usenix LISA14

Linux Performance Analysis New Tools and Old Secrets

Brendan Gregg

Senior Performance Architect

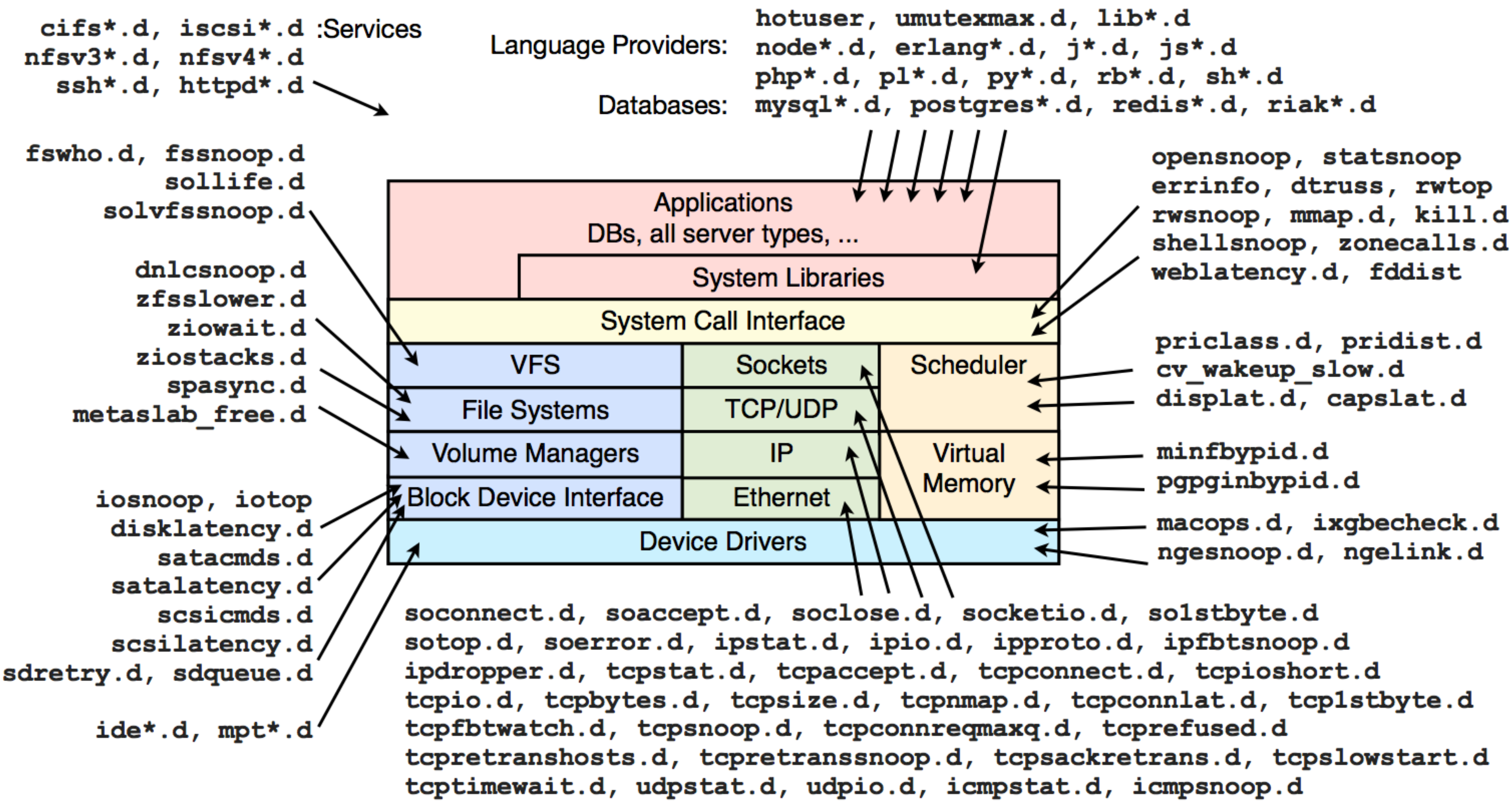
Performance Engineering Team

bgregg@netflix.com

@brendangregg

NETFLIX

Porting these to Linux…

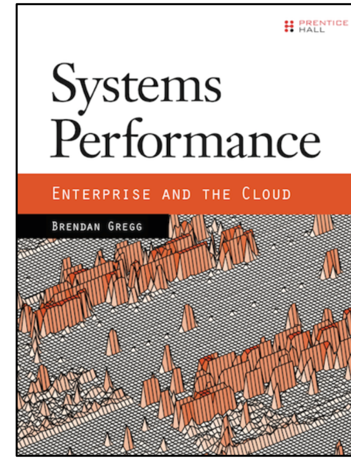


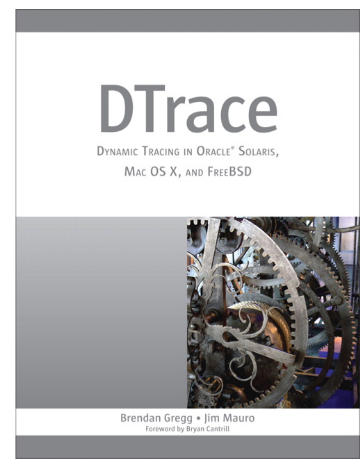
NETFLIX

- Massive Amazon EC2 Linux cloud
  - Tens of thousands of instances
  - Autoscale by ~3k each day
  - CentOS and Ubuntu
- FreeBSD for content delivery
  - Approx 33% of US Internet traffic at night
- Performance is critical
  - Customer satisfaction: >50M subscribers
  - $$$ price/performance
  - Develop tools for cloud-wide analysis; use server tools as needed

Brendan Gregg

- Senior Performance Architect, Netflix
  - Linux and FreeBSD performance
  - Performance Engineering team (@coburnw)
- Recent work:
  - Linux perf-tools: ftrace & perf_events
  - Testing of other tracers: eBPF
- Previously:
  - Performance of Linux, Solaris, ZFS, DBs, TCP/IP, hypervisors, …
  - Flame graphs, heat maps, methodologies, DTrace tools, DTraceToolkit





Agenda

1. Some one-liners
2. Background
3. Technology
4. Tools

1. Some one-liners

(cut to the chase!)

tpoint for disk I/O

- Who is creating disk I/O, and of what type?

```
# ./tpoint -H block:block_rq_insert
Tracing block:block_rq_insert. Ctrl-C to end.
# tracer: nop
#
#       TASK-PID    CPU#    TIMESTAMP  FUNCTION
#          | |       |          |         |
 flush-9:0-9318  [013] 1936182.007914: block_rq_insert: 202,16 W 0 () 160186560 + 8 [flush-9:0]
 flush-9:0-9318  [013] 1936182.007939: block_rq_insert: 202,16 W 0 () 280100936 + 8 [flush-9:0]
      java-9469  [014] 1936182.316184: block_rq_insert: 202,1 R 0 () 1319592 + 72 [java]
      java-9469  [000] 1936182.331270: block_rq_insert: 202,1 R 0 () 1125744 + 8 [java]
      java-9469  [000] 1936182.341418: block_rq_insert: 202,1 R 0 () 2699008 + 88 [java]
      java-9469  [000] 1936182.341419: block_rq_insert: 202,1 R 0 () 2699096 + 88 [java]
      java-9469  [000] 1936182.341419: block_rq_insert: 202,1 R 0 () 2699184 + 32 [java]
      java-9469  [000] 1936182.345870: block_rq_insert: 202,1 R 0 () 1320304 + 24 [java]
      java-9469  [000] 1936182.351590: block_rq_insert: 202,1 R 0 () 1716848 + 16 [java]
^C
Ending tracing...
```

- tpoint traces a given tracepoint. -H prints the header.

tpoint for disk I/O

- Who is creating disk I/O, and of what type?

```
# ./tpoint -H block:block_rq_insert
Tracing block:block_rq_insert. Ctrl-C to end.
# tracer: nop
#
#     TASK-PID    CPU#    TIMESTAMP  FUNCTION
#        | |       |          |          |
 flush-9:0-9318  [013] 1936182.007914: block_rq_insert: 202,16 W 0 () 160186560 + 8 [flush-9:0]
 flush-9:0-9318  [013] 1936182.007939: block_rq_insert: 202,16 W 0 () 280100936 + 8 [flush-9:0]
      java-9469  [014] 1936182.316184: block_rq_insert: 202,1 R 0 () 1319592 + 72 [java]
      java-9469  [000] 1936182.331270: block_rq_insert: 202,1 R 0 () 1125744 + 8 [java]
      java-9469  [000] 1936182.341418: block_rq_insert: 202,1 R 0 () 2699008 + 88 [java]
      java-9469  [000] 1936182.341419: block_rq_insert: 202,1 R 0 () 2699096 + 88 [java]
      java-9469  [000] 1936182.341419: block_rq_insert: 202,1 R 0 () 2699184 + 32 [java]
      java-9469  [000] 1936182.345870: block_rq_insert: 202,1 R 0 () 1320304 + 24 [java]
      java-9469  [000] 1936182.351590: block_rq_insert: 202,1 R 0 () 1716848 + 16 [java]
^C
Ending tracing...
```

tracepoint

type

size (sectors)

dev

offset

- tpoint traces a given tracepoint. -H prints the header.

tpoint -l

```
# ./tpoint -l
block:block_bio_backmerge
block:block_bio_bounce
block:block_bio_complete
block:block_bio_frontmerge
block:block_bio_queue
block:block_bio_remap
block:block_getrq
block:block_plug
block:block_rq_abort
block:block_rq_complete
block:block_rq_insert
block:block_rq_issue
block:block_rq_remap
block:block_rq_requeue
[…]
# ./tpoint –l | wc –l
1257
```

