustack() and jstack()
- Java 1.4.2 and 1.5
- Mustang (Java 1.6)

ustack

```

$ ./ustk.d -c "java -version"
dtrace: script './ustk.d' matched 1 probe
java version "1.5.0_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_01-b08)
Java HotSpot(TM) Client VM (build 1.5.0_01-b08, mixed mode, sharing)
CPU   ID           FUNCTION:NAME
 0   13           write:entry
      libc.so.1`_write+0x8
      libjvm.so`JVM_Write+0xb8
      libjava.so`0xfe99f580
      libjava.so`Java_java_io_FileOutputStream_writeBytes+0x3c
      java/io/FileOutputStream.writeBytes
      java/io/FileOutputStream.writeBytes
      java/io/FileOutputStream.write
      java/io/BufferedOutputStream.flushBuffer
      java/io/BufferedOutputStream.flush
      java/io/PrintStream.write
      sun/nio/cs/StreamEncoder$CharsetSE.writeBytes
      sun/nio/cs/StreamEncoder$CharsetSE.implFlushBuffer
      sun/nio/cs/StreamEncoder.flushBuffer
      java/io/OutputStreamWriter.flushBuffer
      java/io/PrintStream.write
      java/io/PrintStream.print
      java/io/PrintStream.println
      sun/misc/Version.print
      0xf8c05764
      0xf8c00218
      libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_pnMmethodHandle_pnRJavaCallArguments_pnGThread__v_+0x548
      libjvm.so`jni_CallStaticVoidMethod+0x4a8
      java`main+0x824
      java`_start+0x108

```

ustack1.d

```

#!/usr/sbin/dtrace -s
syscall::write:entry
/pid == $target/
{
    ustack(50,500);
}

```

jstack() action

- jstack action prints mixed mode stack trace
- Both java frames and native (C/C++) frames are shown
- Only JVM versions 5.0_01 and above are supported
- jstack shows hex numbers for JVM versions before 5.0_01
- Example (usejstack.d)

```
#!/usr/sbin/dtrace -s
syscall::pollsys:entry
/ pid == $1 / {
    jstack(50);
}
```

- Integer argument to limit the number of frames shown

Java 1.4.2 and 1.5

The dvm provider

- java.net project to add DTrace support in 1.4.2 and 1.5
<https://solaris10-dtrace-vm-agents.dev.java.net/>
- Download shared libs
 - > **libdvm_{ti}.so** → java 1.5
 - > **libdvm_{pi}.so** → java 1.4.2
- Add location of libs to LD_LIBRARY_PATH variable
- Set JAVA_TOOL_OPTIONS to -Xrundvmti:all
- Name of provider - “**dvm**”

dvm provider - probes

- Here is a list of dvm probes and their signatures
 - > `vm-init()`
 - > `vm-death();`
 - > `thread-start(char *thread_name);`
 - > `thread-end();`
 - > `class-load(char *class_name);`
 - > `class-unload(char *class_name);`
 - > `gc-start();`
 - > `gc-finish();`
 - > `gc-stats(long used_objects, long used_object_space);`
 - > `object-alloc(char *class_name, long size);`
 - > `object-free(char *class_name);`
 - > `method-entry(char *class_name, char *method_name, char *method_signature);`
 - > `method__return(char *class_name, char *method_name, char *method_signature);`

Mustang Beta (-XX:+ExtendedDTraceProbes)

- Certain probes are expensive and hence turned off by default
- These probes are
 - > object-alloc
 - > method-entry, method-return
 - > monitor probes (monitor-wait, monitor-contended-enter etc.)
- Requires you to start your application with the flag
-XX:+ExtendedDTraceProbes
- This requirement may be removed in future! (work-in-progress)

DTrace in Mustang

- In Mustang will support DTrace “out of the box”
- It will provide the probes dvm provided and will include the following
 - > Method compilation (method-compile-begin/end)
 - > Compiled method load/unload(compiled-method-load/unload)
 - > JNI method probes.
 - > DTrace probes as entry and return from each JNI method.
- java.net version of Mustang has the code for these probes.

hotspot provider (Mustang)

