DTrace

Example: Tracing Disk I/O

- Who is reading from my disk?

- ```
  io:::start {
      printf("%20s %2s %s\n",
         execname,
         args[0]->b_flags & B_READ ? "R" : "W",
         args[2]->fi_pathname);
  }
  ```
  ```

Example: Tracing Disk I/O

```
     Terminal  W ??/com.apple.Terminal.savedState/windows.plist
     Terminal  W ??/com.apple.Terminal.savedState/window_2.data
     Terminal  W ??/com.apple.Terminal.savedState/window_2.data
     Terminal  R ??/com.apple.Terminal.savedState/data.data
      Spotify  R ??/58/58dcba58af4289648d5dc8ee0334943b994b6409.file
      Spotify  R ??/58/58dcba58af4289648d5dc8ee0334943b994b6409.file
      Spotify  R ??/b6/b60f1e8ef8c2a557475a0c85ee14f0ead43b1a24.file
      Spotify  R ??/b6/b60f1e8ef8c2a557475a0c85ee14f0ead43b1a24.file
      Spotify  R ??/44/4493c6e5cbc9309c6a3d8f008e693a9fba0236a3.file
      Spotify  R ??/44/4493c6e5cbc9309c6a3d8f008e693a9fba0236a3.file
      Spotify  R ??/1a/1a256589090825bef9f2b0e3c39d10ddf39de1d5.file
      Spotify  R ??/1a/1a256589090825bef9f2b0e3c39d10ddf39de1d5.file
Sublime Text 2 W ??/pal-battle/logistic_regression.py
Sublime Text 2 W ??/Settings/Auto Save Temp Session.sublime_session
Sublime Text 2 W ??/com.sublimetext.2.savedState/data.data
Sublime Text 2 W ??/com.sublimetext.2.savedState/windows.plist
```

Example: Tracing Disk I/O

- Just tell me who uses the disk the most, ok?

- ```
  io:::start {
      @[execname] = sum(args[0]->b_bcount);
  }
  ```
  ```

Example: Tracing Disk I/O

Adobe Photoshop	4096
Dock	4096
Moom	4096
sandboxd	8192
syslogd	20480
Microsoft Power	196608
Sublime Text 2	270336
launchd	1179648
Spotify	1392640
Terminal	2383872
firefox	9355264

Example: Tracing Disk I/O

- Wait – what‘s that in MiBs?

- ```
  io:::start {
      @[execname] = sum(args[0]->b_bcount);
  }

  END {
      normalize(@, 1024 * 1024)
  }
  ```
  ```

Example: Tracing Disk I/O

```
Terminal                    0
fontd                       0
mds                         1
Spotify                     3
lssave                      4
kernel_task                 5
launchd                     6
plugin-containe            15
firefox                    22
```

Agenda

1. Example: Tracing Disk I/O
2. What is DTrace?
3. The D Language
4. Example: Most expensive syscalls
5. What DTrace cannot do
6. Comparison to strace and systemtap

- (find a more complex example on how to use DTrace in my written report)

What is DTrace?

- A **D**ynamic **Trac(e)**ing framework
- First developed by Sun Microsystems for Solaris
- Now also available for *BSD, OS X, and Linux
- For OS X, Instruments is a GUI utilizing DTrace
  - Easier to handle, but by far not as powerful

What is DTrace?

- Designed to give an integrated view of various indicators in
  - CPUs (e.g. caching, branches) – compare to PAPI
  - Memory (allocations, page faults, ...)
  - Kernel functions: File I/O, Networking, Scheduling
  - User functions
  - Script languages
  - Database operations (e.g. index reads or locking)

What is DTrace?

- Users shall be able to use indicators from different sources at the same time
  - „Which files are written by Firefox in the 100ms after a TCP call finishes and how long does this take?“
- The performance impact shall be negligible
- It shall be possible to instrument even sensitive areas such as the scheduler

The D Language

- DTrace programs are compiled from one or more functions, written in D
- DTrace D ist not D, the “proper“ programming language

- ```
  probe
  /predicate/
  {
      actions;
  }
  ```
  ```

The D Language: probes

- `probe /predicates/ {actions;}`
- `probe := provider:module:function:name`
- `fbt:mach_kernel:pthread_list_lock_spin:entry`
- Probes define the part of the system which you want to instrument
- A provider is a program that can call DTrace whenever something happens, e.g. a file is written