Listing tracepoints

- 1,257 tracepoints for this Linux kernel

tpoint -h

```
# ./tpoint -h
USAGE: tpoint [-hHsv] [-d secs] [-p PID] tracepoint [filter]
       tpoint -l
                 -d seconds      # trace duration, and use buffers
                 -p PID          # PID to match on I/O issue
                 -v              # view format file (don't trace)
                 -H              # include column headers
                 -l              # list all tracepoints
                 -s              # show kernel stack traces
                 -h              # this usage message

Note that these examples may need modification to match your kernel
version's function names and platform's register usage.
   eg,
       tpoint -l | grep open
                                 # find tracepoints containing "open"
       tpoint syscalls:sys_enter_open
                                 # trace open() syscall entry
       tpoint block:block_rq_issue
                                 # trace block I/O issue
       tpoint -s block:block_rq_issue
                                 # show kernel stacks

See the man page and example file for more info.
```

Some tpoint One-Liners

```
# List tracepoints
tpoint -l

# Show usage message
tpoint -h

# Trace disk I/O device issue with details:
tpoint block:block_rq_issue

# Trace disk I/O queue insertion with details:
tpoint block:block_rq_insert

# Trace disk I/O queue insertion with details, and include header:
tpoint -H block:block_rq_insert

# Trace disk I/O queue insertion, with kernel stack trace:
tpoint -s block:block_rq_insert

# Trace disk I/O queue insertion, for reads only:
tpoint -s block:block_rq_insert 'rwbs ~ "*R*"'

# Trace 1,000 disk I/O device issues:
tpoint block:block_rq_issue | head -1000
```

DEMO

One-Liners

- Useful
  - Keep a collection, copy-n-paste when needed
- Instructive
  - Teaches tool usage by-example
  - Can also show what use cases are most useful
- Intuitive
  - Follows Unix/POSIX/IEEE traditions/standards
  - getopts, -h for help, Ctrl-C to end, etc.
- Competitive
  - Why this tool? Demonstrate key, competitive, features.

DTrace One-Liners (Wikipedia)

Command line examples [edit]

DTrace scripts can be invoked directly from the command line, providing one or more probes and actions as arguments. Some examples:

```
# New processes with arguments
dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'

# Files opened by process
dtrace -n 'syscall::open*:entry { printf("%s %s",execname,copyinstr(arg0)); }'

# Syscall count by program
dtrace -n 'syscall:::entry { @num[execname] = count(); }'

# Syscall count by syscall
dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'

# Syscall count by process
dtrace -n 'syscall:::entry { @num[pid,execname] = count(); }'

# Disk size by process
dtrace -n 'io:::start { printf("%d %s %d",pid,execname,args[0]->b_bcount); }'

# Pages paged in by process
dtrace -n 'vminfo:::pgpgin { @pg[execname] = sum(arg0); }'
```

DTrace One-Liners (Wikipedia)

- Good examples: Useful, Instructive, Intuitive
- Taken from a longer list:
  - http://www.brendangregg.com/dtrace.html
  - (I wish they'd have included latency quantize as well)
- And, competitive
  - Linux couldn't do these in 2005

Linux One-Liners

- Porting them to Linux:

```
# New processes with arguments
execsnoop

# Files opened by process
opensnoop

# Syscall count by program
syscount

# Syscall count by syscall
funccount 'sys_*'

# Syscall count by process
syscount -v

# Disk size by process
iosnoop -Q

# Pages paged in by process
iosnoop –Qi '*R*'
```

perf-tools

- These Linux one-liners (and tpoint) are from my collection of Linux performance analysis tools
  - https://github.com/brendangregg/perf-tools
- New tools for old Linux secrets
  - Designed to work on 3.2+ kernels
  - Uses ftrace & perf_events, which have existed for years

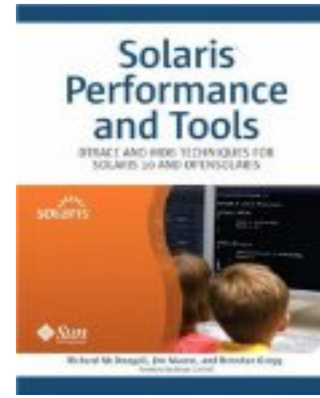
2. Background

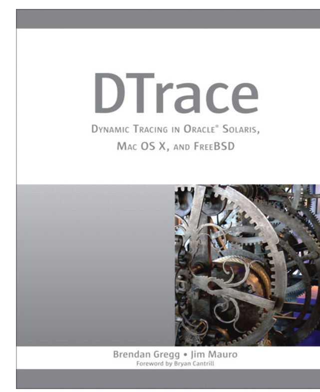
Background

- Linux tracing is:
  1. ftrace
  2. perf_events (perf)
  3. eBPF
  4. SystemTap
  5. ktap
  6. LTTng
  7. dtrace4linux
  8. Oracle Linux DTrace
  9. sysdig
- Understanding these is time consuming & complex
  – May be best told through personal experience...

Personal Experience

- Became a systems performance expert
  - Understood tools, metrics, inference, interpretation, OS internals
- Became a DTrace expert (2005)
  - Wrote scripts, books, blogs, courses
  - Helped Sun compete with Linux
- Began analyzing Linux perf (2011)
  - Tried SystemTap, dtrace4linux, ktap, …
  - Limited success, much pain & confusion
- Switched to Linux (2014)
  - And expected it to be hell (bring it on!)





The one that got away…

- Early on at Netflix, I had a disk I/O issue
  - Only 5 minutes to debug, then load is migrated
  - Collected iostat/sar, but needed a trace
    - No time to install any tracers (system was too slow)
  - Failed to solve the issue. Furious at Linux and myself.
  - Noticed the system did have this thing called ftrace…
- Ftrace?
  - Part of the Linux kernel
  - /sys interface for static and dynamic tracing
  - Already enabled on all our Linux 3.2 & 3.13 servers

WHY AM I NOT USING FTRACE ALREADY?

WHY IS NO ONE
USING FTRACE ALREADY?

Linux Secrets

- Re-focused on what Linux already has in-tree
  - ftrace & perf_events
  - These seem to be well-kept secrets: **no marketing**
- Clears up some confusion (and pain)
  - Instead of comparing 9 tracing options, it’s now:
    1. In-tree tools (currently: ftrace, perf_events)
    2. Everything else
  - Works for us; you may prefer picking one tracer
- Many of our tracing needs can now be met
  - Linux has been closing the tracing gap
    It’s not 2005 anymore

A Tracing Timeline

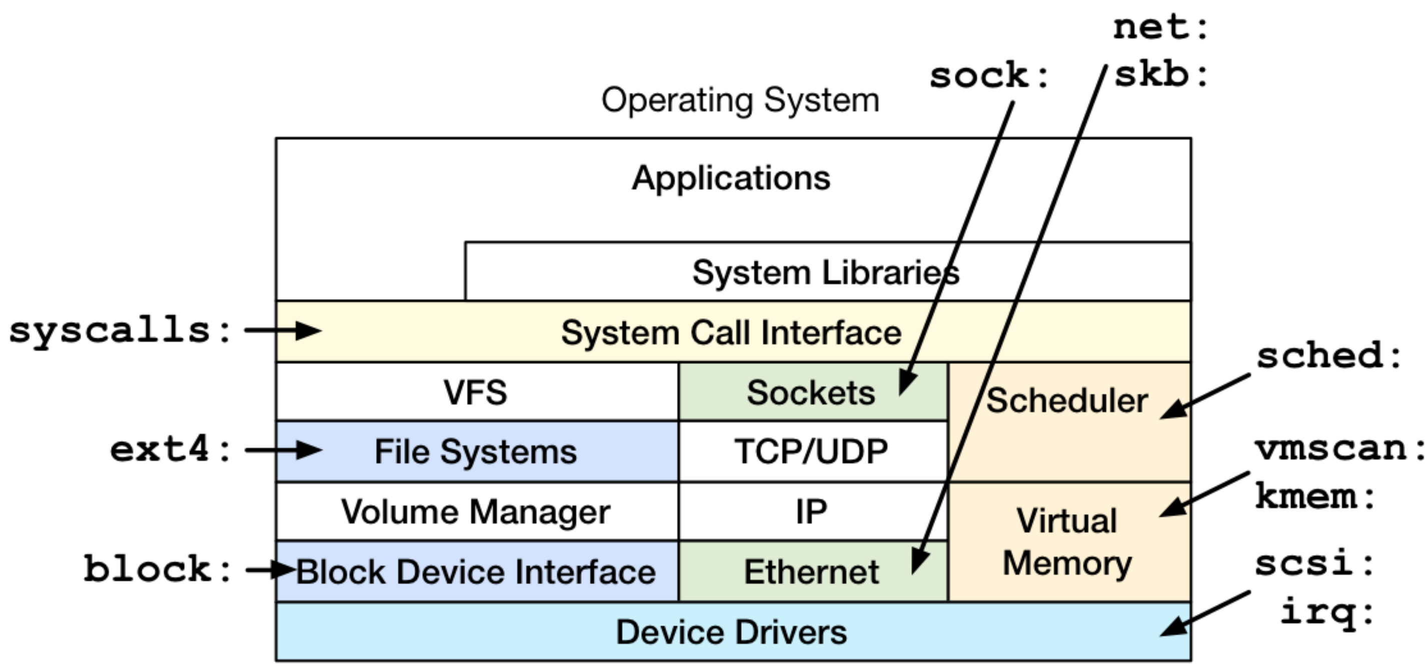
- 1990's: Static tracers, prototype dynamic tracers
- 2004: Linux kprobes (2.6.9)
  - Dynamic kernel tracing, difficult interface
- 2005: Solaris DTrace (s10)
  - Static & dynamic tracing, user & kernel level, production ready, easy to use, far better than anything of the time, and, marketed
- 2008: Linux ftrace (2.6.27)
- 2009: Linux perf (2.6.31)
- 2009: Linux tracepoints (2.6.32)
  - Static kernel tracing
- 2010-2014: ftrace & perf_events enhancements
- 2014: eBPF patches
  - Fast (JIT'd) in kernel aggregations and programs

