Dtrace
Using SDT probes

Alan Hargreaves

Product Technical Support Kernel Engineer

Sun Microsystems

Overview

- What is a SDT probe?
- Example – kernel io provider
- Slight Digression – kernel mib provider
- Example – User Space

What is an SDT Probe?

- Statically Defined Tracing Probe
- All probes (except fbt and pid) are SDT probes
- Data gathering at specific points in the code
- Implemented by DTRACE_PROBE* macros in <sys/sdt.h>
- When the probe is not enabled, there are a sequence of “nops” in the binary, making for minimal impact.

Kernel SDTs

- Relatively self explanatory

```
#define DTRACE_PROBE1(name, type1, arg1) {            \
        extern void __dtrace_probe_##name(uintptr_t); \
        __dtrace_probe_##name((uintptr_t)(arg1));     \
}
```

- name – name of the probe
- type1 – type of first value in the probe
- arg1 – the actual value the probe will report

Kernel Example – io provider

- All kinds of nifty stuff when dealing with I/O

```
$ dtrace -l -P io
   ID   PROVIDER            MODULE                     FUNCTION NAME
 521         io          genunix                  biodone done
 522         io          genunix                  biowait wait-done
 523         io          genunix                  biowait wait-start
 532         io          genunix           default_physio start
 533         io          genunix            bdev_strategy start
 534         io          genunix                  aphysio start
1263         io              nfs                 nfs4_bio done
1264         io              nfs                 nfs3_bio done
1265         io              nfs                  nfs_bio done
1266         io              nfs                 nfs4_bio start
1267         io              nfs                 nfs3_bio start
1268         io              nfs                  nfs_bio start
```

io provider - implemetation

- Wrappers around DTRACE_PROBE* macros in /usr/include/sys/sdt.h

```
#define DTRACE_IO(name)                                                 \
        DTRACE_PROBE(__io_##name);

#define DTRACE_IO1(name, type1, arg1)                                   \
        DTRACE_PROBE1(__io_##name, type1, arg1);

#define DTRACE_IO2(name, type1, arg1, type2, arg2)                      \
        DTRACE_PROBE2(__io_##name, type1, arg1, type2, arg2);

#define DTRACE_IO3(name, type1, arg1, type2, arg2, type3, arg3) \
        DTRACE_PROBE3(__io_##name, type1, arg1, type2, arg2,    \
            type3, arg3);

#define DTRACE_IO4(name, type1, arg1, type2, arg2,              \
    type3, arg3, type4, arg4)                                   \
        DTRACE_PROBE4(__io_##name, type1, arg1, type2, arg2,    \
            type3, arg3, type4, arg4);
```

- Generally placed where the I/O kstats are updated

io provider - example

```
$ dtrace -q -n '
tick-10s { exit(0); }
io:::wait-start /execname == "soffice.bin"/ {
        self->start = timestamp;
}
io:::wait-done /self->start/ {
        @[execname] = quantize(timestamp - self->start);
        self->start = 0;
}'

      soffice.bin
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@                                   33
            8192 |@@@@@@@@@@@@@@@@@@@@                     106
           16384 |                                         0
           32768 |                                         1
           65536 |                                         2
          131072 |@@@@@@                                   31
          262144 |@@@@                                     22
          524288 |@                                        4
         1048576 |@@                                       8
         2097152 |                                         1
         4194304 |                                         1
         8388608 |                                         0
```

A Slight Digression

- Probes listed as SDT probes are a special case of generic probes
- In kernel space they simply don't have a provider
- The Dtrace team is working on making it possible for third parties to create probes under their own providers
- The MIB provider
  - > Example of how a small change can create a host of useful probes

mib provider - kstats

- Two macros in /usr/include/inet/mib2.h

```
#define BUMP_MIB(s, x)          {                                \
        extern void __dtrace_probe___mib_##x(int, void *);       \
        void *stataddr = &((s)->x);                              \
        __dtrace_probe___mib_##x(1, stataddr);                   \
        (s)->x++;                                                \
}

#define UPDATE_MIB(s, x, y)     {                                \
        extern void __dtrace_probe___mib_##x(int, void *);       \
        void *stataddr = &((s)->x);                              \
        __dtrace_probe___mib_##x(y, stataddr);                   \
        (s)->x += (y);                                           \
}
```

