Open Source Systems Performance

Brendan Gregg

Lead Performance Engineer
Joyent

Jul, 2013

A Play in Three Acts

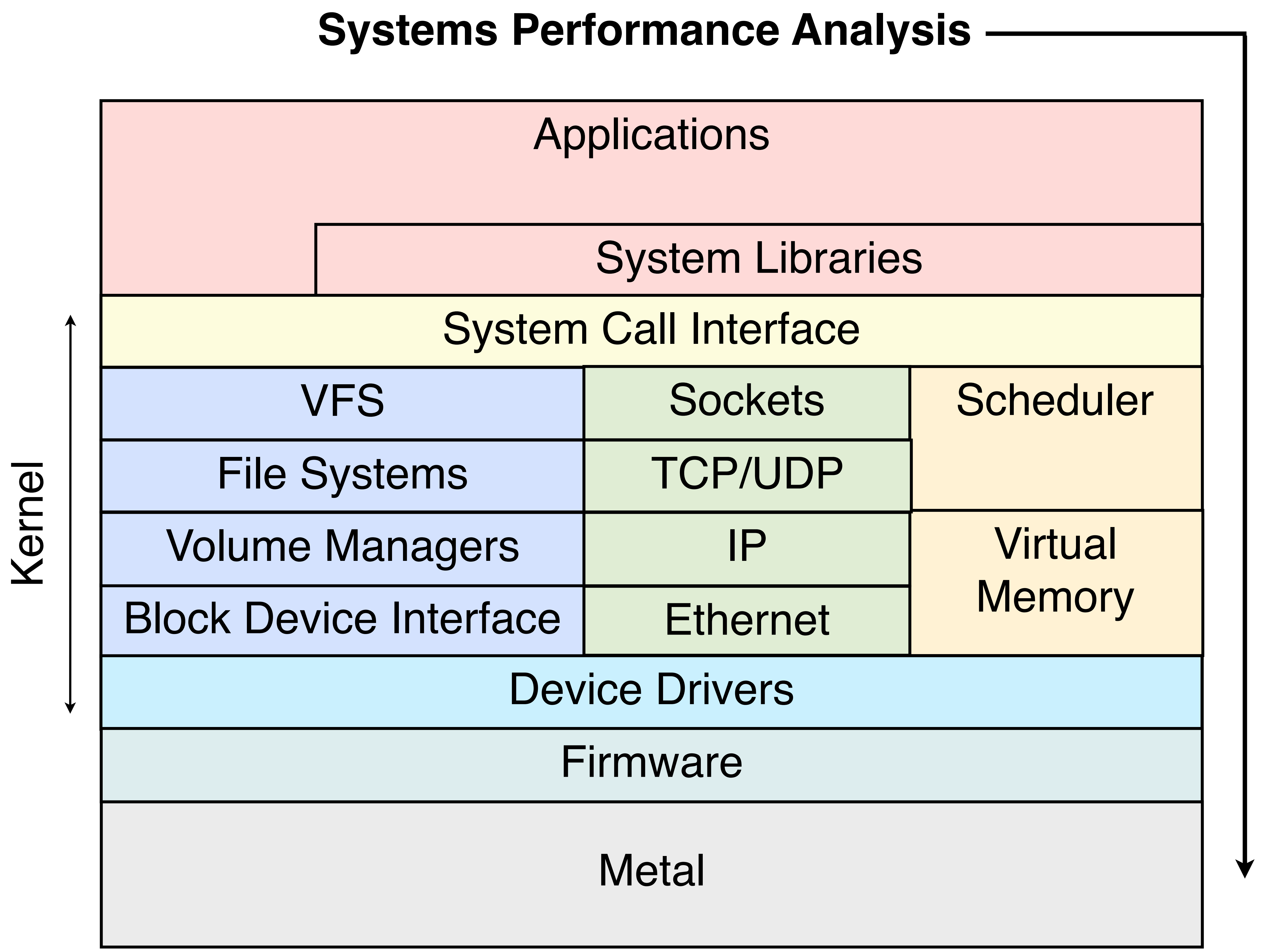
- A tale of operating systems, performance, and open source
- Dramatis Personae
  - Solaris, an Operating System
  - Brendan Gregg, a Performance Engineer
  - Linux, a Kernel
- Acts
  - 1. Before open source (traditional tools)
  - 2. Open source (source code-based tracing)
  - 3. Closed source

Setting the Scene: Why Performance?

- Reduce IT Spend
  - price/performance
- Choose performing components
  - evaluation (benchmarking) of software and hardware
- Develop scalable architectures
  - understand system limits and develop around them
- Solve issues

Setting the Scene: What is Systems Performance?

- Analysis of:
  - A) the kernel
    - 2-20% wins: tuning TCP, NUMA, etc
    - 2-200x wins: latency outliers, bugs, etc
  - B) applications from system context
    - 2-2000x wins: eliminating unnecessary work
- The *basis* is the system
- The *target* is everything, down to metal
- Think LAMP not AMP



Part 1. Before Open Source

Part 1. Before Open Source

- The year is 2002
- Enter Solaris 9, stage left
- Solaris 9 is not open source

Solaris 9

- Numerous performance observability tools

Scope	Type	Tools
system	counters	vmstat(1M), iostat(1M), netstat(1M), kstat(1M), sar(1)
system	tracing	snoop(1M), prex(1M), tnfdump(1)
process	counters	ps(1), prstat(1M), ptime(1)
process	tracing	truss(1), sotruss(1), apptrace(1)
both	profiling	lockstat(1M), cpustat(1M), cputrack(1)

- Performance, including resource controls and observability, were main features

Systems Performance

- Typified by Unix tools like vmstat(1M) (from BSD):