3. Technology

Tracing Sources

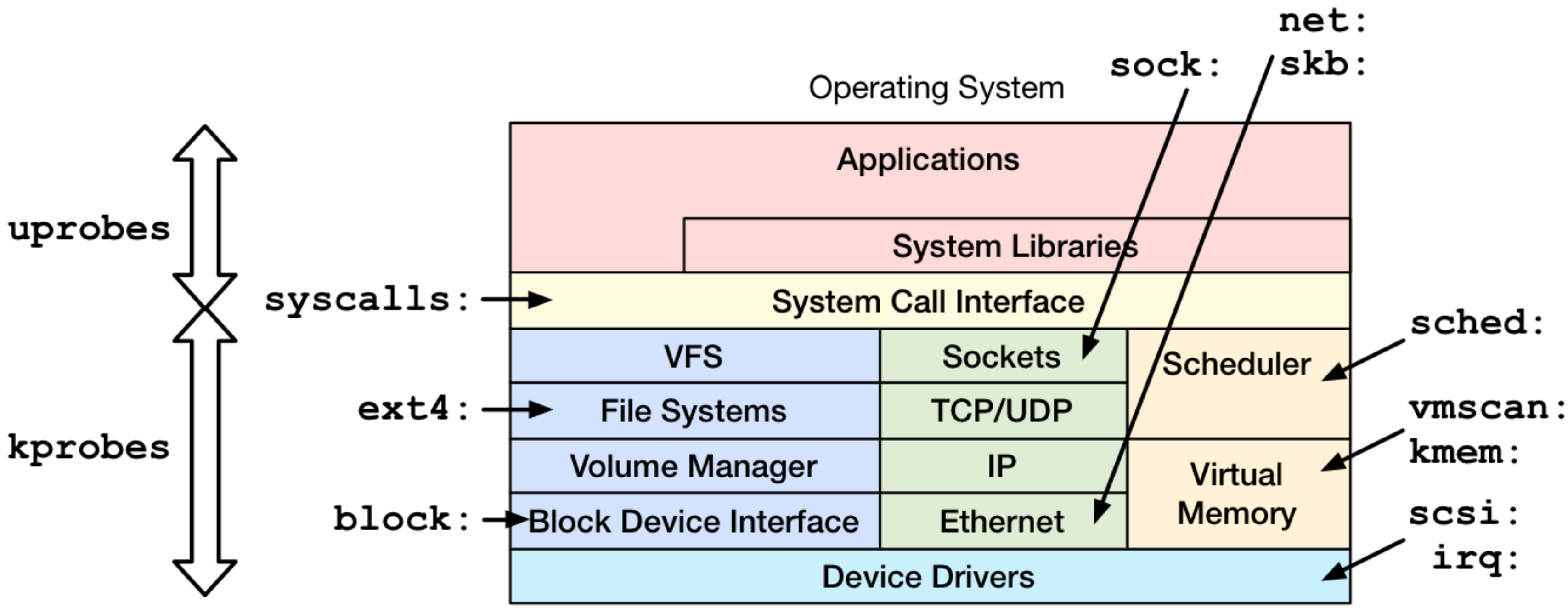
- Linux provides three tracing sources
  - **tracepoints**: kernel static tracing
  - **kprobes**: kernel dynamic tracing
  - **uprobes**: user-level dynamic tracing

Tracepoints



- Statically placed at logical places in the kernel
- Provides key event details as a "format" string
  - more on this later…

Probes



- kprobes: dynamic kernel tracing
  - function calls, returns, line numbers
- uprobes: dynamic user-level tracing

Tracers

- Tracing sources are used by the tracers
  - **In-tree**: ftrace, perf_events, eBPF (soon?)
  - **Out-of-tree**: SystemTap, ktap, LTTng, dtrace4linux, Oracle Linux DTrace, sysdig

ftrace

- A collection of tracing capabilities
  - Tracing, counting, graphing (latencies), filters
  - Uses tracepoints, kprobes
  - Not programmable (yet)
- Use via /sys/kernel/debug/tracing/…
  - Or use via front-end tools
- Added by Steven Rostedt and others since 2.6.27
  - Out of necessity for Steven’s real time work
- Can solve many perf issues

ftrace Interface

- Static tracing of block_rq_insert tracepoint

```
# cd /sys/kernel/debug/tracing
# echo 1 > events/block/block_rq_insert/enable
# cat trace_pipe
# echo 0 > events/block/block_rq_insert/enable
```

- Dynamic function tracing of tcp_retransmit_skb():

```
# cd /sys/kernel/debug/tracing
# echo tcp_retransmit_skb > set_ftrace_filter
# echo function > current_tracer
# cat trace_pipe
# echo nop > current_tracer
# echo > set_ftrace_filter
```

- Available tracing capabilities:

```
# cat available_tracers
blk function_graph mmiotrace wakeup_rt wakeup function nop
```

I Am SysAdmin (And So Can You!)

- What would a sysadmin do?

```
# cd /sys/kernel/debug/tracing
# echo tcp_retransmit_skb > set_ftrace_filter
# echo function > current_tracer
# cat trace_pipe
# echo nop > current_tracer
# echo > set_ftrace_filter
```

- Automate:

```
# functrace tcp_retransmit_skb
```

- Document:

```
# man functrace
[…]
SYNOPSIS
       functrace [-hH] [-p PID] [-d secs] funcstring
[…]
```

ftrace Interface

- Plus many more capabilities
  - buffered (trace) or live tracing (trace_pipe)
  - filters for conditional tracing
  - stack traces on events
  - function triggers to enable/disable tracing
  - functions with arguments (via kprobes)
- See Documentation/trace/ftrace.txt

perf_events

- Use via the "perf" command
- Add from linux-tools-common, …
  - Source code is in Linux: tools/perf
- Powerful multi-tool and profiler
  - interval sampling, CPU performance counter events
  - user and kernel dynamic tracing
  - kernel line tracing and local variables (debuginfo)
  - kernel filtering, and in-kernel counts (perf stat)
- Not very programmable, yet
  - limited kernel summaries. May improve with eBPF.

perf_events tracing

- Static tracing of block_rq_insert tracepoint:

```
# perf record -e block:block_rq_insert -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.172 MB perf.data (~7527 samples) ]

# perf script
# ========
# captured on: Wed Nov 12 20:50:05 2014
# hostname : bgregg-test-i-92b81f78
[…]
# ========
#
  java 9940 [015] 1199510.044783: block_rq_insert: 202,1 R 0 () 4783360 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783448 + 88 [java]
  java 9940 [015] 1199510.044786: block_rq_insert: 202,1 R 0 () 4783536 + 24 [java]
  java 9940 [000] 1199510.065194: block_rq_insert: 202,1 R 0 () 4864000 + 88 [java]
  java 9940 [000] 1199510.065195: block_rq_insert: 202,1 R 0 () 4864088 + 88 [java]
  java 9940 [000] 1199510.065196: block_rq_insert: 202,1 R 0 () 4864176 + 80 [java]
  java 9940 [000] 1199510.083745: block_rq_insert: 202,1 R 0 () 4864344 + 88 [java]
[…]
```

trace, dump, post-process

perf_events One-Liners

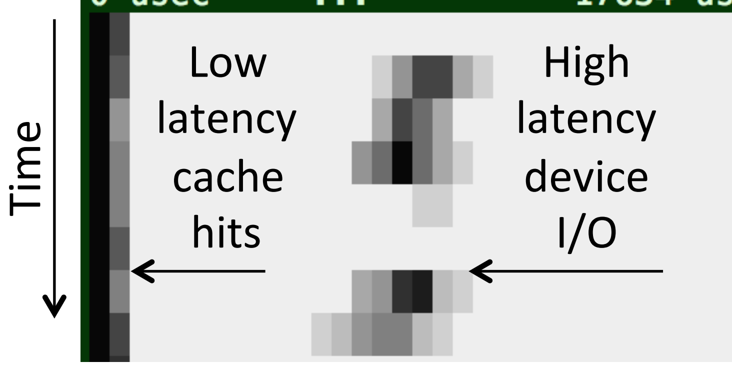
- Great one-liners. From http://www.brendangregg.com/perf.html:

```
# List all currently known events:
perf list

# Various basic CPU statistics, system wide, for 10 seconds:
perf stat -e cycles,instructions,cache-references,cache-misses -a sleep 10

# Count ext4 events for the entire system, for 10 seconds:
perf stat -e 'ext4:*' -a sleep 10

# Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds:
perf record -F 99 -ag -- sleep 10

# Sample CPU stack traces, once every 100 last level cache misses, for 5 seconds:
perf record -e LLC-load-misses -c 100 -ag -- sleep 5

# Trace all block device (disk I/O) requests with stack traces, until Ctrl-C:
perf record -e block:block_rq_issue –ag

# Add a tracepoint for the kernel tcp_sendmsg() function return:
perf probe 'tcp_sendmsg%return'

# Add a tracepoint for tcp_sendmsg, with size and socket state (needs debuginfo):
perf probe 'tcp_sendmsg size sk->__sk_common.skc_state'

# Show perf.data as a text report, with data coalesced and percentages:
perf report –n --stdio
```

eBPF

- Extended BPF: programs on tracepoints
  - High performance filtering: JIT
  - In-kernel summaries: maps
- eg, in-kernel latency heat map (showing bimodal):

```
root@bgregg-test-i-b7874e9d:/mnt/src/linux-3.16bpf2/samples/bpf# ./ex3
writing bpf-7 -> /sys/kernel/debug/tracing/events/block/block_rq_issue/filter
writing bpf-9 -> /sys/kernel/debug/tracing/events/block/block_rq_complete/filter
waiting for events to determine average latency...
  IO latency in usec
  █ - many events with this latency
  ░ - few events
0 usec     ...          17634 usec
```




```
captured=270 missed=0 max_lat=0 usec
captured=3694 missed=0 max_lat=0 usec
captured=3485 missed=12 max_lat=18902 usec
captured=3541 missed=19 max_lat=82377 usec
captured=1945 missed=33 max_lat=24441 usec
captured=1636 missed=0 max_lat=0 usec
captured=3441 missed=18 max_lat=51263 usec
captured=2864 missed=71 max_lat=60497 usec
```

