The DTrace backend on Solaris for x86/x64

Frank Hofmann

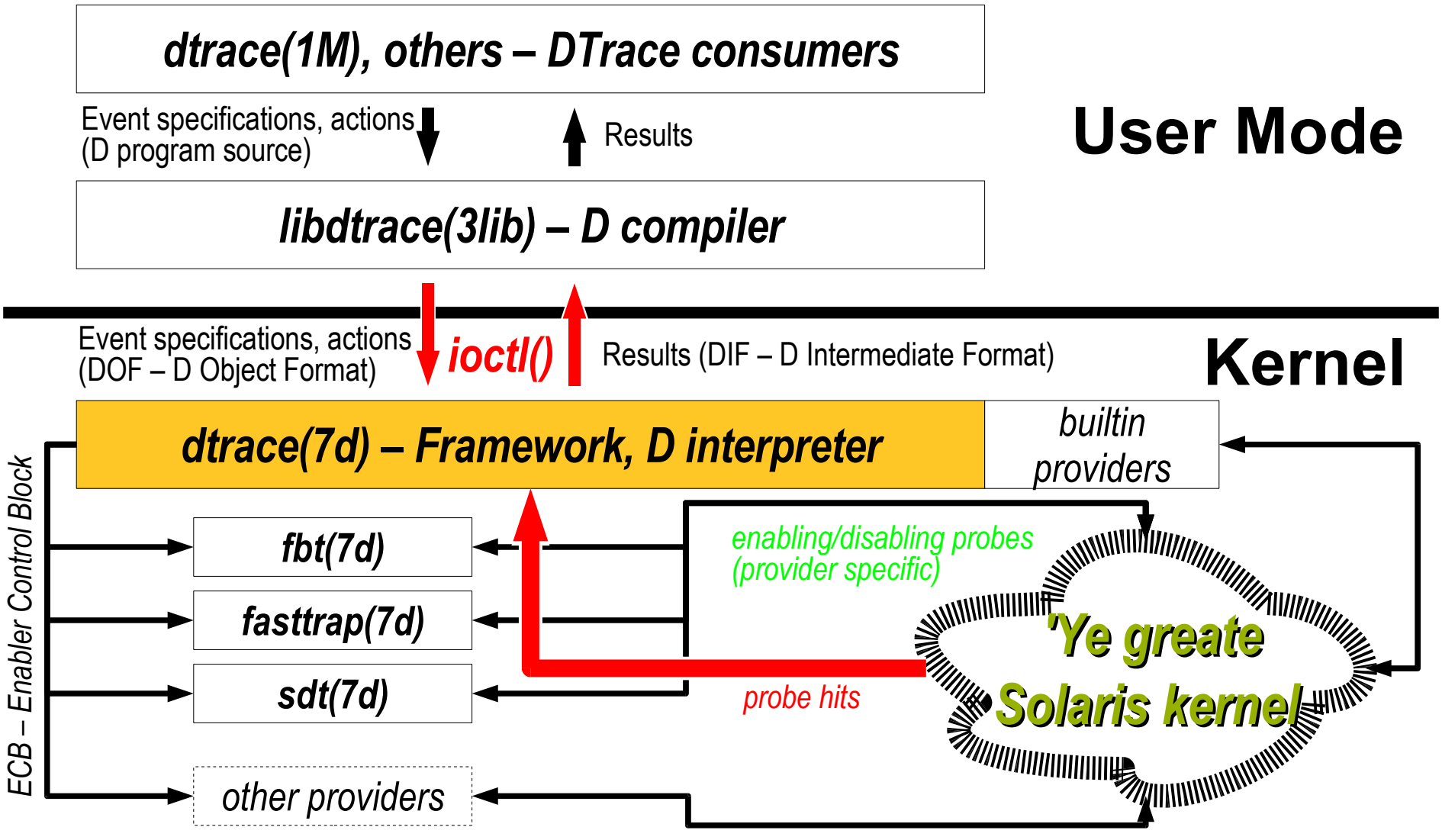
OP/N1 Released Products Engineering

Sun Microsystems UK

Overview

- General overview on DTrace capabilities
- DTrace providers
- Architecture-specifics: Dynamic Instrumentation
- Implementation of DTrace providers:
  - fbt(7d) – kernel function boundary tracing
  - fasttrap(7d) – the PID provider
  - sdt(7d) – statically-defined tracing
-

DTrace architecture overview



Architecture Dependence in DTrace

- Most parts of DTrace (including many providers) are fully generic
- Architecture Dependencies found in:
  - Safe access to kernel memory (stray pointer detection)
  - Areas that differ via ABI:
    - Function argument retrieval
    - Stacktracing
  - ***Providers with "tracepoint"-style probes***
- The devil is in the detail ...