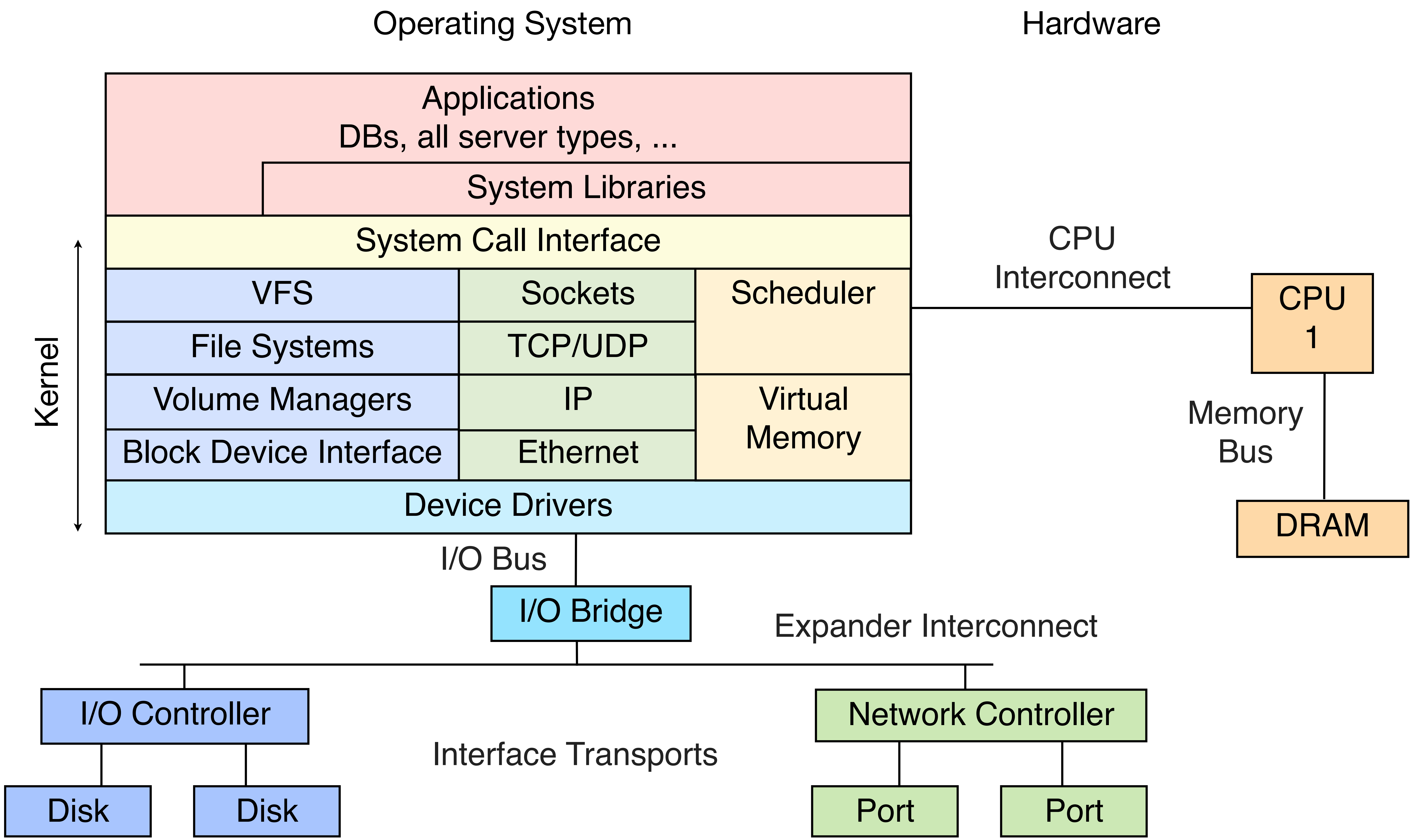
```
$ vmstat 1
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd cd s0 s5   in   sy   cs us sy id
 0 0 0 8475356 565176 2   8  0  0  0  0  1  0  0 -0 13  378  101  142  0  0 99
 1 0 0 7983772 119164 0   0  0  0  0  0  0 224 0  0  0 1175 5654 1196  1 15 84
 0 0 0 8046208 181600 0   0  0  0  0  0  0 322 0  0  0 1473 6931 1360  1  7 92
[...]
```

- Some drill-down were possible with options; eg, the Solaris -p:

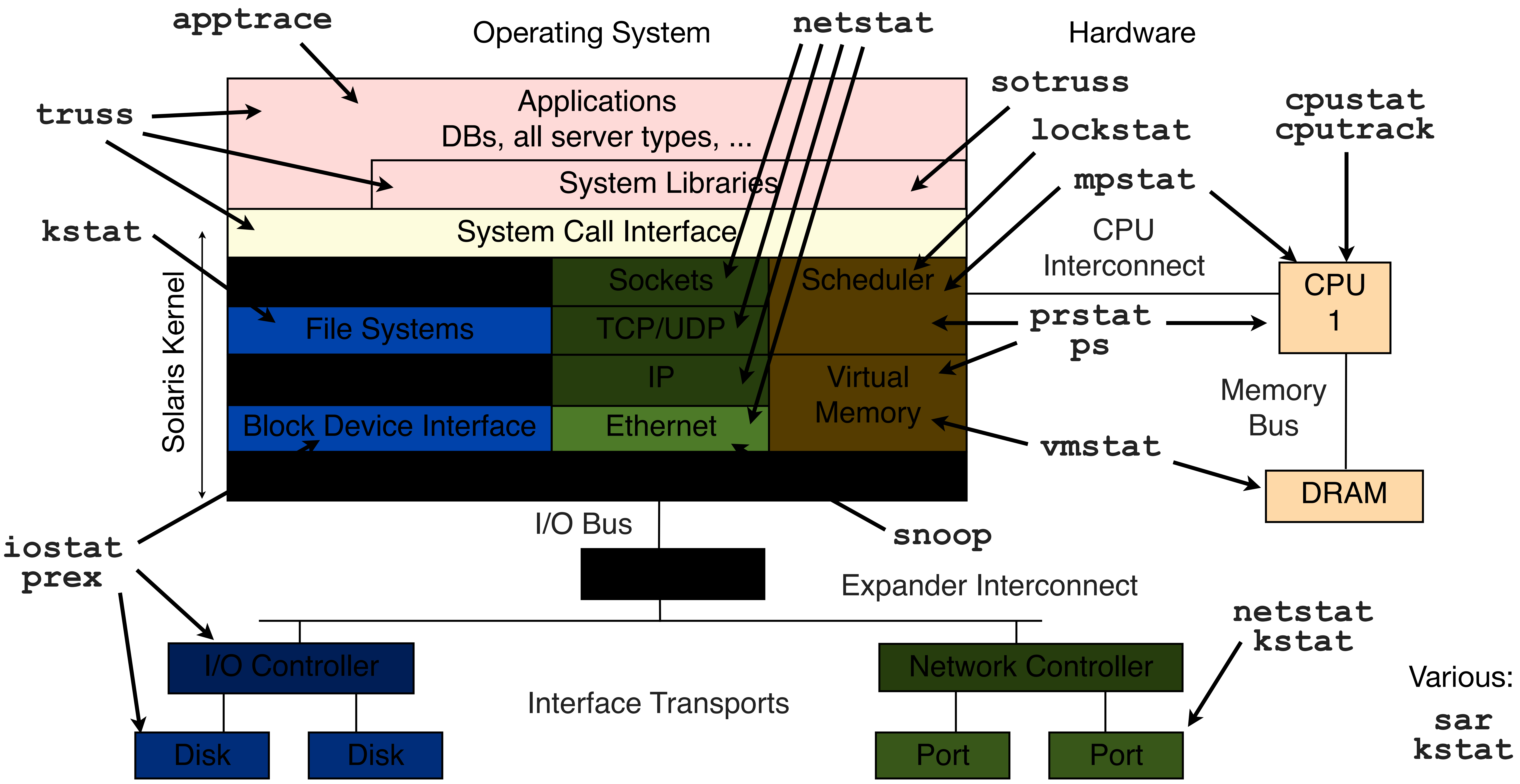
```
$ vmstat -p 1
     memory           page          executable      anonymous      filesystem
   swap  free  re  mf  fr  de  sr  epi  epo  epf  api  apo  apf  fpi  fpo  fpf
 8475336 565160 2   8   0   0   1    0    0    0    0    0    0    0    0    0
 7972332 107648 1  29   0   0   0    0    0    0    0    0    0    0    0    0
 7966188 101504 0   0   0   0   0    0    0    0    0    0    0    0    0    0
[...]
```