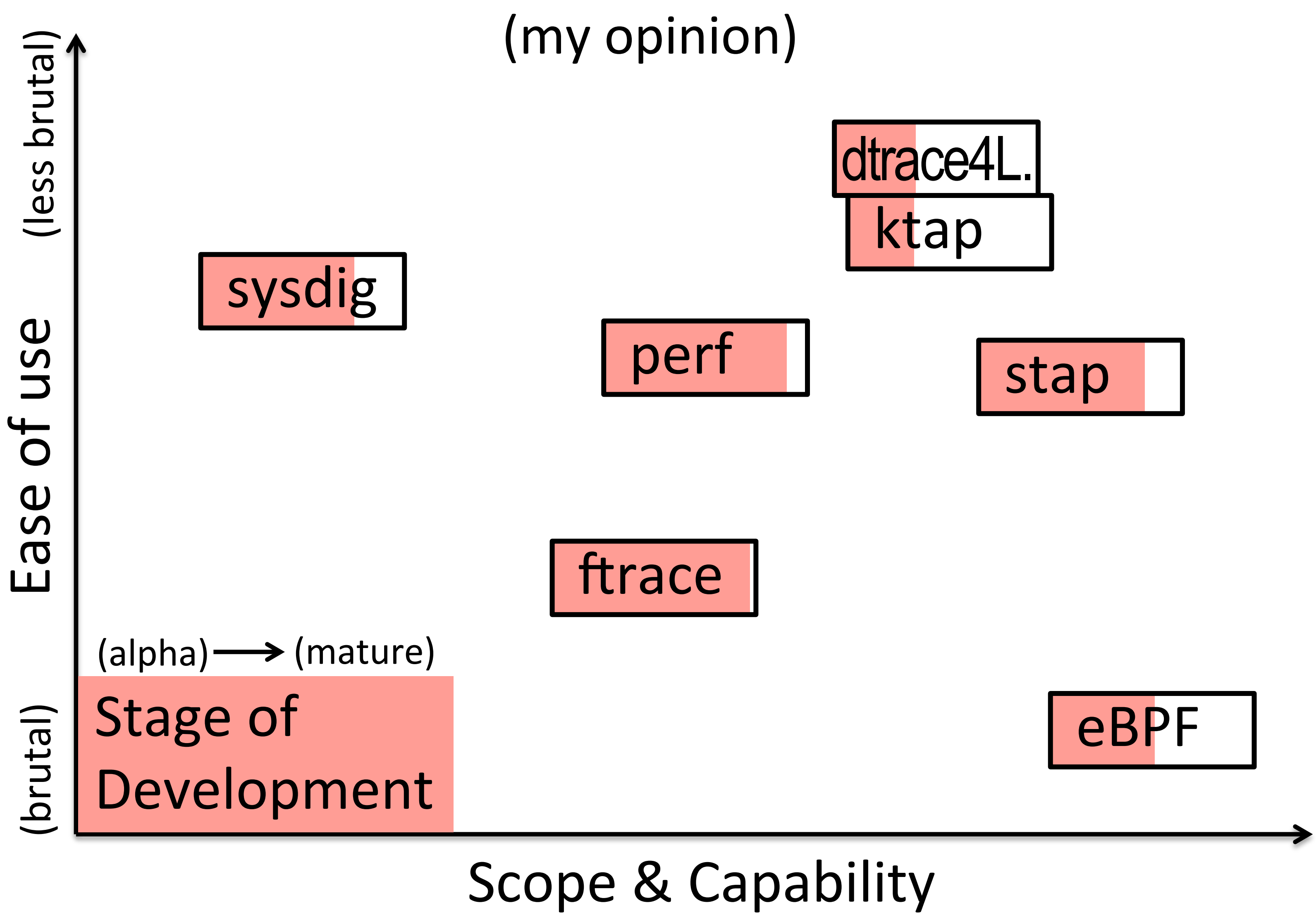
eBPF

- Created by Alexei Starovoitov
- Gradually being included in Linux (see lkml)
- Has been difficult to program directly
  - Other tools can become front-ends: ftrace, perf_events, SystemTap, ktap?

Other Tracers

- Discussion:
  - SystemTap
  - ktap
  - LTTng
  - DTrace ports
  - sysdig

The Tracing Landscape, Nov 2014



4. Tools

Tools

one-liners:	*many*
front-end tools:	`perf`, `trace-cmd`, perf-tools
tracing frameworks:	ftrace, perf_events, eBPF, …
back-end instrumentation:	tracepoints, kprobes, uprobes

Front-end Tools

- For ftrace
  - `trace-cmd` by Steven Rostedt
  - perf-tools: `tpoint`, `iosnoop`, `execsnoop`, `kprobe`, …
- For perf_events
  - `perf` (how perf_events is commonly used)
  - perf-tools: eg, `syscount`, `bitesize`, …
- For eBPF
  - *still evolving*
  - Could be used via ftrace, perf_events, SystemTap, ktap?

Tool Types

- Multi-tools
  - `perf`
  - `trace-cmd`
  - perf-tools: `tpoint`, `kprobe`, `funccount`, …
  - Narrow audience: engineers & developers who can take the time to learn them; others via canned one-liners
- Single purpose tools
  - perf-tools: `iosnoop`, `execsnoop`, `bitesize`, ...
  - Wide audience: sysadmins, developers, everyone
  - Unix philosophy: do one thing and do it well

perf-tools

- A collection of tools for both ftrace and perf_events
  - https://github.com/brendangregg/perf-tools
- Each tool has:
  - The tool itself
  - A man page
  - An examples file
  - A symlink under /bin

```
perf-tools> ls -l execsnoop bin/execsnoop man/man8/execsnoop.8 \
    examples/execsnoop_example.txt
lrwxr-xr-x  1 bgregg  1001    12 Jul 26 16:35 bin/execsnoop@ -> ../execsnoop
-rw-r--r--+ 1 bgregg  1001  2533 Jul 31 15:34 examples/execsnoop_example.txt
-rwxrwxr-x+ 1 bgregg  1001  8529 Jul 31 15:36 execsnoop*
-rw-r--r--+ 1 bgregg  1001  3497 Jul 31 22:40 man/man8/execsnoop.8
```

perf-tools

- WARNING: These are unsupported hacks
  - May not work on some kernel versions without tweaking
    - I've tried to use stable approaches as much as possible, but it isn't always possible
  - May have higher overhead than expected
    - Extreme case: slow target app by 5x
    - See the "OVERHEAD" section in the man pages
    - If this is a problem, re-implement tool in C using perf_events style interface (dynamic buffered)
  - Over time this will improve as Linux includes more tracing features, and workarounds can be rewritten

Dependencies

- Depends on your Linux distribution
  - On our Ubuntu servers, perf-tools just works
- Might need
  - mount -t debugfs none /sys/kernel/debug
  - CONFIG_DEBUG_FS, CONFIG_FUNCTION_PROFILER, CONFIG_FTRACE, CONFIG_KPROBES, …
  - awk (awk, mawk, or gawk), perl

perf-tools

- Current single purpose tools (Nov 2014):

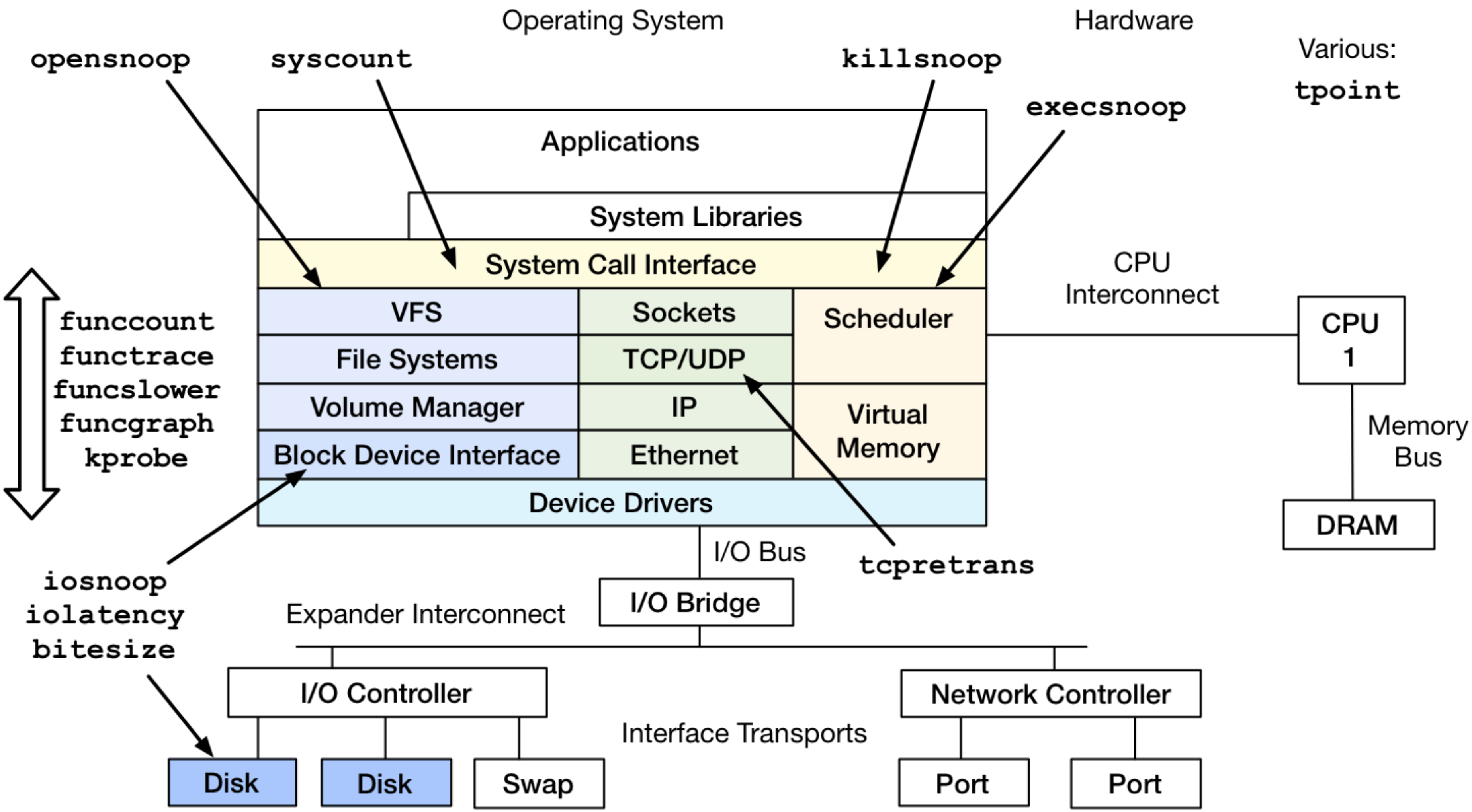
Tool	Description
iosnoop	trace disk I/O with details including latency
iolatency	summarize disk I/O latency as a histogram
execsnoop	trace process exec() with command line argument details
opensnoop	trace open() syscalls showing filenames
killsnoop	trace kill() signals showing process and signal details
syscount	count syscalls by syscall or process
disk/bitesize	histogram summary of disk I/O size
net/tcpretrans	show TCP retransmits, with address and other details
tools/reset-ftrace	reset ftrace state if needed

