Advanced debugging with eBPF and Linux tools

Background

New set of Linux "performance" tools

I don't see our community using them much

They can save hours! Days!

Not really an "HTCondor" talk

But real reason for this talk is …

What do experts say?

"It depends"

But I'm not an expert here, so "I don't know"

A word on "Advanced"

Go Install these packages! NOW!

```
$ sudo yum install perf
$ sudo yum install bpftrace bcc

$ sudo apt-get install linux-perf
$ sudo apt-get install bpfcc-
tools
```

Motivating example

Job takes 20 minutes to run on researcher's laptop is taking 20+ hours to run on our "fast" cluster HTC computers –

WHY?

(also: running right now)

Initial investigation on EP

```
$ condor_ssh_to_job 17012325
Welcome to slot1_2@e2550.chtc.wisc.edu!
Your condor job is running with pid(s) 3437472.
$ uptime
 11:11:56 up 127 days  load average: 183.74, 181.60, 181.89
[gthain@e2550 ~]$ grep -c ^processor /proc/cpuinfo
256
$ ps auxww | grep 3437472
gthain   3437472 4364  1328 ? Ds   11:03   7:33 science_job
```

Essence of Debugging: Binary Search

What I know right now:

What I want to know:

perf trace command

```
$ sudo /bin/bash
Password:
# perf trace -p 3437472 -duration 10
```

-p <pid to trace>

Only show syscalls whose duration is at least 10 milliseconds

(why 10?)

"Duration" of a syscall

Duration is real time

Some long durations are (probably) OK:

    e.g., sleep

But sleep is not a syscall – "nanosleep" is

Also, the sleep-like calls:

    select, epoll, **futex**

perf trace command

```
# perf trace -p 3437472 -duration 10
95.705 (63.135 ms): futex(val: 895) = 0
95.411 (63.417 ms): futex(val: 895) = 0
95.694 (63.155 ms): futex(val: 895) = 0
95.714 (63.126 ms): futex(val: 895) = 0
95.741 (63.098 ms): futex(val: 895) = 0
…
```

perf trace command

```
# perf trace -p 3437472 -duration 10 -e '!futex'
```

But only show

NOT (!)
Futex calls

perf trace command

```
# perf trace -p 3437472 -duration 10 -e '!futex'
5.3 (821.412 ms): read(fd: 3</staging/big_file>, …) = 8192
7.5 (738.578 ms): read(fd: 3</staging/big_file>, …) = 8192
7.7 (819.972 ms): rea                              >, …) = 8192
8.6 (828.883 ms):                                  >, …) = 8192
```

WHOAH!
Way too big!