- VM lifecycle probes
- Thread lifecycle probes
- Classloading probes
- Garbage collection probes
- Method compilation probes
- Monitor probes
- Application probes (object alloc, method entry/return)

VM Lifecycle Probes

```
hotspot$1:::vm-init-begin {  
    /* actions */  
}  
hotspot$1:::vm-init-end {  
    /* actions */  
}  
hotspot$1:::vm-shutdown {  
    /* actions */  
}
```

Thread Lifecycle Probes

```
hotspot$1:::thread-start {  
    self->ptr = (char*) copyin(arg0, arg1+1);  
    self->ptr[arg1] = '\0';  
    self->threadname = (string) self->ptr;  
    printf("Thread %s started\n", self->threadname);  
}  
hotspot$1:::thread-stop {  
    /* actions */  
}
```

Classloading Probes

```
hotspot$1:::class-loaded {  
    self->str_ptr = (char*) copyin(arg0, arg1+1);  
    self->str_ptr[arg1] = '\0';  
    self->name = (string) self->str_ptr;  
    printf("class %s loaded\n", self->name);  
}  
hotspot$1:::class-unloaded {  
    /* actions */  
}
```

Garbage Collection Probes

```
hotspot$1:::gc-begin {  
    self->gc_ts = timestamp;  
}  
hotspot$1:::gc-end {  
    self->time = (timestamp - self->gc_ts) / 1000;  
    printf("GC in %10d second(s)\n", self->time);  
    self->gc_ts = 0;  
}
```

Method Compilation Probes

```
hotspot$1::method-compile-begin {  
    self->str = (char*) copyin(arg2, arg3+1);  
    self->str[arg3] = '\0';  
    self->classname = (string)self->str;  
    self->str = (char*) copyin(arg4, arg5+1);  
    self->str[arg5] = '\0';  
    self->methodname = (string)self->str;  
    printf("Compile begin %s.%s\n",  
        self->classname, self->methodname);  
}
```

Monitor Probes

```
hotspot$1:::monitor-contended-enter {  
    /* actions */  
}
```

```
hotspot$1:::monitor-contended-entered {  
    /* actions */  
}
```

```
hotspot$1:::monitor-wait {  
    /* actions */  
}
```

Object Allocation Probes

```
hotspot$1::object-alloc {  
    self->str_ptr = (char*) copyin(arg1, arg2+1);  
    self->str_ptr[arg2] = '\0';  
    self->classname = (string) self->str_ptr;  
    @allocs_count[self->classname] = count();  
    @allocs_size[self->classname] = sum(arg3);  
}
```

Method entry/return Probes

```
hotspot$1::method-entry {  
    self->ptr = (char*)copyin(arg1, arg2+1);  
    self->ptr[arg2] = '\0';  
    self->classname = (string)self->ptr;  
    self->ptr = (char*)copyin(arg3, arg4+1);  
    self->ptr[arg4] = '\0';  
    self->methodname = (string)self->ptr;  
    printf("Entering into %s.%s\n", classname, methodname);  
}
```

Method Frequency

```
hotspot$1:::method-entry {  
    self->ptr = (char*)copyin(arg1, arg2+1);  
    self->ptr[arg2] = '\0';  
    self->classname = (string)self->ptr;  
    self->ptr = (char*)copyin(arg3, arg4+1);  
    self->ptr[arg4] = '\0';  
    self->methodname = (string)self->ptr;  
    @[self->classname, self->methodname] = count();  
}  
END {  
    printa("%-10@u %s.%s()\n", @);  
}
```

Exception (mixed mode) Stack Trace!

```

hotspot$1:::method-entry {
    self->ptr = (char*)copyin(arg1, arg2+1);
    self->ptr[arg2] = '\0';
    self->classname = (string)self->ptr;
    self->ptr = (char*)copyin(arg3, arg4+1);
    self->ptr[arg4] = '\0';
    self->methodname = (string)self->ptr;
}

```

```

hotspot$1:::method-entry

```

```

/ self->classname == "java/lang/Throwable" && self->methodname == "<init>" / {
    jstack();
}