perf-tools

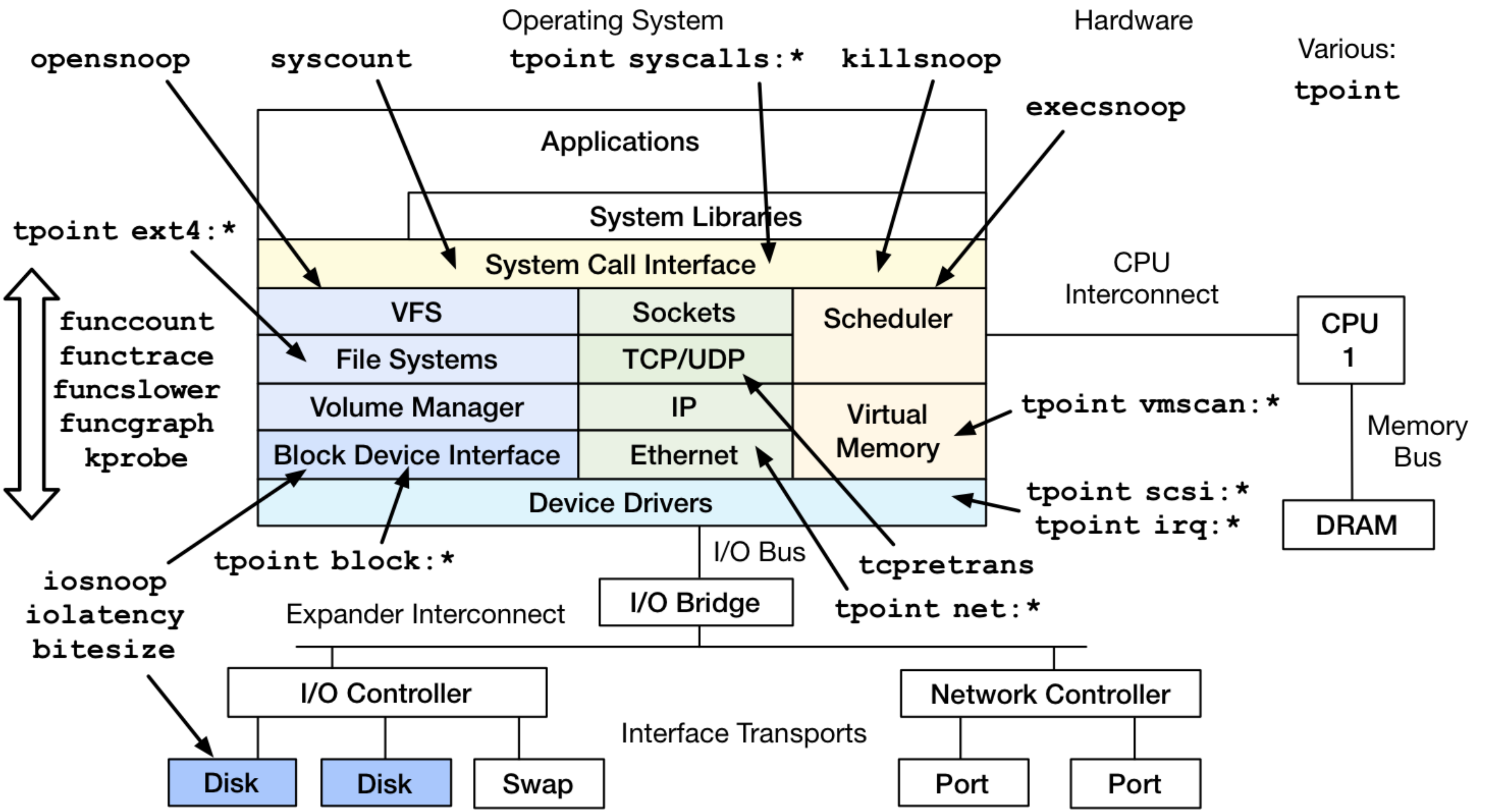
- Current multi-tools (Nov 2014):

Tool	Description
system/tpoint	trace a given tracepoint
kernel/funccount	count kernel function calls, matching a string
kernel/functrace	trace kernel function calls, matching a string
kernel/funcslower	trace kernel functions slower than a threshold
kernel/funcgraph	graph kernel function calls, showing children and times
kernel/kprobe	dynamically trace a kernel function call or its return, with variables

perf-tools (so far…)



perf-tools (so far…)



DEMO

iosnoop

- Block I/O (disk) events with latency:

```
# ./iosnoop –ts
Tracing block I/O. Ctrl-C to end.
STARTs          ENDs            COMM         PID    TYPE DEV      BLOCK        BYTES LATms
5982800.302061 5982800.302679 supervise  1809   W    202,1    17039600     4096    0.62
5982800.302423 5982800.302842 supervise  1809   W    202,1    17039608     4096    0.42
5982800.304962 5982800.305446 supervise  1801   W    202,1    17039616     4096    0.48
5982800.305250 5982800.305676 supervise  1801   W    202,1    17039624     4096    0.43
[…]
```

```
# ./iosnoop –h
USAGE: iosnoop [-hQst] [-d device] [-i iotype] [-p PID] [-n name] [duration]
                 -d device       # device string (eg, "202,1)
                 -i iotype       # match type (eg, '*R*' for all reads)
                 -n name         # process name to match on I/O issue
                 -p PID          # PID to match on I/O issue
                 -Q              # include queueing time in LATms
                 -s              # include start time of I/O (s)
                 -t              # include completion time of I/O (s)
                 -h              # this usage message
                 duration        # duration seconds, and use buffers
[…]
```

iolatency

- Block I/O (disk) latency distributions:

```
# ./iolatency
Tracing block I/O. Output every 1 seconds. Ctrl-C to end.

  >=(ms) .. <(ms)   : I/O      |Distribution                          |
       0 -> 1       : 1144     |######################################|
       1 -> 2       : 267      |#########                             |
       2 -> 4       : 10       |#                                     |
       4 -> 8       : 5        |#                                     |
       8 -> 16      : 248      |#########                             |
      16 -> 32      : 601      |####################                  |
      32 -> 64      : 117      |####                                  |
[…]
```

- User-level processing sometimes can't keep up
  - Over 50k IOPS. Could buffer more workaround, but would prefer in-kernel aggregations

opensnoop

- Trace open() syscalls showing filenames:

```
# ./opensnoop -t
Tracing open()s. Ctrl-C to end.
TIMEs              COMM         PID     FD FILE
4345768.332626     postgres     23886   0x8 /proc/self/oom_adj
4345768.333923     postgres     23886   0x5 global/pg_filenode.map
4345768.333971     postgres     23886   0x5 global/pg_internal.init
4345768.334813     postgres     23886   0x5 base/16384/PG_VERSION
4345768.334877     postgres     23886   0x5 base/16384/pg_filenode.map
4345768.334891     postgres     23886   0x5 base/16384/pg_internal.init
4345768.335821     postgres     23886   0x5 base/16384/11725
4345768.347911     svstat       24649   0x4 supervise/ok
4345768.347921     svstat       24649   0x4 supervise/status
4345768.350340     stat         24651   0x3 /etc/ld.so.cache
4345768.350372     stat         24651   0x3 /lib/x86_64-linux-gnu/libselinux…
4345768.350460     stat         24651   0x3 /lib/x86_64-linux-gnu/libc.so.6
4345768.350526     stat         24651   0x3 /lib/x86_64-linux-gnu/libdl.so.2
4345768.350981     stat         24651   0x3 /proc/filesystems
4345768.351182     stat         24651   0x3 /etc/nsswitch.conf
[…]
```

funcgraph

- Trace a graph of kernel code flow:

```
# ./funcgraph -Htp 5363 vfs_read
Tracing "vfs_read" for PID 5363... Ctrl-C to end.
# tracer: function_graph
#
#     TIME        CPU  DURATION                  FUNCTION CALLS
#      |          |     |   |                     |   |   |   |
4346366.073832 |   0)               |  vfs_read() {
4346366.073834 |   0)               |    rw_verify_area() {
4346366.073834 |   0)               |      security_file_permission() {
4346366.073834 |   0)               |        apparmor_file_permission() {
4346366.073835 |   0)   0.153 us    |          common_file_perm();
4346366.073836 |   0)   0.947 us    |        }
4346366.073836 |   0)   0.066 us    |        __fsnotify_parent();
4346366.073836 |   0)   0.080 us    |        fsnotify();
4346366.073837 |   0)   2.174 us    |      }
4346366.073837 |   0)   2.656 us    |    }
4346366.073837 |   0)               |    tty_read() {
4346366.073837 |   0)   0.060 us    |      tty_paranoia_check();
[…]
```

funccount

- Count a kernel function call rate:

```
# ./funccount -i 1 'bio_*'
Tracing "bio_*"... Ctrl-C to end.

FUNC                                    COUNT
bio_attempt_back_merge                     26
bio_get_nr_vecs                           361
bio_alloc                                 536
bio_alloc_bioset                          536
bio_endio                                 536
bio_free                                  536
bio_fs_destructor                         536
bio_init                                  536
bio_integrity_enabled                     536
bio_put                                   729
bio_add_page                             1004

[...]
```