/Staging is on ceph…

Solution: call ceph admin

Call ceph admin, inform fs system

Ceph admin understands problem, fixes it

5 minute later, job starting running fast

perf trace command -- after

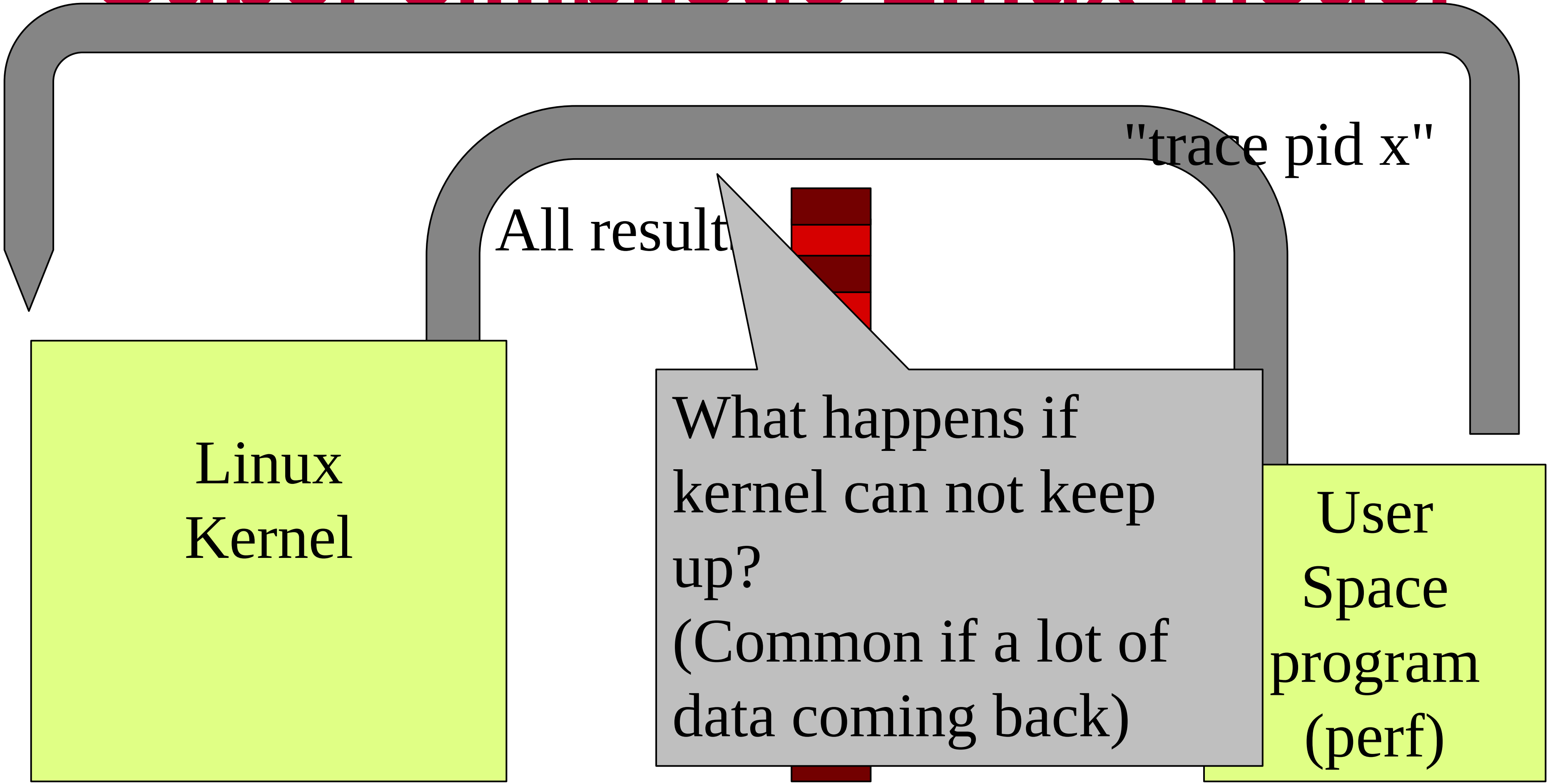
```
# perf trace -p 3437472 -duration 10 -e '!futex'
5.3 (21.412 ms): read(fd: 3</staging/big_file>, …) = 8192
7.5 (17.578 ms): read(fd: 3</staging/big_file>, …) = 8192
7.7 (89.972 ms): read(fd: 3</staging/big_file>, …) = 8192
8.6 (28.883 ms): read(fd: 3</staging/big_file>, …) = 8192
```

And the job finished in roughly 20 minutes!