- Despite many tools, options, and metrics, the extent of observability was limited. This can be illustrated using a functional diagram

Operating System Functional Diagram



Solaris 9 Observability Coverage

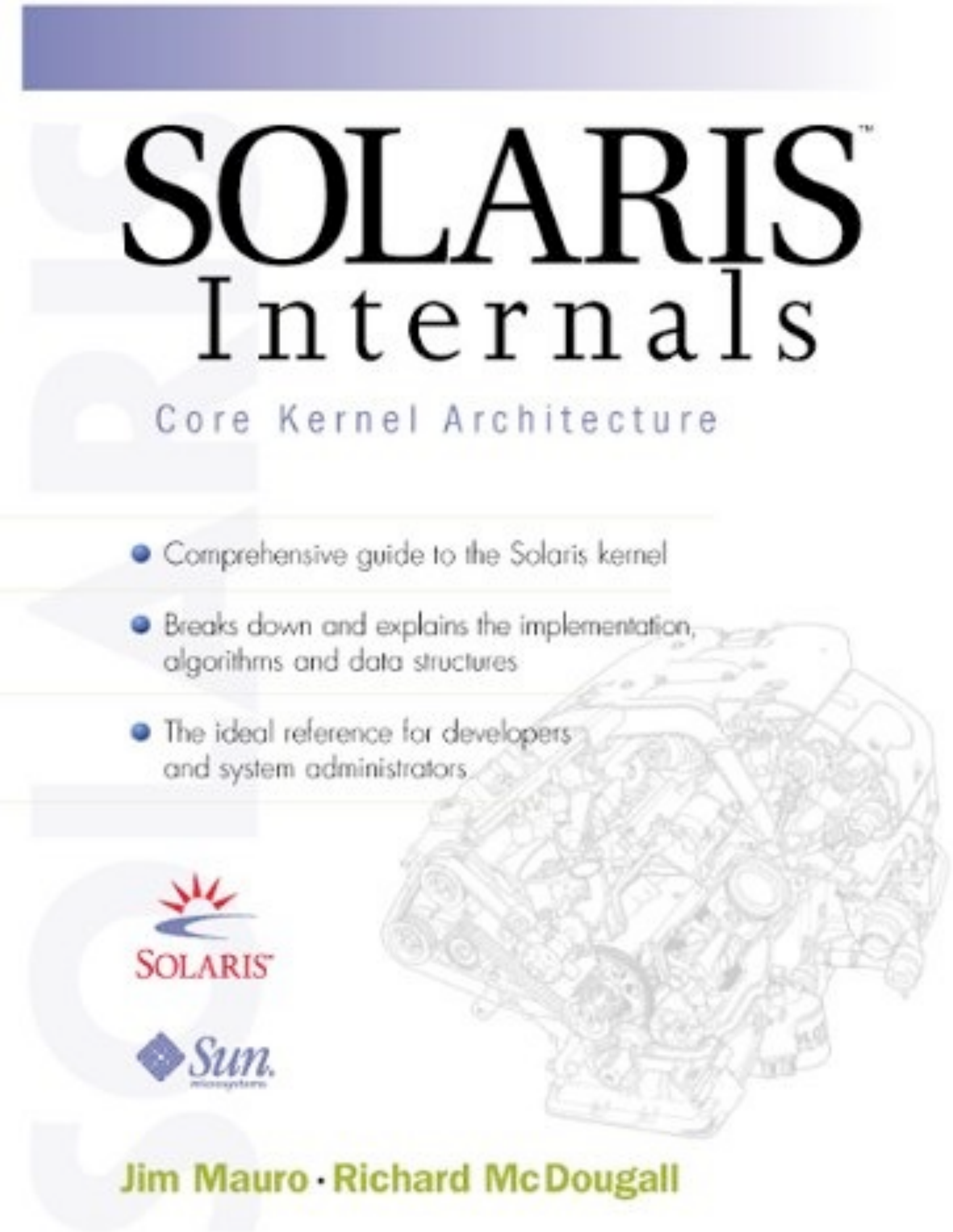


Problems

- Below the syscall interface was dark, if not pitch black
- Many components either had:
  - No metrics at all
  - Undocumented metrics (kstat)
- Certain performance issues could not be analyzed
  - Time from asking Sun for a new performance metric to having it in production could be months or years or never
  - You solve what the current tools let you: the “tools method” of iterating over existing tools and metrics
- Situation largely accepted as a better way wasn’t known
- Much systems performance literature was written in this era, and is still around

High Performance Tuning

- Performance experts were skilled in the art of inference and experimentation
  - Study Solaris Internals for background
  - Determine kernel behavior based on indirect metrics
  - Create known workloads to test undocumented metrics, and to explore system behavior
  - Heavy use of the Scientific method
- Science is good, source is better



... If the Universe was Open Source

```
vi universe/include/electron.h:

struct electron {
        mass_t e_mass;           /* electron mass */
        charge_t e_charge;       /* electron charge */
        uint64_t e_flags;        /* 0x01 particle; 0x10 wave */
        int e_orbit;             /* current orbit level */
        boolean_t e_matter;      /* 1 = matter; 0 = antimatter */
        [...]
} electron_t;

vi universe/particles.c:

photon_t *
spontaneous_emission(electron_t *e) {
        photon_t *p;
        if (e->e_orbit > 1) {
                p = palloc(e);
                e->e_orbit--;
        } else {
                electron_capture(e->e_nucleusp);
                return (NULL)
        }
        return (p);
}
```

Part 2. Open Source

Part 2. Open Source

- The year is 2005
- Solaris 10, as OpenSolaris, becomes open source
  - In response to Linux, which always was

Open Source Metrics

- Undocumented kstats could now be understood from source
  - it was like being handed the source code to the Universe
  - I wasn't a Sun badged employee; I'd been working without source access
- Tool metrics could also be better understood, and exact behavior of the kernel

```
$ vmstat 1
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd cd s0 s5   in   sy   cs us sy id
 0 0 0 8475356 565176 2   8  0  0  0  0  1  0  0 -0 13  378  101  142  0  0 99
 1 0 0 7983772 119164 0   0  0  0  0  0  0 224 0  0  0 1175 5654 1196  1 15 84
```

- For example, where does "r" come from?