Counts are in-kernel, for low overhead

  – -i: set an output interval (seconds), otherwise until Ctrl-C

kprobe

- Just wrapping capabilities eases use. Eg, kprobes:

```
# ./kprobe 'p:open do_sys_open filename=+0(%si):string' 'filename ~ "*stat"'
Tracing kprobe myopen. Ctrl-C to end.
        postgres-1172  [000] d... 6594028.787166: open: (do_sys_open
+0x0/0x220) filename="pg_stat_tmp/pgstat.stat"
        postgres-1172  [001] d... 6594028.797410: open: (do_sys_open
+0x0/0x220) filename="pg_stat_tmp/pgstat.stat"
        postgres-1172  [001] d... 6594028.797467: open: (do_sys_open
+0x0/0x220) filename="pg_stat_tmp/pgstat.stat"
^C
Ending tracing...
```

- By some definition of “ease”. Would like easier symbol usage, instead of +0(%si).

tpoint One-Liners

```
# List tracepoints
tpoint -l

# Trace disk I/O device issue with details:
tpoint block:block_rq_issue

# Trace disk I/O queue insertion, with kernel stack trace:
tpoint -s block:block_rq_insert

# Show output format string and filter variables:
tpoint -v block:block_rq_insert

# Trace disk I/O queue insertion, for reads only:
tpoint block:block_rq_insert 'rwbs ~ "*R*"'

# Trace 1,000 disk I/O device issues:
tpoint block:block_rq_issue | head -1000

# Trace syscall open():
tpoint syscalls:sys_enter_open
```

Tracepoint Format Strings

```
# ./tpoint -H block:block_rq_insert
Tracing block:block_rq_insert. Ctrl-C to end.
# tracer: nop
#
#       TASK-PID    CPU#    TIMESTAMP  FUNCTION
#          | |       |          |         |
      java-9469  [000] 1936182.331270: block_rq_insert: 202,1 R 0 () 1125744 + 8 [java]
```

Comes from

```
include/trace/events/block.h:
DECLARE_EVENT_CLASS(block_rq,
[...]
 TP_printk("%d,%d %s %u (%s) %llu + %u [%s]",
                  MAJOR(__entry->dev), MINOR(__entry->dev),
                  __entry->rwbs, __entry->bytes, __get_str(cmd),
                  (unsigned long long)__entry->sector,
                  __entry->nr_sector, __entry->comm)
```

– Kernel source may be the only docs for tracepoints

Tracepoint Format Strings

- Can also use tpoint -v for reminders:

```
# ./tpoint -v block:block_rq_issue
name: block_rq_issue
ID: 942
format:
	field:unsigned short common_type;	offset:0;	size:2; signed:0;
	field:unsigned char common_flags;	offset:2;	size:1; signed:0;
	field:unsigned char common_preempt_count;	offset:3;	size:1; signed:0;
	field:int common_pid;	offset:4;	size:4; signed:1;

	field:dev_t dev;	offset:8;	size:4; signed:0;
	field:sector_t sector;	offset:16;	size:8; signed:0;
	field:unsigned int nr_sector;	offset:24;	size:4; signed:0;
	field:unsigned int bytes;	offset:28;	size:4; signed:0;
	field:char rwbs[8];	offset:32;	size:8; signed:1;
	field:char comm[16];	offset:40;	size:16;	signed:1;
	field:__data_loc char[] cmd;	offset:56;	size:4; signed:1;

print fmt: "%d,%d %s %u (%s) %llu + %u [%s]", ((unsigned int) ((REC->dev) >> 20)),
((unsigned int) ((REC->dev) & ((1U << 20) - 1))), REC->rwbs, REC->bytes, __get_str(cmd),
(unsigned long long)REC->sector, REC->nr_sector, REC->comm
```

  - Fields can be used in filters. Eg:
    - `tpoint block:block_rq_insert 'rwbs ~ "*R*"'`

funccount One-Liners

```
# Count all block I/O functions:
funccount 'bio_*'

# Count all block I/O functions, print every 1 second:
funccount -i 1 'bio_*'

# Count all vfs functions for 5 seconds:
funccount -t 5 'vfs*'

# Count all "tcp_" functions, printing the top 5 every 1 second:
funccount -i 1 -t 5 'tcp_*'

# Count all "ext4*" functions for 10 seconds, print the top 25:
funccount -t 25 -d 10 'ext4*'

# Check which I/O scheduler is in use:
funccount -i 1 'deadline*'
funccount -i 1 'noop*'

# Count syscall types, summarizing every 1 second (one of):
funccount -i 1 'sys_*'
funccount -i 1 'SyS_*'
```

kprobe One-Liners

```
# Trace calls to do_sys_open():
kprobe p:do_sys_open

# Trace returns from do_sys_open(), and include column header:
kprobe -H r:do_sys_open

# Trace do_sys_open() return as "myopen" alias, with return value:
kprobe 'r:myopen do_sys_open $retval'

# Trace do_sys_open() calls, and print register %cx as uint16 "mode":
kprobe 'p:myopen do_sys_open mode=%cx:u16'

# Trace do_sys_open() calls, with register %si as a "filename" string:
kprobe 'p:myopen do_sys_open filename=+0(%si):string'

# Trace do_sys_open() filenames, when they match "*stat":
kprobe 'p:myopen do_sys_open filename=+0(%si):string' 'filename ~ "*stat"'

# Trace tcp_init_cwnd() with kernel call stack:
kprobe -s 'p:tcp_init_cwnd'

# Trace tcp_sendmsg() for PID 81 and for 5 seconds (buffered):
kprobe -p 81 -d 5 'p:tcp_sengmsg'
```