DTrace Sourcecode structure

Check OpenSolaris source tree:
http://cvs.opensolaris.org/source/xref/on/usr/src/

- Generic sourcecode organization:
  - “top half” - generic interfaces, common to all architectures
  - “bottom half” - platform-specific backend.

DTrace Sourcecode structure

Program	*generic source*	*platform-specific*
dtrace(1m)	cmd/dtrace/	cmd/dtrace/amd64/ cmd/dtrace/i386/ cmd/dtrace/sparc/
libdtrace(3lib)	lib/libdtrace/common/	lib/libdtrace/amd64/ lib/libdtrace/i386/ lib/libdtrace/sparc/
dtrace(7d)	uts/common/dtrace	uts/intel/dtrace/ uts/sparc/dtrace/

Tracepoints – heart of dynamic tracing

- Allow to instrument ***everything*** ...
- ***Zero overhead*** if tracing is inactive

What DTrace does:

- Actively manipulate binary code (program text)
- Insert instructions that cause traps
- Interpose on the trap handler(s).

Traced vs. non-traced: PID provider

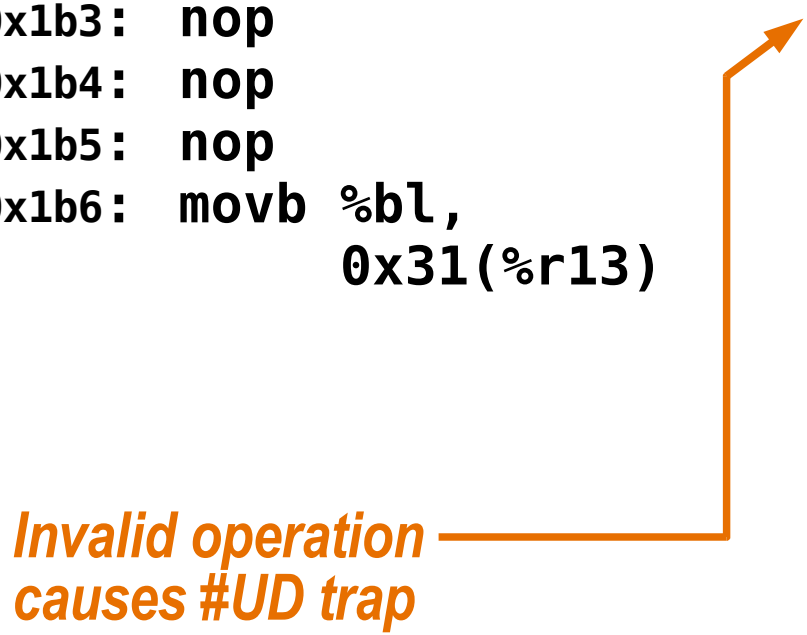
machine code, tracing inactive:		*traced*:
`main:`	`pushl %ebp`	`int $0x3`
`main+1:`	`movl %esp,%ebp`	`int $0x3` *{junk}*
`main+3:`	`andl $0xfffffff0,%esp`	`int $0x3` *{junk}*
`main+6:`	`pushl %ebx`	`int $0x3`
`main+7:`	`pushl %esi`	`int $0x3`
`main+8:`	`pushl %edi`	`int $0x3`
`main+9:`	`pushl $0x8050a4c`	`int $0x3` *{junk}*
`main+0xe:`	`pushl $0x6`	`int $0x3` *{junk}*
`main+0x10:`	`call -0x19f`	`int $0x3` *{junk}*
	`libc.so.1`setlocale`	
`main+0x15:`	`addl $0x8,%esp`	`int $0x3` *{junk}*
`main+0x18:`	`pushl $0x8050a3c`	`int $0x3` *{junk}*
`main+0x1d:`	`call -0x19c`	`int $0x3` *{junk}*
	`libc.so.1`textdomain`	
`main+0x22:`	`...`	`...`

Traced vs. non-traced: FBT provider

machine code,	*tracing inactive:*	*traced*:
`ufs_mount:`	`pushq %rbp`	`int $0x3`
`ufs_mount+1:`	`movq %rsp,%rbp`	`movq %rsp,%rbp`
`ufs_mount+4:`	`subq $0x88,%rsp`	`subq $0x88,%rsp`
`ufs_mount+0xb:`	`pushq %rbx`	`pushq %rbx`
[...]	*[...]*	*[...]*
`ufs_mount+0x3f3:`	`popq %rbx`	`popq %rbx`
`ufs_mount+0x3f4:`	`movq %rbp,%rsp`	`movq %rbp,%rsp`
`ufs_mount+0x3f7:`	`popq %rbp`	`popq %rbp`
`ufs_mount+0x3f8:`	`ret`	`int $0x3`