```

Observability Tools on “events”

```
int cnt;
BEGIN {
    cnt = 0; /* initialize count */
}
hotspot1274::gc_begin {
    cnt++;
}
hotspot1274::gc-begin
/ cnt == 100 / {
    /* create java heap dump */
    system("jmap -dump:format=b,file=heap.bin 1274");
}
```

hotspot_jni provider (Mustang)

- Probes for Java Native Interface (JNI)
- Located at entry/return points of all JNI functions
- Probe arguments are same as corresponding JNI function arguments (for `_entry` probes)
- For `XXX_return` probes, probe argument is return value
- Examples:
 - `hotspot_jni$1::GetPrimitiveArrayCritical_entry`
 - `hotspot_jni$1::GetPrimitiveArrayCritical_return`

JNI Calls Stat

```
hotspot$1::*_entry {  
    JNI_CALLS++;  
    @jni_calls[probename] = count();  
}  
:::END {  
    printf("Total JNI calls %d\n", JNI_CALLS);  
    printa("%10d %s\n", @jni_calls);  
}
```

- **probename** is a built-in variable

dvm provider – GC times

- To see time taken by GC

```
#!/usr/sbin/dtrace -s
dvm$target:::gc-start
{
    self->ts = vtimestamp;
}
dvm$target:::gc-finish
{
    printf("GC ran for %d nsec\n", vtimestamp - self->ts);
}
# ./java_gc.d -p `pgrep -n java`
```

java_gc.d

dvm provider - alloc/free

- To see objects that are allocated and freed

```
#!/usr/sbin/dtrace -qs
dvm$target:::object-alloc
{
    printf("%s allocated %d size object\n",copyinstr(arg0), arg1);
}

dvm$target:::object-free
{
    printf("%s freed %d size object\n",copyinstr(arg0), arg1);
}

# ./java_alloc.d -p `pgrep -n java`
```

java_alloc.d

dvm provider - methods

- To get count of java methods called

```
#!/usr/sbin/dtrace -s
```

java_method_count.d

```
dvm$target:::method-entry
```

```
{  
    @[copyinstr(arg0),copyinstr(arg1)] = count();  
}
```

```
# ./java_method_count.d -p `pgrep -n java`
```

dvm provider - time-spent

- To get time-spent on java methods

```

#!/usr/sbin/dtrace -s java_method.d
dvm$target:::method-entry
{
    self->ts[copyinstr(arg0),copyinstr(arg1)] = vtimestamp;
}

dvm$target:::method-return
{
    @ts[copyinstr(arg0),copyinstr(arg1)] = sum(vtimestamp -
        self->ts[copyinstr(arg0),copyinstr(arg1)]);
}

# ./java_method.d -p `pgrep -n java`

```

DTrace resources

Granting privilege to run DTrace

- A system admin can grant any user privileges to run DTrace using the Solaris Least Privilege facility privileges(5).
- DTrace provides for three types of privileges.
 - dtrace_proc - provides access to process level tracing no kernel level tracing allowed. (pid provider is about all they can run)
 - dtrace_user – provides access to process level and kernel level probes but only for process to which the user has access. (ie) they can use syscall provider but only for syscalls made by process that they have access.
 - dtrace_kernel – provides all access except process access to user level procs that they do not have access.
- Enable these priv by editing `/etc/user_attr`.
 - > format ***user-name*::::defaultpriv=basic,privileges**

DTrace Resources

- Here are a few of the many DTrace resources available for you
 - > “Solaris Dynamic Tracing Guide” is an excellent resource. Many of the examples in this presentation are from the Guide.
<http://docs.sun.com/db/doc/817-6223>
 - > The BigAdmin DTrace web page has a lot of good info
<http://www.sun.com/bigadmin/content/dtrace/>
 - > Open Solaris DTrace community page
<http://www.opensolaris.org/os/community/dtrace/>
 - > DTrace toolkit contains a lot of very useful scripts
<http://www.opensolaris.org/os/community/dtrace/dtrace>

More DTrace Resources

- The DTrace HowTo guide
<http://www.sun.com/software/solaris/howtoguides/dtracehowto.jsp>
- Read the Blogs from Bryan Cantrill, Adam Leventhal, Mike Shapiro
<http://blogs.sun.com/roller/page/bmc>
<http://blogs.sun.com/roller/page/ahl>
<http://blogs.sun.com/mws>
- A. Sundararajan, Dtrace and Java Blog
<http://blogs.sun.com/sundararajan>
- Of course you can google DTrace.
<http://www.google.com/search?q=dtrace>

Thank you!