The D Language: probes

- `probe /predicates/ {actions;}`
- `probe := provider:module:function:name`

- Modules are used to categorize probes within a bigger provider
- Functions are functions from the kernel or your code
- Names describe the probe, such as *entry* for a function or *PAPI_l1_icm-user-500* for the CPU

The D Language: probes

- `probe /predicates/ {actions;}`
- Ruby, JavaScript, MySQL, ...
- My installation of OS X has 214,200 probes

- Everyone can write their own provider:

```
#include <sys/sdt.h>
DTRACE_PROBE2(provider, name, arg1, arg2)
```

The D Language: predicates

- `probe /predicates/ {actions;}`
- Just syscall:::entry is probably too noisy
- When a probe fires, DTrace checks whether the system‘s state matches the predicate
- E.g.: Filter by application:

```
/execname == "firefox"/
```

- E.g.: Filter by variables:

```
/start - timestamp > 50000/
```

The D Language: actions

- `probe /predicates/ {actions;}`
- Actions collect and print information
- Print it right away (like strace / truss) or store and aggregate it using variables
- Variables can be shared between actions and processes

The D Language: More features

- Access C data types, both in the kernel and your code:

```
args[2]->fi_pathname
```

- Show the stack as well as flowcharts:

```
-> thread_dispatch
  -> thread_tell_urgency
  <- thread_tell_urgency
  -> timer_call_enter1
  <- timer_call_enter1
  -> timer_queue_assign
  <- timer_queue_assign
  -> timer_call_enqueue_deadline_unlocked
    -> lck_mtx_lock_spin_always
    <- lck_mtx_lock_spin
    -> lck_mtx_unlock
    <- lck_mtx_unlock
  <- timer_call_enqueue_deadline_unlocked
<- thread_dispatch
```

The D Language: More features

- Statistic features, such as quantization:

```
syscall::write:return { @ = quantize(arg0) }
```

```
value  ------------- Distribution ------------- count
    0 |                                         0
    1 |@@@@@@@@@@@@@@@@@@@@@@                   134
    2 |                                         0
    4 |@@@@@@                                   38
    8 |                                         0
   16 |@                                        6
   32 |@@@                                      19
   64 |                                         0
  128 |@@@                                      20
  256 |                                         0
  512 |                                         1
 1024 |                                         1
 2048 |                                         0
 4096 |                                         0
 8192 |@@                                       12
16384 |@@@                                      19
32768 |@                                        6
65536 |                                         2
131072 |                                        0
```

Example: Most expensive syscalls

```
syscall:::entry /execname != "dtrace"/ {
    starttime[$pid] = timestamp
}

syscall:::return /execname != "dtrace"/ {
    duration = timestamp - starttime[$pid];
    @byf[probefunc] = avg(duration);
    @byp[execname] = avg(duration);
}

END {
    normalize(@byf, 1000); trunc(@byf, 10);
    normalize(@byp, 1000); trunc(@byp, 10);
    printf("Most expensive by function (in ms):"); printa(@byf);
    printf("Most expensive by program (in ms):"); printa(@byp);
}
```

Example: Most expensive syscalls

```
Most expensive by function called (in ms):
  writev                                                    222
  mmap                                                      239
  kevent                                                    254
  select_nocancel                                           420
  fsync                                                     577
  poll                                                      612
  psynch_cvwait                                            2089
  select                                                   2109
  sigsuspend                                               2616
  __semwait_signal                                     18292990
Most expensive by calling program (in ms):
  DashlaneAgent                                             830
  Dashlane                                                  898
  firefox                                                   953
  com.apple.qtkit                                          1147
  thunderbird                                              1171
  AppleSpell                                               1281
  Terminal                                                 1328
  AdobeCrashDaemo                                          1490
  coreaudiod                                               3673
  Spotify                                              17035744
```

What DTrace cannot do

- D is not turing-complete
  - No loops, not even branches (except for predicates)
- You won‘t ruin your program or panic the kernel
- But you also can‘t change your programs execution as e.g. in gdb

Comparison to systemtap

- SystemTap‘s language is a bit more flexible than DTrace‘s
- SystemTap supports DTrace syntax
- SystemTap can trace lines in C code
- SystemTap can manipulate the program execution
- DTrace can trace script languages, and user applications (there is a compatibility layer)
- DTrace is lockless
- DTrace is more flexible when tracing in the kernel

Comparison to strace

- DTrace is more flexible - strace / truss only monitors system calls
- The OS X version of strace, dtruss, is written as a DTrace script
- truss has a performance impact of 11500% in a multi-threaded environment
- dtruss slows down the program by only 30%