Understanding "r"

- Starting with vmstat(1M)'s source, and drilling down:

```
usr/src/cmd/stat/vmstat/vmstat.c:

static void
printhdr(int sig)
{
[...]
        if (swflag)
                (void) printf(" r b w   swap  free  si  so pi po fr de sr ");
        else
                (void) printf(" r b w   swap  free  re  mf pi po fr de sr ");
[...]

static void
dovmstats(struct snapshot *old, struct snapshot *new)
{
[...]
        adjprintf(" %*lu", 1, DELTA(s_sys.ss_sysinfo.runque) / sys_updates);
```

Understanding “r”, cont.

- Searching on ss_sysinfo:

```
usr/src/cmd/stat/common/statcommon.h:

struct sys_snapshot {
        sysinfo_t ss_sysinfo;
[...]


usr/src/uts/common/sys/sysinfo.h:

typedef struct sysinfo {        /* (update freq) update action          */
        uint_t  updates;        /* (1 sec) ++                           */
        uint_t  runque;         /* (1 sec) += num runnable procs        */
        uint_t  runocc;         /* (1 sec) ++ if num runnable procs > 0 */
        uint_t  swpque;         /* (1 sec) += num swapped procs         */
        uint_t  swpocc;         /* (1 sec) ++ if num swapped procs > 0  */
        uint_t  waiting;        /* (1 sec) += jobs waiting for I/O      */
} sysinfo_t;
```

Understanding “r”, cont.

- ss_sysinfo is populated from kstat:

```
usr/src/cmd/stat/common/acquire.c:

int
acquire_sys(struct snapshot *ss, kstat_ctl_t *kc)
{
        size_t i;
        kstat_named_t *knp;
        kstat_t *ksp;

        if ((ksp = kstat_lookup(kc, "unix", 0, "sysinfo")) == NULL)
                return (errno);

        if (kstat_read(kc, ksp, &ss->s_sys.ss_sysinfo) == -1)
                return (errno);
[...]
```

Understanding "r", cont.

- Searching on runque population, in the kernel:

```
usr/src/uts/common/os/clock.c:

static void
clock(void)
{
         * There is additional processing which happens every time
         * the nanosecond counter rolls over which is described
         * below - see the section which begins with : if (one_sec)
[...]
        do {
                uint_t cpu_nrunnable = cp->cpu_disp->disp_nrunnable;
                nrunnable += cpu_nrunnable;
[...]
        } while ((cp = cp->cpu_next) != cpu_list);
[...]
        if (one_sec) {
[...]
                if (nrunnable) {
                        sysinfo.runque += nrunnable;
                        sysinfo.runocc++;
                }
```

Once-a-second snapshots?
That's good to know!

Statistic Spelunking

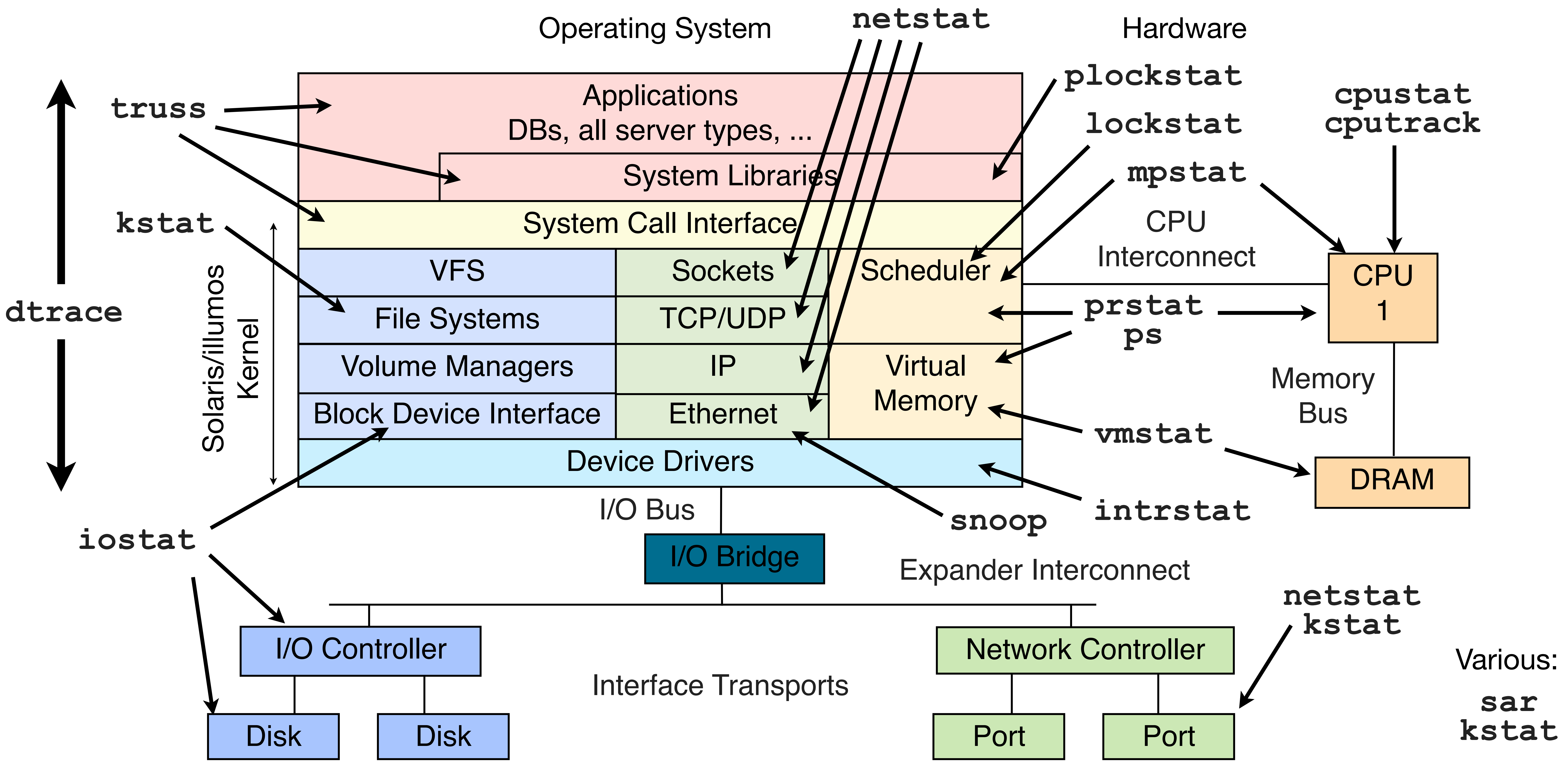
- A matter of browsing and reading source code
  - I use cscope, a text-based source code browser:
- Doesn't require expertise to begin with: keep reading code until it makes sense
- Might take hours or days if you are new to a complex code base
- You may only do this three times in your career, but each time was worth it!

```
C symbol: runque

  File       Function     Line
0 sa.h       <global>      188 uint64_t runque;
1 sysinfo.h  <global>      132 uint_t runque;
2 sar.c      prt_q_opt     919 (float )xx->si.runque / (float )xx->si.runocc,
3 kstat.c    save_sysinfo 1066 SAVE_UINT32(ksi, sysinfo, runque);
4 vmstat.c   dovmstats     316 adjprintf(" %*lu", 1,
                               DELTA(s_sys.ss_sysinfo.runque) / sys_updates);
5 clock.c    clock         862 sysinfo.runque += nrunnable;

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Open Source Dynamic Tracing

- Solaris 10 also provided Dynamic Tracing (DTrace), which can observe virtually everything
- Core feature of all later OpenSolaris derivatives, including SmartOS and OmniOS
- Observability gaps now filled

Solaris 10/SmartOS/OmniOS Observability Coverage



Open Source Dynamic Tracing: Example

- Given the kernel source code, eg, ZFS SPA sync:

```
usr/src/uts/common/fs/zfs/spa.c:

/*
 * Sync the specified transaction group.  New blocks may be dirtied as
 * part of the process, so we iterate until it converges.
 */
void
spa_sync(spa_t *spa, uint64_t txg)
{
        dsl_pool_t *dp = spa->spa_dsl_pool;
[...]
```

- Trace and time it using the DTrace function boundary tracing (fbt) provider:

```
# dtrace -n 'fbt::spa_sync:entry { self->ts = timestamp; } fbt::spa_sync:return /self->ts/
{ printf("%Y %d ms", walltimestamp, (timestamp - self->ts) / 1000000); self->ts = 0; }'
dtrace: description 'fbt::spa_sync:entry ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  53625                  spa_sync:return 2013 Jul 26 17:37:02 12 ms
  0  53625                  spa_sync:return 2013 Jul 26 17:37:08 726 ms
  6  53625                  spa_sync:return 2013 Jul 26 17:37:17 6913 ms
  6  53625                  spa_sync:return 2013 Jul 26 17:37:17 59 ms
```

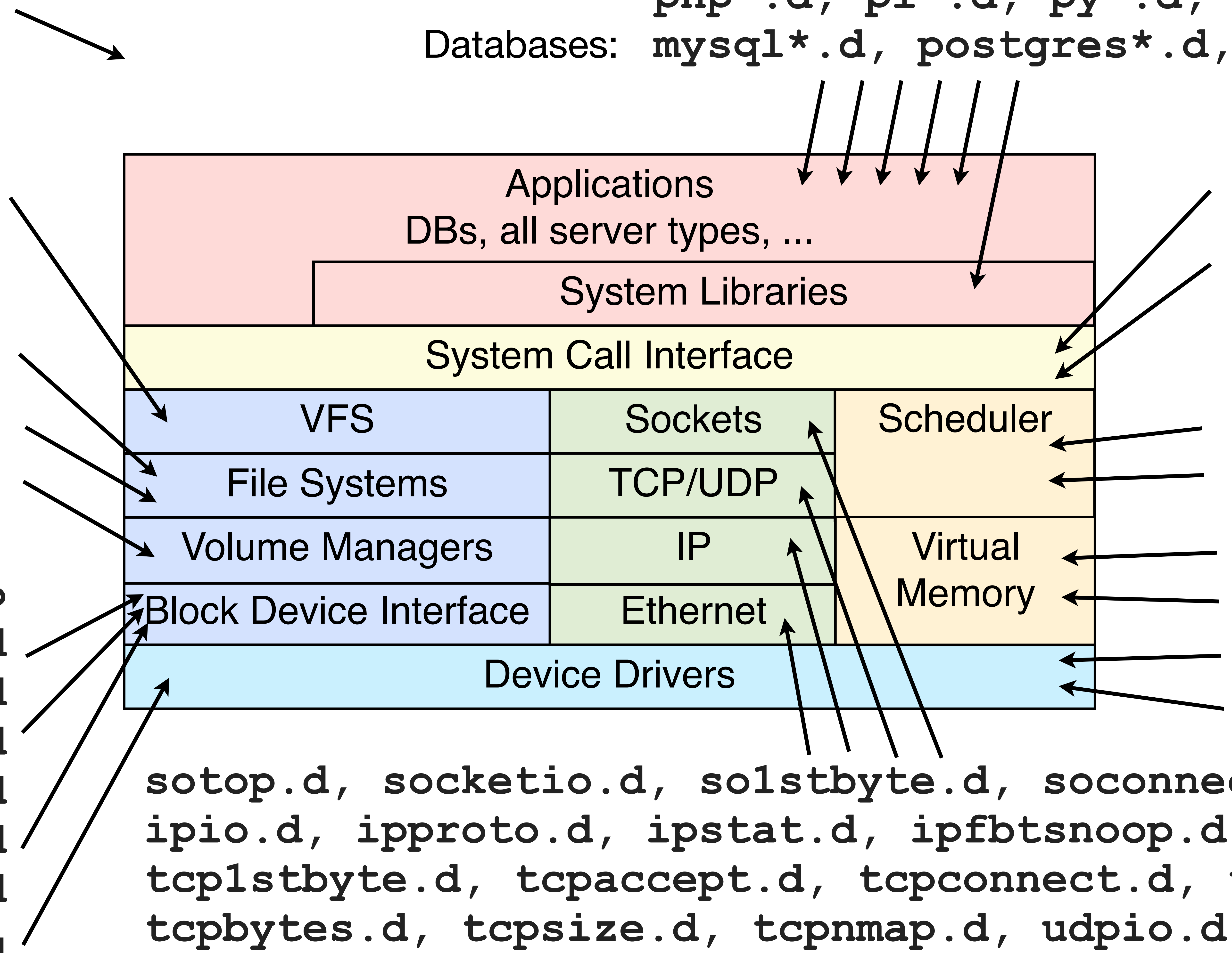
Awesome!

Dynamic Tracing Scripts

cifs*.d, iscsi*.d :Services
nfsv3*.d, nfsv4*.d
ssh*.d, httpd*.d

Language Providers: hotuser, umutexmax.d, lib*.d
node*.d, erlang*.d, j*.d, js*.d
php*.d, pl*.d, py*.d, rb*.d, sh*.d

Databases: mysql*.d, postgres*.d, redis*.d, riak*.d



fswho.d, fssnoop.d
sollife.d
solvfssnoop.d

dnlcsnoop.d
zfsslower.d
ziowait.d
ziostacks.d
spasync.d
metaslab_free.d

iosnoop, iotop
disklatency.d
satacmds.d
satalatency.d
scsicmds.d
scsilatency.d
sdretry.d, sdqueue.d
ide*.d, mpt*.d

opensnoop, statsnoop
errinfo, dtruss, rwtop
rwsnoop, mmap.d, kill.d
shellsnoop, zonecalls.d
weblatency.d, fddist

priclass.d, pridist.d
cv_wakeup_slow.d
displat.d, capslat.d

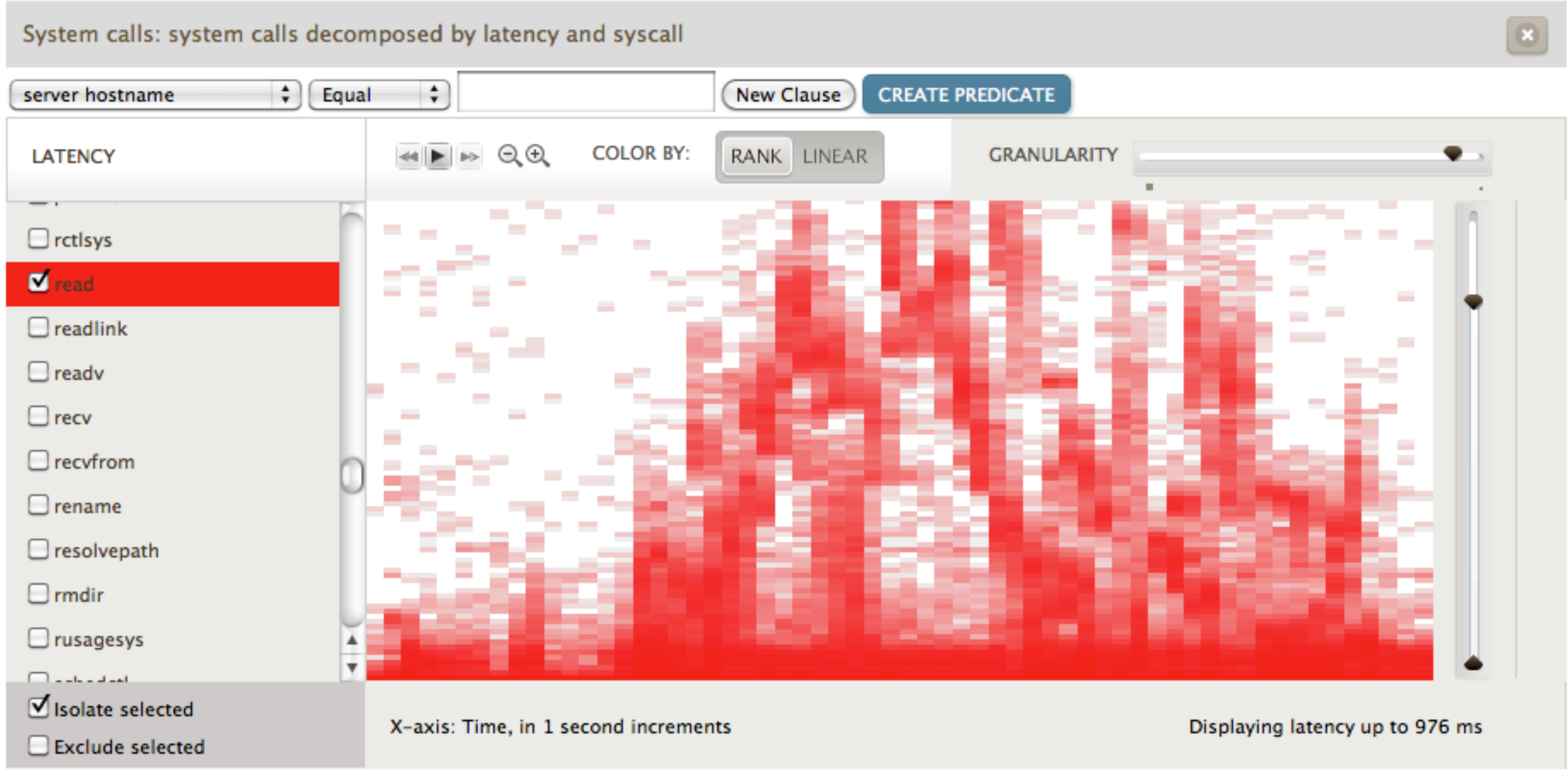
minfbypid.d
pgpginbypid.d

macops.d
ngesnoop.d, ngelink.d

sotop.d, socketio.d, so1stbyte.d, soconnect.d, soaccept.d
ipio.d, ipproto.d, ipstat.d, ipfbtsnoop.d, icmpsnoop.d
tcp1stbyte.d, tcpaccept.d, tcpconnect.d, tcpconnlat.d, tcpio.d
tcpbytes.d, tcpsize.d, tcpnmap.d, udpio.d, udpstat.d

Modern Systems Performance

- Typified by an abundance of high resolution useful metrics (latency)
- No longer a problem of missing metrics, but how to visualize many metrics, and across clouds
  - eg, latency heat maps:



- Prior tools are useful as starting points, with tracing to dig deeper
- In the following sections, I'll describe modern Linux Systems Performance, summarizing how traditional and new tools can be used together