Peter Karlsson
Technology Evangelist
Sun Microsystems
<http://opensolaris.org>

Reference Slides

Predicate example usage

- Predicates can be used as flow control constructs.

```
#!/usr/sbin/dtrace -qs
dtrace:::BEGIN{
msecond = 0;
speed = 0;
}
profile:::tick-1msec
/speed < 60/
{
    speed = 32*msecond/1000/5280*3600; /* converting 9.8 m/s/s = 32 ft/s/s. 5280 feet in a mile*/
    msecond ++;
}

profile:::tick-1msec
/speed >= 60/
{
    printf("0 to %d in %d milli seconds\n", speed, msecond);
    exit(0);
}
```

0-60.d

Provider details

Here is some detailed info on providers

dtrace Provider

- The dtrace provider provides three probes (BEGIN, END, ERROR)
 - > BEGIN
 - > BEGIN is the first probe to fire.
 - > All BEGIN clauses will fire before any other probe fires.
 - > Typically used to initialize.
 - > END
 - > Will fire after all other probes are completed
 - > Can be used to output results
 - > ERROR
 - > Will fire under an error condition
 - > For error handling

dtrace provider. Example

dtrace.d

```
#!/usr/sbin/dtrace -s
BEGIN
{
    i = 0;
    exit(0);
}

ERROR
{
    printf("Error has occurred!");
}

END
{
    printf("Exiting and dereferencing a null pointer\n");
    *(char *)i;
}
```

lockstat Provider

- lockstat has two kinds of probes. *contention-event* probes and *hold-event* probes.
 - > *contention-event* – Used to track contention events. As these are rare it does not impose too much of an overhead and so can be safely enabled
 - > *hold-event* – These are to track acquiring and releasing locks. Enabling these probes can incur an overhead as these events are more common.
- lockstat allows you to probe *adaptive*, *spin*, *thread and reader and writer* locks.

lockstat probes

	Contention-event	Hold-event
Adaptive	adaptive-block, adaptive-spin	adaptive-acquire, adaptive block
Spin	spin-spin	spin-acquire, spin-block
Thread	thread-spin	
Reader Writer	rw-block	rw-acquire, rw-upgrade, rw-downgrade, rw-release

lockstat - Example

Here is an example. It counts all the lock events of the given executable.

```
#!/usr/sbin/dtrace -qs  
lockstat::  
/execname==$$/  
{  
    @locks[probename]=count();  
}
```

lockstat.d

plockstat provider

- One final provider that may be of interest is plockstat
- plockstat is the user land equivalent of lockstat in kernel.
- Three types of lock events can be traced.
 - Contention events – probes for user level lock contention
 - Hold events – probes for lock acquiring, releasing etc.
 - Error events – error conditions.
- There are two families of probes
 - Mutex Probes
 - Reader Writer lock Probes

plockstat Providers

	Contention Probe	Hold Probe	Error Probe
Mutex Probes	mutex-block mutex-spin	mutex-acquire mutex-release	mutex-error
Reader/Writer lock probes	rw-block	rw-acquire rw-release	rw-error

sched Provider

- The sched provider allows users to gain insight into how a process is scheduled. It helps answer questions like why and when did the thread of interest change priority.
- The following are a few probes that are part of the sched provider.
 - change-pri** – When priority changes
 - dqueue/enqueue** – when proc taken off or put on the run queue
 - off-cpu / on-cpu** – when thread taken off or put on a cpu.
 - preempt** – when thread preempted
 - sleep / wakeup** – when thread sleep on a synchronization object and when it wakes up.

sched examples.