Why not grep?

```
# perf trace -p 3437472 |
        grep -v 'futex'
```

Super simplistic Linux model



Two choices

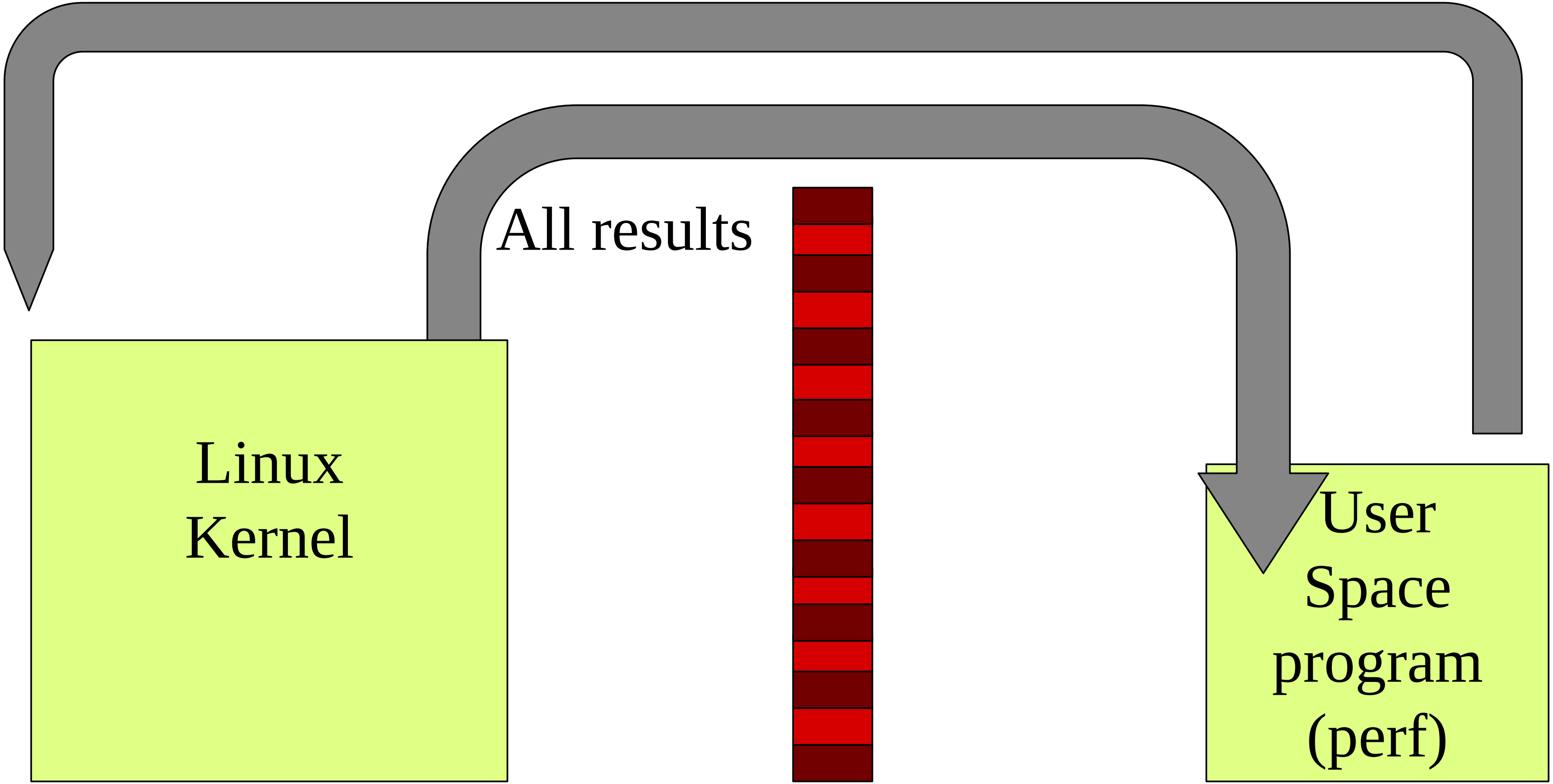
Drop

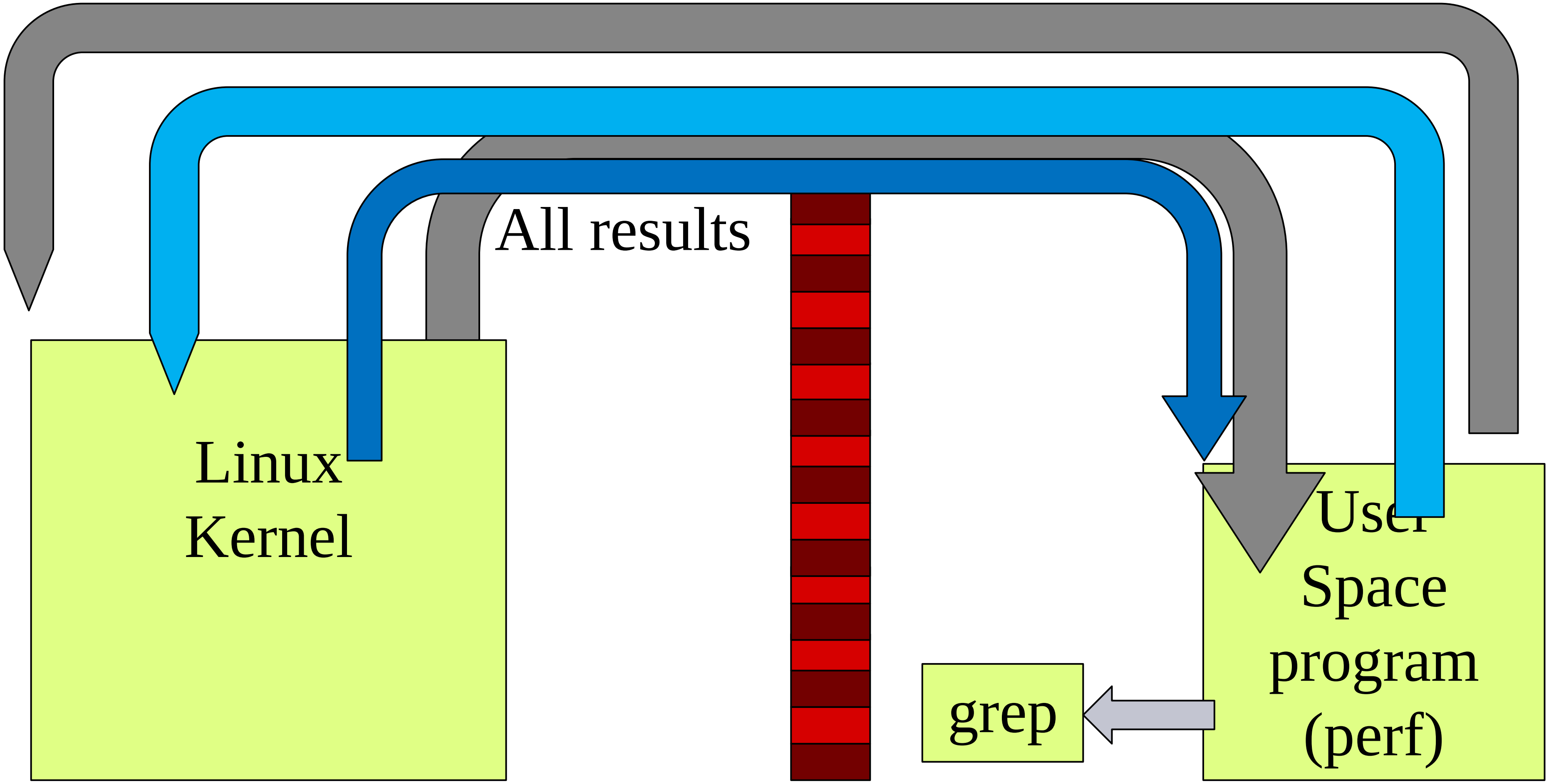
- Don't send info, just drop on floor

Block

- Slow down traced process

```
# perf trace
 1056.747 ( 0.478 ms): ... [continued]: read()) = 8192
LOST 47 events!
  1056.747 ( 0.568 ms): ... [continued]: read()) = 8192
LOST 45 events!
  1056.747 ( 0.654 ms): ... [continued]: read()) = 8192
LOST 39 events!
  1056.747 ( 0.741 ms): ... [continued]: read()) = 8192
LOST 53 events!
  1056.747 ( 0.853 ms):... [continued]: read()) = 8192
LOST 8 events!
LOST 38 events!
```


All results
Linux Kernel
User Space program (perf)


All results
Linux
Kernel
User
Space
program
(perf)
grep

Why not strace?

strace blocks, not drops

Performance isn't **biggest** problem but is one

Impact on traced processes is.

Strace uses ptrace(2), slow, clunky, generic

A word on "performance"

Use case:
perf trace --summary

We added

htcondor eventlog read

For sanity check, checked memory and cpu performance (with time) – MUCH SLOWER

condor_userlog simple, compute

1.) Reads event log file

2.) Deserializes

3.) Prints out one line per event

```
$ condor_userlog /var/log/condor/GlobalEventLog | head

Job       Host            Start Time  Evict Time  Wall Time Good
Time CPU Usage
269288.7 172.22.60.140     9/10 00:36  9/10 00:36    0+00:00
0+00:00    0+00:00
269288.7 172.22.60.138     9/10 00:36  9/10 00:36    0+00:00
0+00:00    0+00:00
269288.7 172.22.60.61      9/10 00:36  9/10 00:36    0+00:00
```

htcondor eventlog read VS condor_userlog

```
$ time condor_userlog
/var/log/condor/GlobalEventLog > /dev/null

real    0m36.707s
user    0m17.462s
sys     0m19.243s
```

Does this look odd?

Any ideas?