  - I'll group dtrace/systemtap/perf/lttng/ktap/etc as "dynamic tracing", which is a simplification: some needs may not be met by all those tools

Linux CPU Analysis

- Traditional tools:
  - 1. system wide usage `vmstat`
  - 2. per-processor usage `mpstat`
  - 3. per-process usage `top, ps`
  - 4. user- or kernel-stack profiling `perf record -agF`
  - 5. cycle analysis `perf stat`
- Modern tools:
  - 6. tracing scheduler latency `perf sched`
  - 7. tracing CPU usage of functions dynamic/static tracing
  - 8. tracing CPU consumption of spin locks dynamic/static tracing
  - 9. CPU cross call tracing dynamic/static tracing
  - 10. interrupt tracing dynamic/static tracing

Linux Memory Analysis

- Traditional tools:
  - 1. system wide usage — `vmstat`
  - 2. per-process usage — `top`
  - 3. kernel usage — `/proc/meminfo, slaptop`
  - 4. swapping activity — `sar`
  - 5. leak detection — valgrind
- Modern tools:
  - 6. tracing allocations — dynamic tracing
  - 7. tracing page faults — dynamic/static tracing
  - 8. tracing kswapd activity — dynamic/static tracing

Linux File System Analysis

- Traditional tools:
  - 1. cache usage `free, /proc/meminfo`
  - 2. syscall access `strace` (expensive)
- Modern tools:
  - 3. tracing VFS accesses dynamic/static tracing
  - 4. tracing file system latency `latencytop`, dynamic/static tracing
  - 5. tracing file system internals dynamic/static tracing

Linux Disk Analysis

- Traditional tools:
  - 1. per-disk statistics `iostat`
  - 2. per-process usage `pidstat -d, iotop`
- Modern tools:
  - 3. disk I/O latency tracing `blktrace`, static tracing
  - 4. lower I/O stack tracing dynamic/static tracing
  - 5. SCSI command tracing dynamic/static tracing
  - 6. device driver tracing dynamic/static tracing

Linux Network Analysis

- Traditional tools:
  - 1. system wide usage — `netstat -s`
  - 2. per-interface usage — `netstat -i, sar -n DEV, ip`
  - 3. TCP statistics — `netstat -s, sar -n TCP`
  - 4. packet tracing — `tcpdump`
  - 5. socket call tracing — `strace` (expensive)
  - 6. experimental tests — `ping, traceroute`
- Modern tools:
  - 6. tracing socket-level latency — dynamic/static tracing
  - 7. TCP retransmit (only) tracing — dynamic tracing
  - 8. tracing TCP kernel internals — dynamic tracing

Linux Network Analysis, Example

- TCP retransmits: given tcp_retransmit_skb(), show the dest IP addr. Source code:

```
net/ipv4/tcp_output.c:

int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int err = __tcp_retransmit_skb(sk, skb);
[...]

include/linux/tcp.h:

struct tcp_sock {
        /* inet_connection_sock has to be the first member of tcp_sock */
        struct inet_connection_sock     inet_conn;
[...]

include/net/inet_connection_sock.h:

struct inet_connection_sock {
        /* inet_sock has to be the first member! */
        struct inet_sock          icsk_inet;
[...]
```

Linux Network Analysis, Example

- ... More spelunking, like earlier. Not trivial, but doable.

```
include/net/inet_sock.h:

struct inet_sock {
        /* sk and pinet6 has to be the first two members of inet_sock */
        struct sock             sk;
#if IS_ENABLED(CONFIG_IPV6)
        struct ipv6_pinfo       *pinet6;
#endif
        /* Socket demultiplex comparisons on incoming packets. */
#define inet_daddr              sk.__sk_common.skc_daddr
[...]

include/net/sock.h

struct sock {
        /*
         * Now struct inet_timewait_sock also uses sock_common, so please just
         * don't add nothing before this first member (__sk_common) --acme
         */
        struct sock_common      __sk_common;
[...]
```

Here it is

Linux Network Analysis, Example Script

- TCP retransmit tracing script, using DTrace4Linux (prototype):

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

#pragma D option quiet

dtrace:::BEGIN { trace("Tracing TCP retransmits... Ctrl-C to end.\n"); }

fbt::tcp_retransmit_skb:entry {
    this->so = (struct sock *)arg0;
    this->d = (unsigned char *)&this->so->__sk_common.skc_daddr;
    printf("%Y: retransmit to %d.%d.%d.%d, by:", walltimestamp,
        this->d[0], this->d[1], this->d[2], this->d[3]);
    stack(99);
}
```

Linux Network Analysis, Example Output

- TCP retransmit tracing script, using DTrace4Linux (prototype):

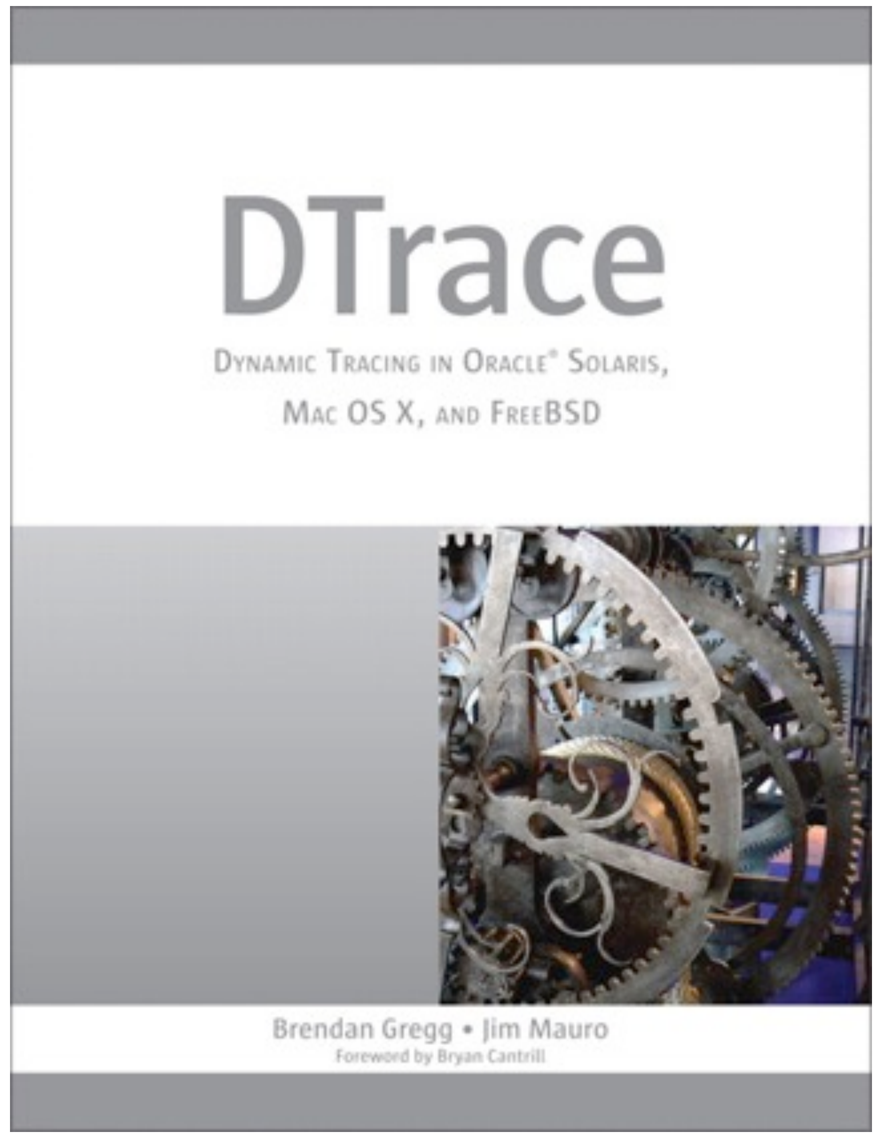
```
# ./tcpretransmit.d
Tracing TCP retransmits... Ctrl-C to end.
2013 Feb 23 18:24:11: retransmit to 10.2.124.2, by:
              kernel`tcp_retransmit_timer+0x1bd
              kernel`tcp_write_timer+0x188
              kernel`run_timer_softirq+0x12b
              kernel`tcp_write_timer
              kernel`__do_softirq+0xb8
              kernel`read_tsc+0x9
              kernel`sched_clock+0x9
              kernel`sched_clock_local+0x25
              kernel`call_softirq+0x1c
              kernel`do_softirq+0x65
              kernel`irq_exit+0x9e
              kernel`smp_apic_timer_interrupt+0x6e
              kernel`apic_timer_interrupt+0x6e
[...]
```

Linux Network Example, cont.