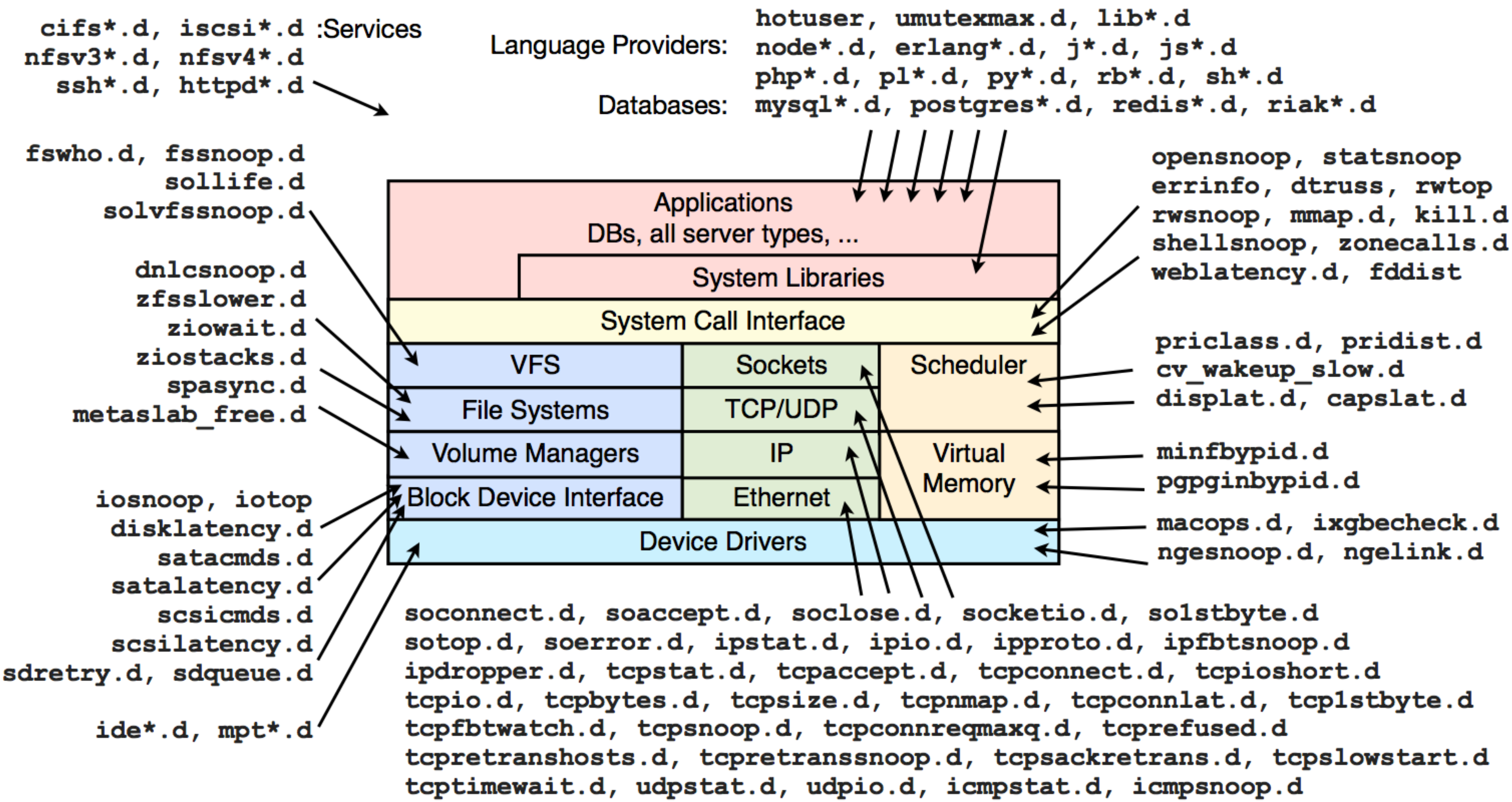
perf-tools Internals

- If you ever read the tool source code…
  - They are designed to be:
    A. As stable as possible
    B. Use fewest dependencies
    C. Short, temporary, programs
       - They may be rewritten when newer tracing features exist
    D. Mindful of overheads
       - C implementations, like perf_event's dynamic buffered approach, would be better. But see (C).
  - Many tools have:
    - SIGPIPE handling, so "| head -100" etc.
    - `-d duration`, which buffers output, lowering overhead
- In order of preference: bash, awk, perl5/python/…

The AWK Wars

- Tools may make use of gawk, mawk, or awk
  - Will check what is available, and pick the best option
  - mawk is faster, but (currently) less featured
- Example issues encountered:
  - gawk has strftime(), mawk doesn't
    - Test: awk 'BEGIN { print strftime("%H:%M:%S") }'
  - gawk honors fflush(), mawk doesn't
  - mawk's "-W interactive" flushes too often: every column
  - gawk and mawk have inconsistent handlings of hex numbers:
    - prints "16 0" in mawk : mawk 'BEGIN { printf "%d %d\n", "0x10", 0x10 }'
    - prints "0 16" in gawk : gawk 'BEGIN { printf "%d %d\n", "0x10", 0x10 }'
    - prints "16 16" in gawk: gawk --non-decimal-data 'BEGIN { printf "%d %d \n", "0x10", 0x10 }'

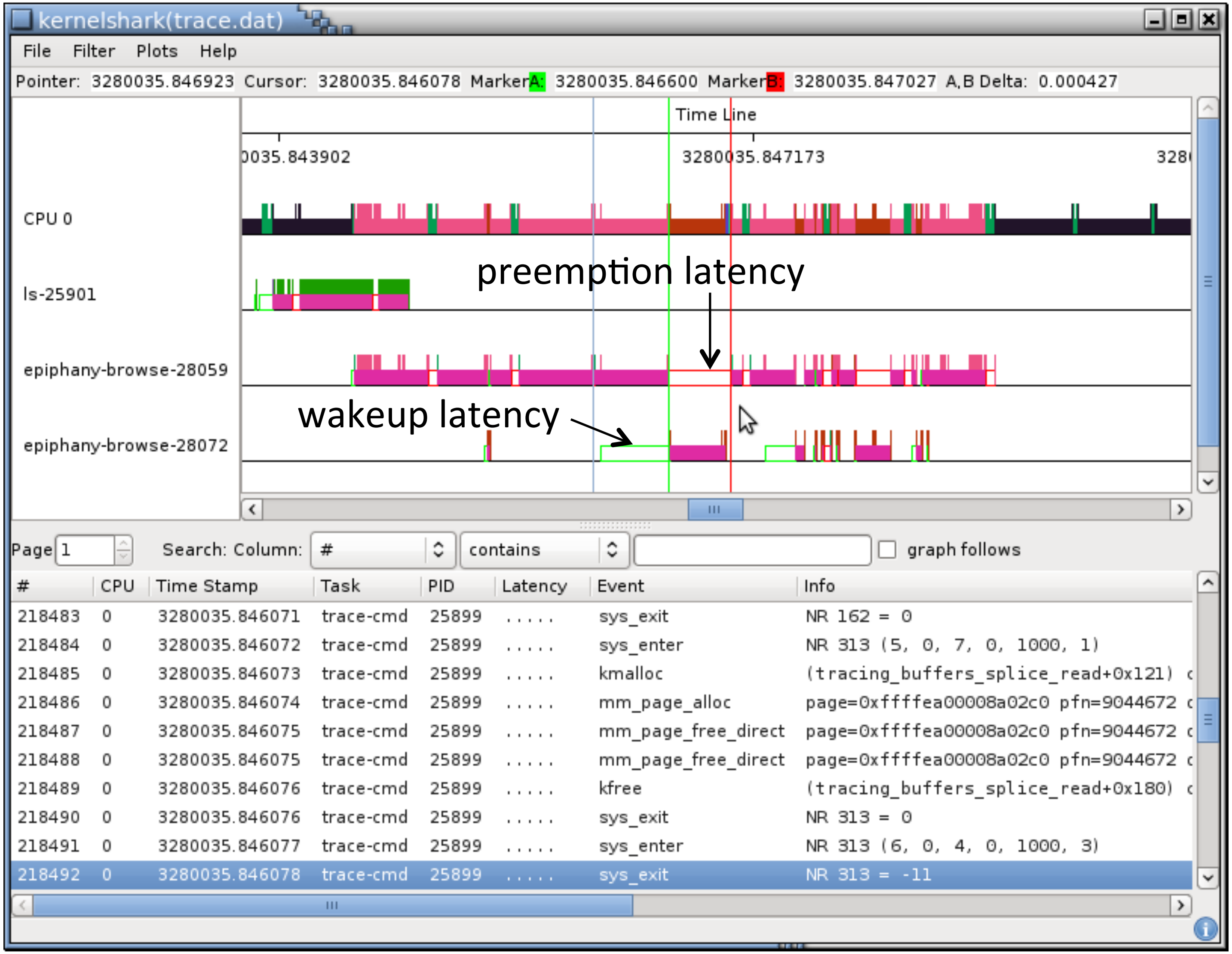
Much more to do… Porting more DTrace scripts



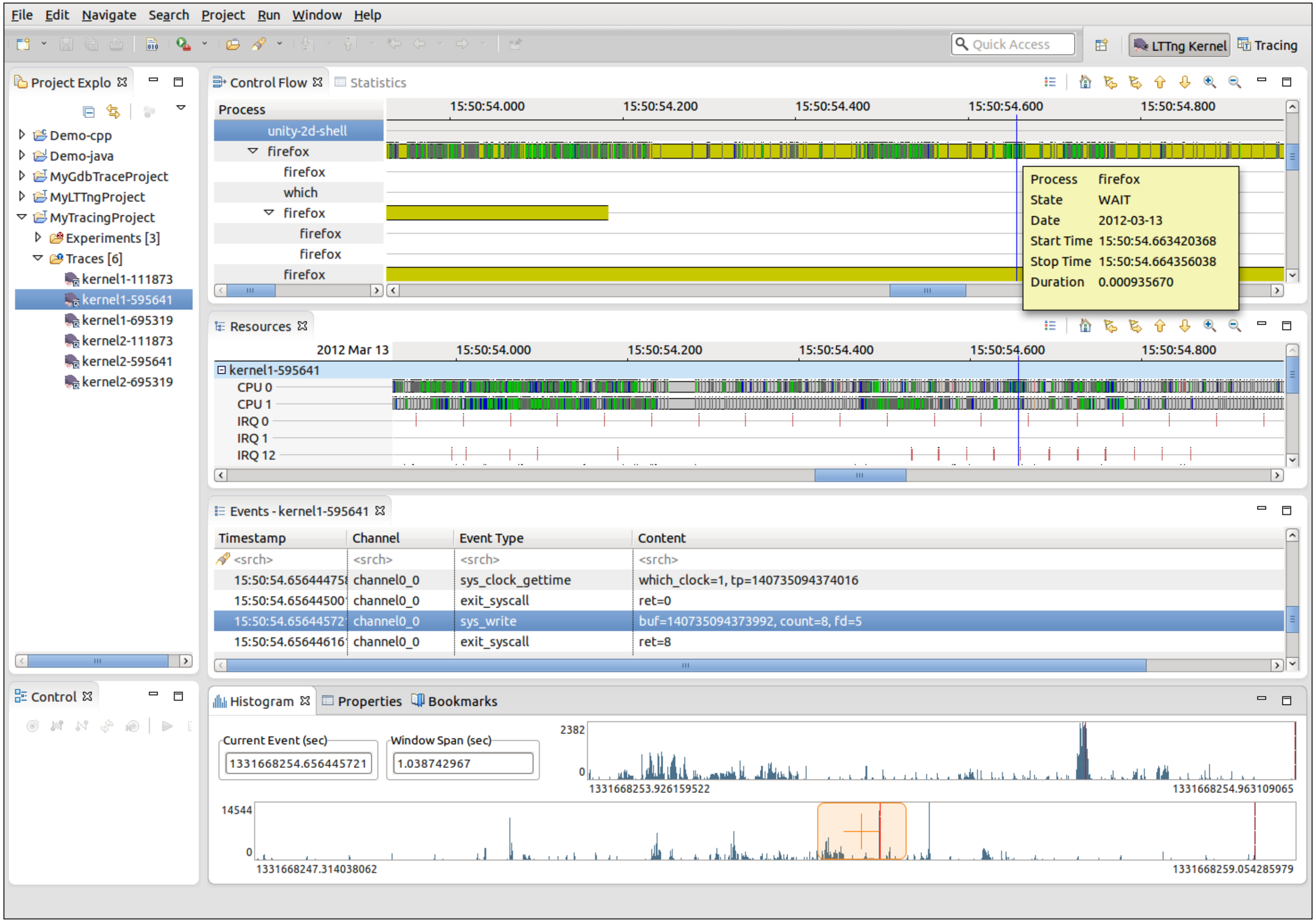
Some Visual Tools

- kernelshark
  - For ftrace
- Trace Compass
  - To visualize LTTng (and more) time series trace data
- Flame graphs
  - For any profiles with stack traces
- Heat maps
  - To show distributions over time