This script prints the distribution of the time threads spends on a cpu.

```
#!/usr/sbin/dtrace -qs
sched:::on-cpu
{
    self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = quantize(timestamp - self->ts);
}
```

sched.d

Arrays

name[key] = expression;

- name – name of array
- key – list of scalar expression values (tuples)
- expression – evaluates to the type of array

```
#!/usr/sbin/dtrace -s
syscall::open*:entry,
syscall::close*:entry
{
    ts[probfunc,pid,tid]=timestamp; /* save time stamp at entry */
}

syscall::open*:return,
syscall::close*:return
{
    timespent = timestamp - ts[probfunc,pid,tid];
    printf("%s threadID %d spent %d nsecs in %s\n", execname, tid, timespent, probfunc);
    /* print time-spent at return */
    ts[probfunc,pid,tid]=0;
    timespent = 0;
}
```

array.d

struct construct

```
struct type{  
    element1;  
    element2;  
}
```

Example

```
struct info{  
    string f_name;  
    int count;  
    int timespent;  
} /* definition of struct info */
```

```
struct info my_callinfo; /* Declaring my_callinfo as variable of type info */
```

```
my_callinfo.f_name; /* access to member of struct */
```

Porting Solaris to PPC – DTrace style

port.d

```
#!/usr/sbin/dtrace -Cs
#include<sys/systeminfo.h>

#pragma D option destructive

syscall::uname:entry
{
    self->addr = arg0;
}

syscall::uname:return
{
    copyoutstr("SunOS", self->addr, 257);
    copyoutstr("PowerPC", self->addr+257, 257);
    copyoutstr("5.10", self->addr+(257*2), 257);
    copyoutstr("Generic", self->addr+(257*3), 257);
    copyoutstr("PPC", self->addr+(257*4), 257);
}
```

Porting Solaris to PPC – DTrace style

```
syscall::systeminfo:entry  
/arg0==SI_ARCHITECTURE/  
{  
    self->arch = arg1;  
}
```

```
syscall::systeminfo:return  
/self->arch/  
{  
    copyoutstr("PowerPC", self->arch,257);  
    self->arch=0;  
}
```

```
syscall::systeminfo:entry  
/arg0==SI_PLATFORM/  
{  
    self->mach = arg1;  
}
```

```
syscall::systeminfo:return  
/self->mach/  
{  
    copyoutstr("APPL,PowerBook-G4", self->mach,257);  
    self->mach=0;  
}
```

Containers – DTrace style

contain.d

```
#!/usr/sbin/dtrace -Cs
#include<sys/systeminfo.h>

#pragma D option destructive

syscall::uname:entry
/ppid == $1/
{
    self->addr = arg0;
}

syscall::uname:return
/ppid == $1/
{
    copyoutstr("SunOS", self->addr, 257);
    copyoutstr("PowerPC", self->addr+257, 257);
    copyoutstr("5.10", self->addr+(257*2), 257);
    copyoutstr("Generic", self->addr+(257*3), 257);
    copyoutstr("PPC", self->addr+(257*4), 257);
}
```

Containers – DTrace style

```
syscall::systeminfo:entry
/arg0==SI_ARCHITECTURE/
{
    self->arch = arg1;
}
```

```
syscall::systeminfo:return
/self->arch && ppid == $1/
{
    copyoutstr("PowerPC", self->arch,257);
    self->arch=0;
}
```

```
syscall::systeminfo:entry
/arg0==SI_PLATFORM && ppid == $1/
{
    self->mach = arg1;
}
```

```
syscall::systeminfo:return
/self->mach && ppid == $1/
{
    copyoutstr("APPL,PowerBook-G4", self->mach,257);
    self->mach=0;
}
```

Stop that killer.

```
#!/usr/sbin/dtrace -wqs
```

sig2.d

```
proc:::signal-send  
/args[1]->pr_fname == $$1/  
{  
    printf("%s(pid:%d) is sending signal %d to %s\n", execname, pid, args[2],args[1]->pr_fname);  
    stop();  
}
```

```
$ ./sig2.d bc  
bash(pid:3987) is sending signal 9 to bc
```

This one is modified to include destructive actions (-w)
We do the stop() which stops the sending process.
You can then examine the sending process.
Note: We are not stopping the signal delivery itself.