- I created a custom performance tool on the fly, without kernel changes
- Would it be possible if the kernel wasn't open source?

Opportunities

- Open source allows dynamic tracing: otherwise you are tracing blind
- Dynamic tracing allows custom metrics and scripts (tools) to be written
  - fill in all observability gaps; can solve most performance issues
- Many people will use dynamic tracing: eg, DTraceToolkit, DTrace book, company tools; only some may author the tools: the OS or perf engineer on your team (which is ok)
- Dynamic tracing also allows new methodologies
  - prior methodologies constrained by existing tools and metrics
  - new methodologies can be explored, as any question posed can be answered
- Examples of new methodologies
  - USE method
  - Thread State Analysis method



Challenges

- Systems performance literature written for the pre-open source Unix days
  - Gives the impression that performance ends with older static tools
- DTrace not on Linux yet
  - Two ports are in progress:
    - DTrace4Linux: https://github.com/dtrace4linux/linux
    - Oracle Linux DTrace
- Instead of waiting, you can try an illumos-kernel based distro like SmartOS
  - illumos is the surviving fork of OpenSolaris. Which brings us to Act 3.

Act 3. Closed Source

Act 3. Closed Source

- The year is 2010
- Oracle stops releasing updates for OpenSolaris
- Oracle Solaris 11 is released a year later, closed source
- Provides us with a unique additional perspective for open source systems performance

Closed Source Metrics

- This closed the only documentation for many metrics and kernel internals
  - Back to inference and experimentation by the end user
  - Will get harder over time as documentation ages: without a Solaris Internals 3rd Edition, kernel internals may become as opaque as it was in the 90's

Closed Source Dynamic Tracing

- Makes using the DTrace fbt provider *much* harder
  - Hypothetical example to show how this could manifest:
    - Dynamic tracing of ZFS SPA sync during a performance investigation:

```
# dtrace -n 'fbt::spa_sync:entry { printf("%Y", walltimestamp); }'
dtrace: invalid probe specifier fbt::spa_sync:entry { printf("%Y", walltimestamp); }:
probe description fbt::spa_sync:entry does not match any probes
```

      Where'd spa_sync() go? Did it get renamed or removed?

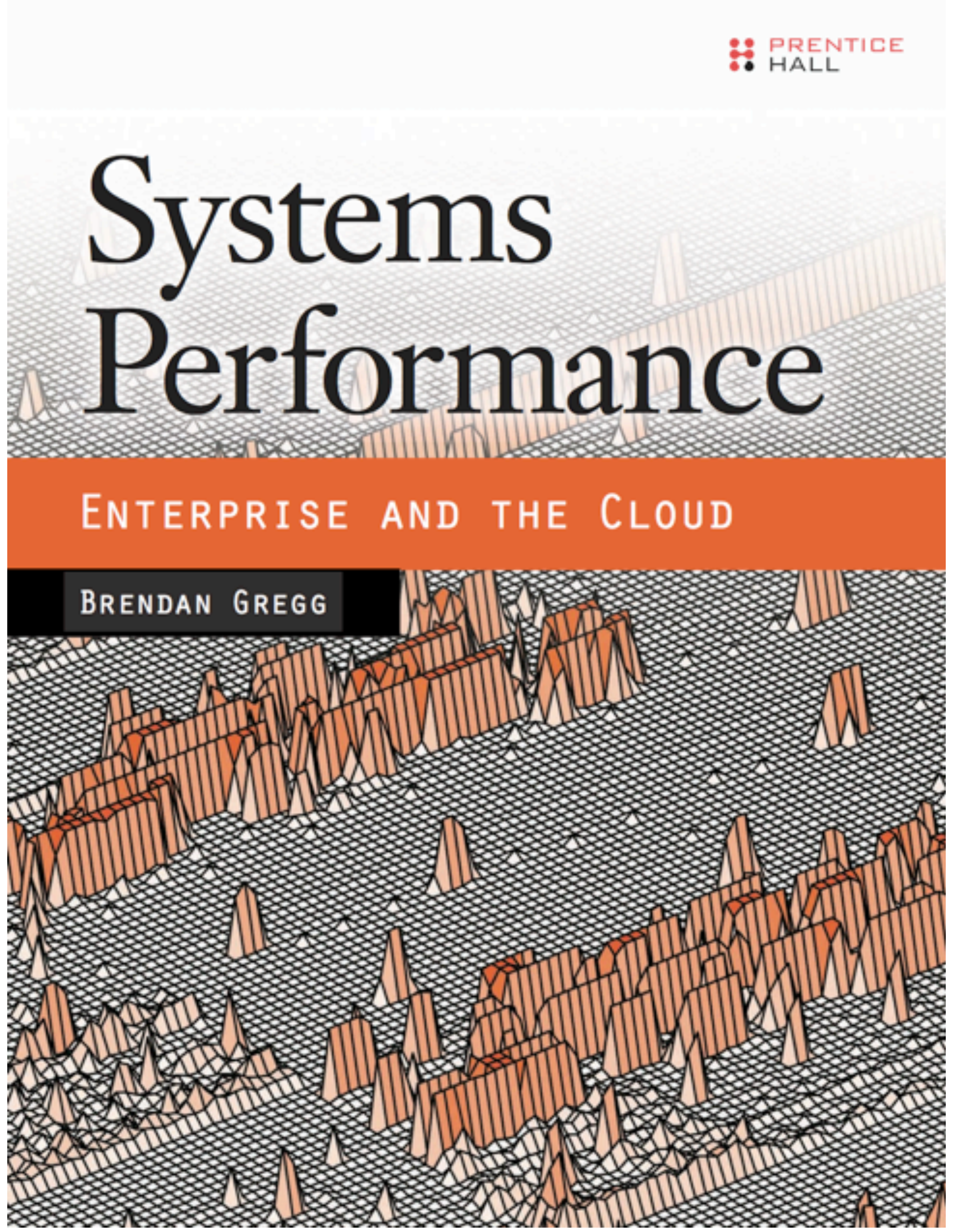
    - Could be worse if tracing succeeds, but produces misleading metrics due to unknown changes
- Note that the capabilities are still there, and can be used by Oracle support

Elsewhere at Oracle

- Their DTrace port for Oracle Linux won't have this handicap
  - although, the fbt provider hasn't been included yet

Epilog: The New Systems Performance

- An era of:
  - Open source
  - Dynamic tracing
  - Methodologies
  - Distributed systems (cloud)
- Covered in my book, out this year:



Thank you!

- email: brendan@joyent.com
- twitter: @brendangregg
- blog: http://dtrace.org/blogs/brendan