- Probe point for every time a kstat is updated with one of these two macros – 436 of them!

mib provider - example

- Give me the stack on the next time udpOutDatagrams is updated

```
$ dtrace -q -n '
mib:::udpOutDatagrams {
        stack(20);
        exit(0);
}'

              unix`putnext+0x1b7
              genunix`strput+0x168
              genunix`kstrputmsg+0x1df
              sockfs`sosend_dgram+0x1ca
              sockfs`sotpi_sendmsg+0x3f1
              sockfs`sendit+0x116
              sockfs`send+0x6b
              unix`sys_call+0x104
```

User Space

- We use a slightly different macro in user space

```
#define DTRACE_PROBE1(provider, name, arg1)
{                           \
        extern void __dtrace_##provider##___##name(unsigned long);     \
        __dtrace_##provider##___##name((unsigned long)arg1);           \
}
```

- provider - name of the provider (duh!)
- name - name of the probe
- arg1, ... - the actual value the probe will report
- Note that we don't define the type. This is done differently in user space

helloworld1.c

- Let's take a simple little program

```
#include <stdio.h>
#include <unistd.h>

int
main(int ac, char **av) {
        int i;
        for (i = 0 ; i < 5; i++) {
                printf("Hello World\n");
                sleep(2);
        }
}
$ /usr/ccs/bin/make helloworld1
cc    -c  helloworld1.c
cc -o helloworld1 helloworld1.o
$ ./helloworld1
Hello World
Hello World
Hello World
Hello World
Hello World
```

- Pretty much what you'd expect

Adding a probe – helloworld2.c

- Say we wanted to monitor the loop variable

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sdt.h>

int
main(int ac, char **av) {
        int i;
        for (i = 0 ; i < 10; i++) {
                DTRACE_PROBE1(world, loop, i);
                printf("Hello World\n");
                sleep(2);
        }
}
```

- We need to include <sys/sdt.h> and add the probe
- But wait, in user space there's more

Adding a probe – myserv.d

- We still need to define the types of the arguments and the stability levels.
- This gets linked into the code later

```
provider world {
        probe loop(int);
};

#pragma D attributes Evolving/Evolving/Common provider world provider
#pragma D attributes Private/Private/Common provider world module
#pragma D attributes Private/Private/Common provider world function
#pragma D attributes Evolving/Evolving/Common provider world name
#pragma D attributes Evolving/Evolving/Common provider world args
```

- The stuff in provider is relatively self explanatory
- See chapter 39 of the manual for the stability stuff

Putting it all together

- In order to build the probes incorporating the provider description we need another step in the build

```
$ make helloworld2
cc   -c  helloworld2.c
dtrace -32 -G -s myserv.d helloworld2.o
cc -o helloworld2 -ldtrace myserv.o helloworld2.o
```

- The *-G* option creates the myserv.o
- Running without dtrace gives us the same result

```
$ ./helloworld2
Hello World
Hello World
Hello World
Hello World
Hello World
```

Tracing the new binary

- Let's look at both the counter and the first argument to printf

```
$ dtrace -q -c ./helloworld2 -n '
world$target:::loop {
       printf("%s:%s loop = %d\n", probemod, probefunc,arg0);
}
pid$target::printf:entry { printf("%s:%s\n", probefunc, copyinstr
(arg0));'
Hello World
helloworld2:main loop = 0
printf:Hello World

Hello World
helloworld2:main loop = 1
printf:Hello World
...
Hello World
helloworld2:main loop = 4
printf:Hello World
```

- *i* is now observable, but with no overhead unless we are tracing it.

Conclusion

- SDT probes are an easy way to make stuff visible
- The helloworld example was trivial, but, ...
- Imagine being able to place probes like this into large applications or drivers
- We get observability without the need for
  - Seperate instrumented binaries
  - Restart
  - Reboot (in the case of drivers/kernel)
- With next to no overhead if they are not being observed

Questions/Comments?

Dtrace
Using SDT Probes

Alan Hargreaves

alan.hargreaves@sun.com