Traced vs. non-traced: SDT provider

machine code,	*tracing inactive:*	*traced*:
[...]	*[...]*	*[...]*
`squeue_enter_chain+0x1af:`	`xorl %eax,%eax`	`xorl %eax,%eax`
`squeue_enter_chain+0x1b1:`	`nop`	`nop`
`squeue_enter_chain+0x1b2:`	`nop`	`nop`
`squeue_enter_chain+0x1b3:`	`nop`	`lock nop`
`squeue_enter_chain+0x1b4:`	`nop`	
`squeue_enter_chain+0x1b5:`	`nop`	`nop`
`squeue_enter_chain+0x1b6:`	`movb %bl, 0x31(%r13)`	`movb %bl, 0x31(%r13)`



How SDT works

- Sourcecode:

Name of statically-defined probe

```
DTRACE_PROBE4(squeue__enqueuechain, squeue_t *, sqp,    \
        mblk_t *, mp, mblk_t *, tail, int, cnt);        \
```

- ELF object:

```
Symbol Table Section:  .symtab
index  value      size           type bind oth ver shndx  name
[2561] 0x000be0d1 0x000000000965 FUNC GLOB D   0   .text  squeue_enter_chain
Relocation Section:
.rela.eh_frame type  offset  addend             section    with respect to
R_AMD64_PC32         0xbe283 0xfffffffffffffffc .rela.text __dtrace_probe_squeue__enqueuechain
```

squeue_enter_chain+0x1b2

Relocation hook

- code in object file:

```
squeue_enter_chain+0x1b1:     e8 00 00 00 00    call   <...>
```

How SDT works (continued)

- Executable file doesn't match running binary:

Code offset	executable file contents	running code
[...]	*[...]*	*[...]*
squeue_enter_chain+0x1b1:	call <_dtrace_probe_...>	nop
squeue_enter_chain+0x1b2:	...	nop
squeue_enter_chain+0x1b3:	...	nop
squeue_enter_chain+0x1b4:	...	nop
squeue_enter_chain+0x1b5:	...	nop

- How does this work ?
- ***Answer***: SDT gets help by the runtime linker !

Zero overhead !

How SDT works (continued)

- Kernel runtime linker, krtld:
  usr/src/uts/intel/amd64/krtld/kobj_reloc.c

```
#define SDT_NOP         0x90
#define SDT_NOPS        5

static int
sdt_reloc_resolve(struct module *mp, char *symname, uint8_t *instr)
{
[ ... ]
        /*
         * The "statically defined tracing" (SDT) provider for DTrace uses
         * a mechanism similar to TNF, but somewhat simpler.  (Surprise,
         * surprise.)  The SDT mechanism works by replacing calls to the
         * undefined routine __dtrace_probe_[name] with nop instructions.
         * The relocations are logged, and SDT itself will later patch the
         * running binary appropriately.
         */
[ ... ]
        for (i = 0; i < SDT_NOPS; i++)
                instr[i - 1] = SDT_NOP;
[ ... ]
```

Tracepoint insertion – DTracing DTrace

- Quick idea:

```
# dtrace -n "fbt::fasttrap_tracepoint_install:entry { stack();ustack();exit(0) }"
```

- Doesn't work – no tracepoints in providers. Use a trick:

```
# mdb -k
> fasttrap_tracepoint_install::dis ! grep call
fasttrap_tracepoint_install+0x28:       call   +0x7aaab2c       <uwrite>
> fasttrap_tracepoint_install+0x28+5=J
                ffffffff401f919
# dtrace -n \
'fbt::uwrite:entry /caller == 0xffffffff401f919/ { stack();ustack();exit(0) }'
```

- In another window, run:

```
# dtrace -n "pid101394::main: {}"
```

Tracepoint insertion – DTracing DTrace

```
dtrace: description 'fbt::uwrite:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  12557                     uwrite:entry
              fasttrap`fasttrap_tracepoint_install+0x2d
              fasttrap`fasttrap_tracepoint_enable+0x272
              fasttrap`fasttrap_pid_enable+0x11b
              dtrace`dtrace_ecb_enable+0xbb
              dtrace`dtrace_ecb_create_enable+0x63
              dtrace`dtrace_match+0x1d6
              dtrace`dtrace_probe_enable+0x8a
              dtrace`dtrace_enabling_match+0x84
              dtrace`dtrace_ioctl+0xdeb
              genunix`cdev_ioctl+0x55
              specfs`spec_ioctl+0x99
              genunix`fop_ioctl+0x2d
              genunix`ioctl+0x180
              unix`sys_syscall+0x275

              libc.so.1`ioctl+0xa
              libdtrace.so.1`dtrace_program_exec+0x51
              dtrace`exec_prog+0x37
              dtrace`main+0xc02
              dtrace`0x4026cc
```