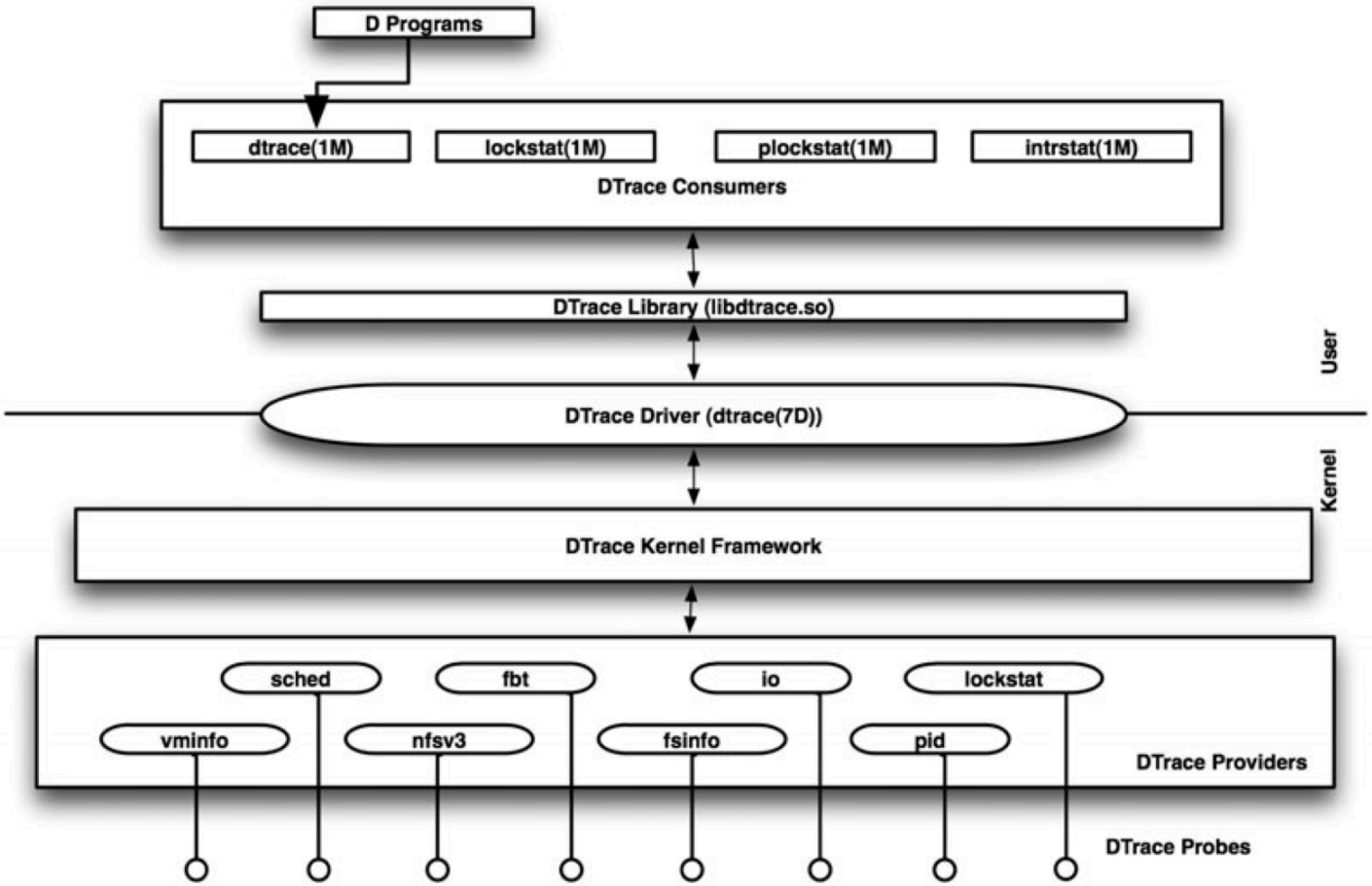
http://milek.blogspot.de/2009/04/truss1m-vs-dtrace1m.html

Summary

- Use DTrace to integrate data from different providers with low overhead
- Use it for Profiling and Reverse Engineering
- You can track a lot – or write your own provider
- This was just a short overview – the DTrace book has 1152 pages of excitement
- Thank you!

Backup Slides

DTrace Internals



Source: DTrace – Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD

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`

x86 breakpoint instruction