Postmortem Tracing

Postmortem tracing

- A nifty feature of DTrace is to be able to dig DTrace related info from a system crash dump.
- Feature could be very useful to support engineers
- Here is how it works.
 - > Load core dump into mdb
 - > `::dtrace_state` – prints out details about all dtrace consumers when the dump was generated.
 - > Take the address for dtrace consumer and
 - > `<addr>::dtrace` – prints all the info from dtrace buffer.

Postmortem tracing

- You can create a ring buffer of data using dtrace
 - > Use the -b option for data size & -bufpolicy=ring for ring buffer policy.
- You can leave this running and if system crashes you can analyze the buffer from the crash dump.
- Options
 - > `<addr>::dtrace -c 1`
 - > Print only info from cpu 1.

Speculation

- We will now see how to catch a bad code path using speculation!
- Here is why we need speculation.
 - > Some time we only see error message after the error has occurred.
For example: We see a function return an error but the problem was caused by something that the function did earlier.
We see the error and want to go back and find out what the function did wrong. But alas the function has already happened
 - > One solution could be to save details every time the function executes but this is wasting trace buffer with a lot of useless data when were are only concerned about the one time the function failed.
 - > A better solution - **speculation**

Speculation - example

```
pid$target::fopen64:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("Path is %s\n", copyinstr(arg0));
}
```

```
pid$target:::entry
/self->spec/
{
    speculate(self->spec);
}
```

```
pid$target:::return
/self->spec/
{
    speculate(self->spec);
}
```

```
pid$target::fopen64:return
/self->spec && arg1 != 0/
{
    discard(self->spec);
    self->spec = 0;
}
```

```
pid$target::fopen64:return
/self->spec && arg1 == 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

spec.d

Increasing max probes

- You can see how easy it is to create a lot of probes on the fly using the pid provider. Just one note before we move on.
- By default the maximum probes that Solaris allows is 250,000. You can increase this by editing `/kernel/drv/fasttrap.conf`
 - > increase `fasttrap-max-probes` variable
 - > run `update_drv fasttrap` or reboot system

Consumers

- A DTrace consumer is a process that interacts with DTrace
- There is no limit on concurrent consumers
- `dtrace(1M)` is a DTrace consumer that acts as a generic front-end to the DTrace facility
- Examples:
sysinfo, lockstat and plockstat.

Make your Own Provider

Defining Provider & Probes

- Creating your own probe is very very simple.
- Step 1: Figure out what probes you want to add to your app. This is probably the hardest part. Remember
 - > probes will be used by users who do not understand your implementation details
 - > probes can be listed so you need to think of how long you want to support these probes.
 - > probe and provider names should be intuitive
 - > location of probe should be intuitive
- You are more than 80% there.

Defining Provider & Probes

- Step 2: Provider & probe definition
 - > create a .d file with the following entries

```
provider foobar {  
    probe foo(int,int);  
    probe bar();  
};
```

You just created a provider foobar and two probes foo & bar. That's it. (almost!)

The arguments are the types of the two arguments your probe exposes.

Defining Provider & Probes

- Step 3: Insert probes into your code.

> in your source code at the location you want the probes to fire.

```
foo {
    if(inp<10){
        val1 = inp^3;
    }else
        val1 = inp^2;
    }
    ...
}
```

Add the probe macro

```
#include <sys/sdt.h>
foo {
    if(inp<10){
        DTRACE_PROBE2(foobar, foo, inp, 3);
        val1=inp^3;
    }else
        ...
}
```

Defining Provider & Probes

- Step 4: Build your code.

- > Compile

- ```
cc -c probeable.c
dtrace -G -32 -s foobar.d probeable.o
cc -o probeable foobar.o probeable.o
```

- The dtrace command compiles the .d file. It takes input from the probeable.o(place in your code where you have added the code)
- -G option generates a .o file
- -32 / -64 for 32 and 64 bit apps.
- The last line compiles all the .o's into your app.
- Ok you are done! Really!

# Using the probes

- Now that you have created your own probes you can use them like any other probe.
- Some things to remember
  - > Access your probe using <provider\_name><pid> format.
  - > So if your process id is 3346 then provider name is foobar3346