Tracepoint insertion, FBT provider

- Tracepoint enabling/disabling: simple memory write

```
static void
fbt_enable(void *arg, dtrace_id_t id, void *parg)
{
        fbt_probe_t *fbt = parg;
        struct modctl *ctl = fbt->fbtp_ctl;
[ ... ]
        for (; fbt != NULL; fbt = fbt->fbtp_next)
                *fbt->fbtp_patchpoint = fbt->fbtp_patchval;
}

static void
fbt_disable(void *arg, dtrace_id_t id, void *parg)
{
        fbt_probe_t *fbt = parg;
        struct modctl *ctl = fbt->fbtp_ctl;
[ ... ]
        for (; fbt != NULL; fbt = fbt->fbtp_next)
                *fbt->fbtp_patchpoint = fbt->fbtp_savedval;
}
```

uts/intel/dtrace/fbt.c

The core of DTrace – trap interposition

uts/intel/ia32/ml/exception.s

```
        /*
         * #BP
         */
        ENTRY_NP(brktrap)
#if defined(__amd64)
        cmpw    $KCS_SEL, 8(%rsp)
        je      bp_jmpud
#endif
        TRAP_NOERR(T_BPTFLT)    /* $3 */
        jmp     dtrace_trap
#if defined(__amd64)
bp_jmpud:
        /*
         * This is a breakpoint in the kernel -- it is very likely that this
         * is DTrace-induced.  To unify DTrace handling, we spoof this as an
         * invalid opcode (#UD) fault.  Note that #BP is a trap, not a fault --
         * we must decrement the trapping %rip to make it appear as a fault.
         * We then push a non-zero error code to indicate that this is coming
         * from #BP.
         */
        decq    (%rsp)
        push    $1                      /* error code -- non-zero for #BP */
        jmp     ud_kernel
#endif
        SET_SIZE(brktrap)
```

Usermode tracepoint hook

The core of DTrace – trap interposition

uts/intel/ia32/ml/exception.s

```
        ENTRY_NP(invoptrap)
        cmpw    $KCS_SEL, 8(%rsp)
        jne     ud_user

        push    $0                      /* error code -- zero for #UD */
ud_kernel:
        push    $0xdddd                 /* a dummy trap number */
        TRAP_PUSH
        movq    REGOFF_RIP(%rsp), %rdi
        movq    REGOFF_RSP(%rsp), %rsi
        movq    REGOFF_RAX(%rsp), %rdx
        pushq   (%rsi)
        movq    %rsp, %rsi
        call    dtrace_invop
        ALTENTRY(dtrace_invop_callsite)
        addq    $8, %rsp
        cmpl    $DTRACE_INVOP_PUSHL_EBP, %eax
        je      ud_push
        cmpl    $DTRACE_INVOP_LEAVE, %eax
        je      ud_leave
        cmpl    $DTRACE_INVOP_NOP, %eax
        je      ud_nop
        cmpl    $DTRACE_INVOP_RET, %eax
        je      ud_ret
        jmp     ud_trap
```

Kernel tracepoint hook

DTrace safety – catching stray pointers

uts/i86pc/ml/locore.s

```
        ENTRY_NP2(cmntrap, _cmntrap)

        TRAP_PUSH

        /*
         * We must first check if DTrace has set its NOFAULT bit.  This
         * regrettably must happen before the TRAPTRACE data is recorded,
         * because recording the TRAPTRACE data includes obtaining a stack
         * trace -- which requires a call to getpcstack() and may induce
         * recursion if an fbt::getpcstack: enabling is inducing the bad load.
         */
        movl    %gs:CPU_ID, %eax
        shlq    $CPU_CORE_SHIFT, %rax
        leaq    cpu_core(%rip), %r8
        addq    %r8, %rax
        movw    CPUC_DTRACE_FLAGS(%rax), %cx
        testw   $CPU_DTRACE_NOFAULT, %cx
        jnz     .dtrace_induced
[ ... ]
```

Check/Catch DTrace-caused faults

References

- DTrace OpenSolaris community: http://www.opensolaris.org/os/community/dtrace/
- Blogs of the DTrace authors:
  - Bryan Cantrill: http://blogs.sun.com/bmc/
  - Adam Leventhal: http://blogs.sun.com/ahl/
  - Mike Shapiro: http://blogs.sun.com/mws/
- OpenSolaris sourcecode archive: http://cvs.opensolaris.org/source/
- Solaris/x86 Internals and Crashdump analysis: http://www.genunix.org/gen/crashdump/index.html

“I am thirsty.”

Jesus
John 19:28-29

The DTrace backend on Solaris for x86/x64

Frank Hofmann
Frank.Hofmann@sun.com