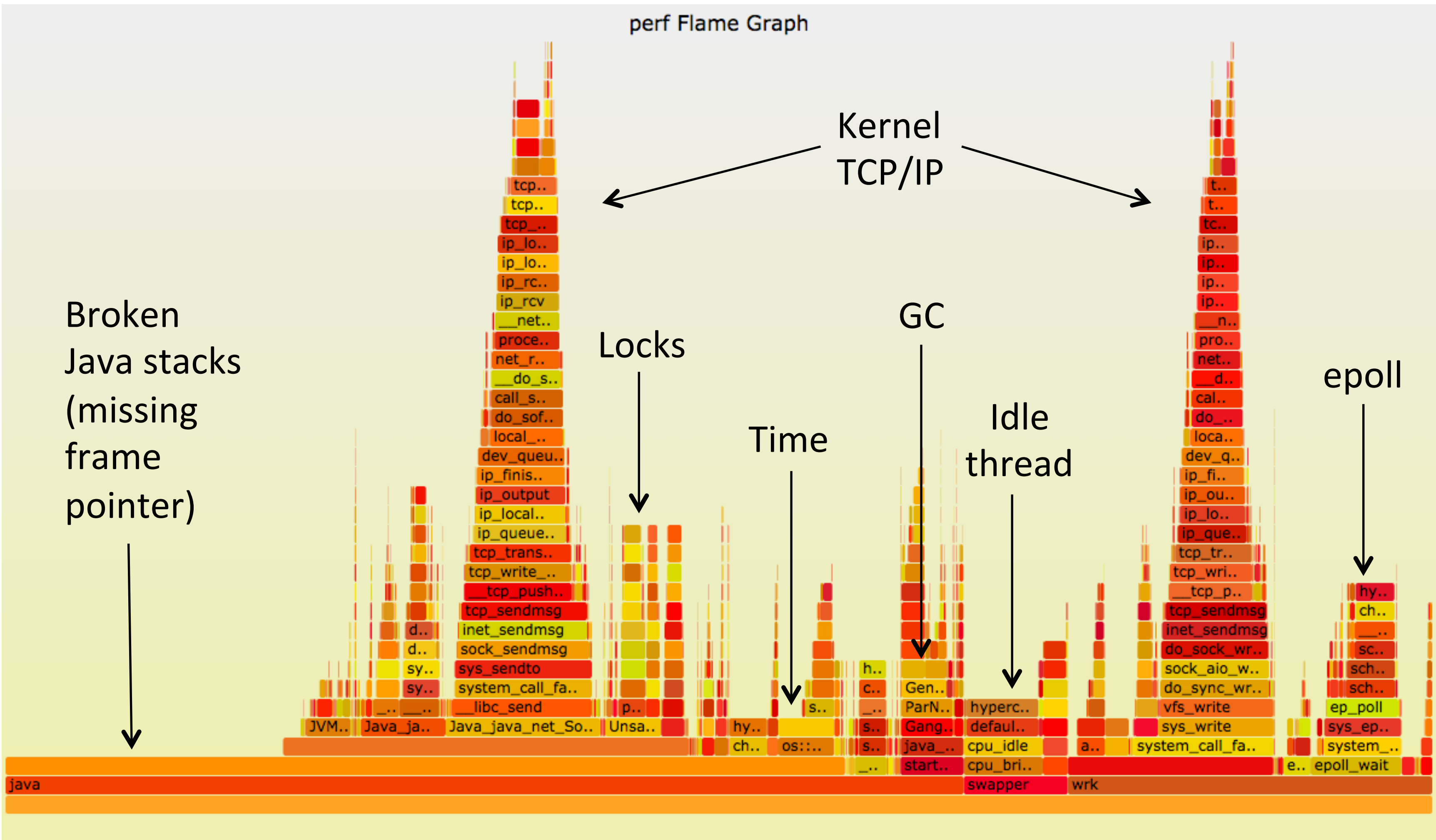
Kernelshark



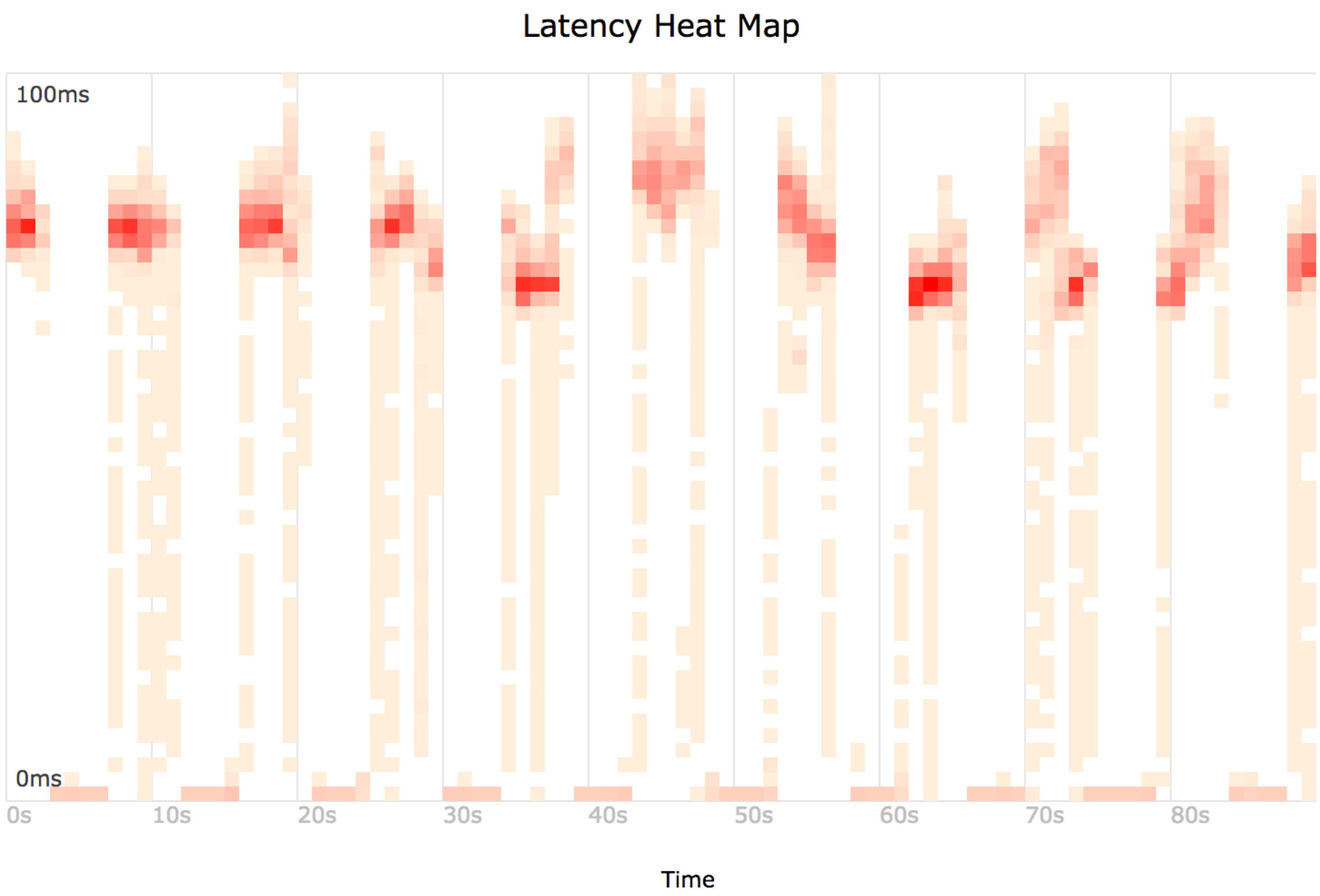
Trace Compass



perf CPU Flame Graph



perf Block I/O Latency Heat Map



Summary

1. Some one-liners
2. Background
3. Technology
4. Tools

Most important take away: Linux can serve many tracing needs today with ftrace & perf_events

Acks

- http://en.wikipedia.org/wiki/DTrace
- http://generalzoi.deviantart.com/art/Pony-Creator-v3-397808116 and Deirdré Straughan for the tracing pony mascots
- I Am SysAdmin (And So Can You!), Ben Rockwood, LISA14
- http://people.redhat.com/srostedt/kernelshark/HTML/ kernelshark screenshot
- https://projects.eclipse.org/projects/tools.tracecompass Trace Compass screenshot

Links

- perf-tools
  - https://github.com/brendangregg/perf-tools
  - http://lwn.net/Articles/608497/
- perf_events
  - https://perf.wiki.kernel.org/index.php/Main_Page
  - http://www.brendangregg.com/perf.html
- perf, ftrace, and more: http://www.brendangregg.com/linuxperf.html
- eBPF: http://lwn.net/Articles/603983/
- ktap: http://www.ktap.org/
- SystemTap: https://sourceware.org/systemtap/
- sysdig: http://www.sysdig.org/
- Kernelshark: http://people.redhat.com/srostedt/kernelshark/HTML/
- Trace Compass: https://projects.eclipse.org/projects/tools.tracecompass
- Flame graphs: http://www.brendangregg.com/flamegraphs.html
- Heat maps: http://www.brendangregg.com/heatmaps.html

Thanks

- Questions?
- http://slideshare.net/brendangregg
- http://www.brendangregg.com
- bgregg@netflix.com
- @brendangregg