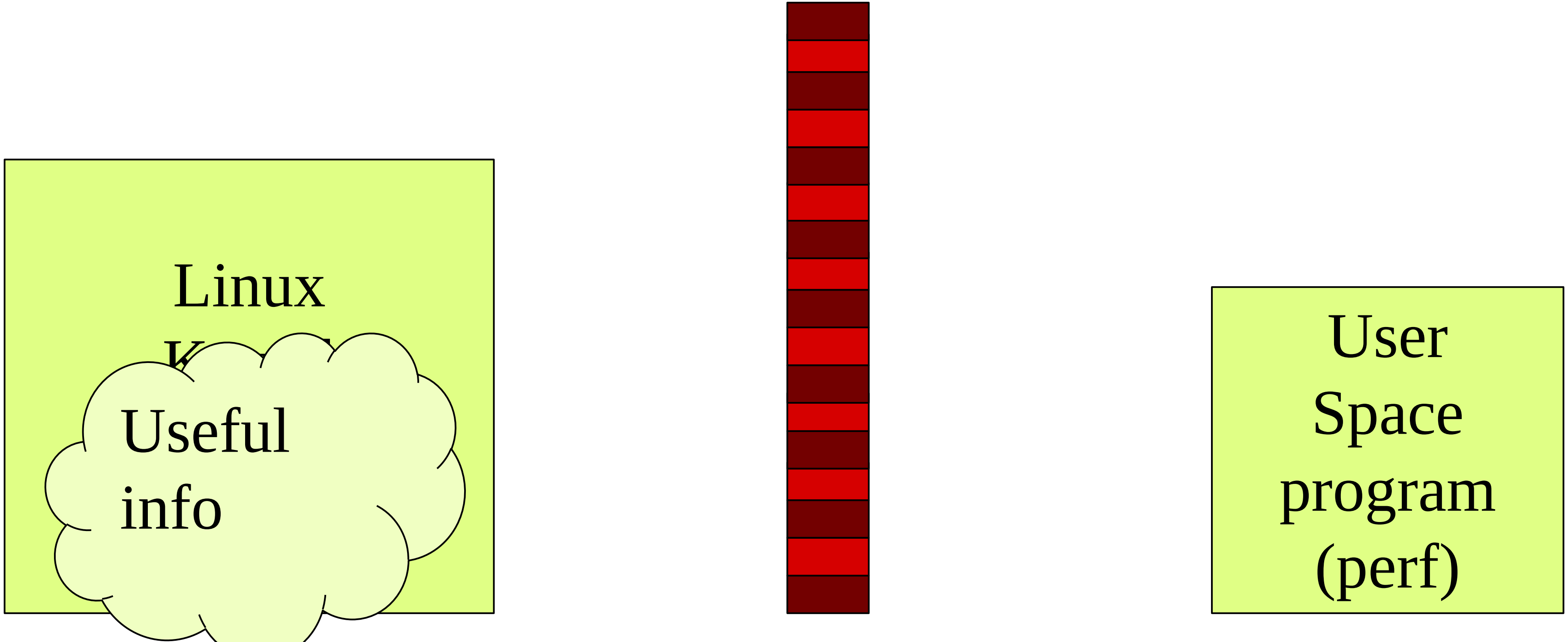
```
# perf trace --summary condor_userlog GlobalEventLog> /dev/null
 Summary of events:
 condor_userlog (1917553), 56611085 events,
100.0%
   syscall            calls   errors
   ------ --------- ---------- -----
   stat            11821024        0
   fstat           10828288        0
   gettimeofday     5413529        0
   read              221032        0
   write              19018        0
   brk                 1017        0
```

```
# perf trace -e stat --call-graph dwarf condor_userlog
1990.826 ( 0.263 ms): condor_userlog/2262058
stat(filename: 0xea3ef543, statbuf: 0x7ffd30495fd0)
= 0
_xstat (inlined)
__tzfile_read (/usr/lib64/libc-2.28.so)
tzset_internal (/usr/lib64/libc-2.28.so)
__tzset (/usr/lib64/libc-2.28.so)
__GI_timelocal (inlined)
ULogEvent::readHeader
ULogEvent::getEvent (/usr/lib64/libcondor_utils_10_7_0.so)
ReadUserLog::readEventNormal
ReadUserLog::rawReadEvent
ReadUserLog::readEventWithLock
```

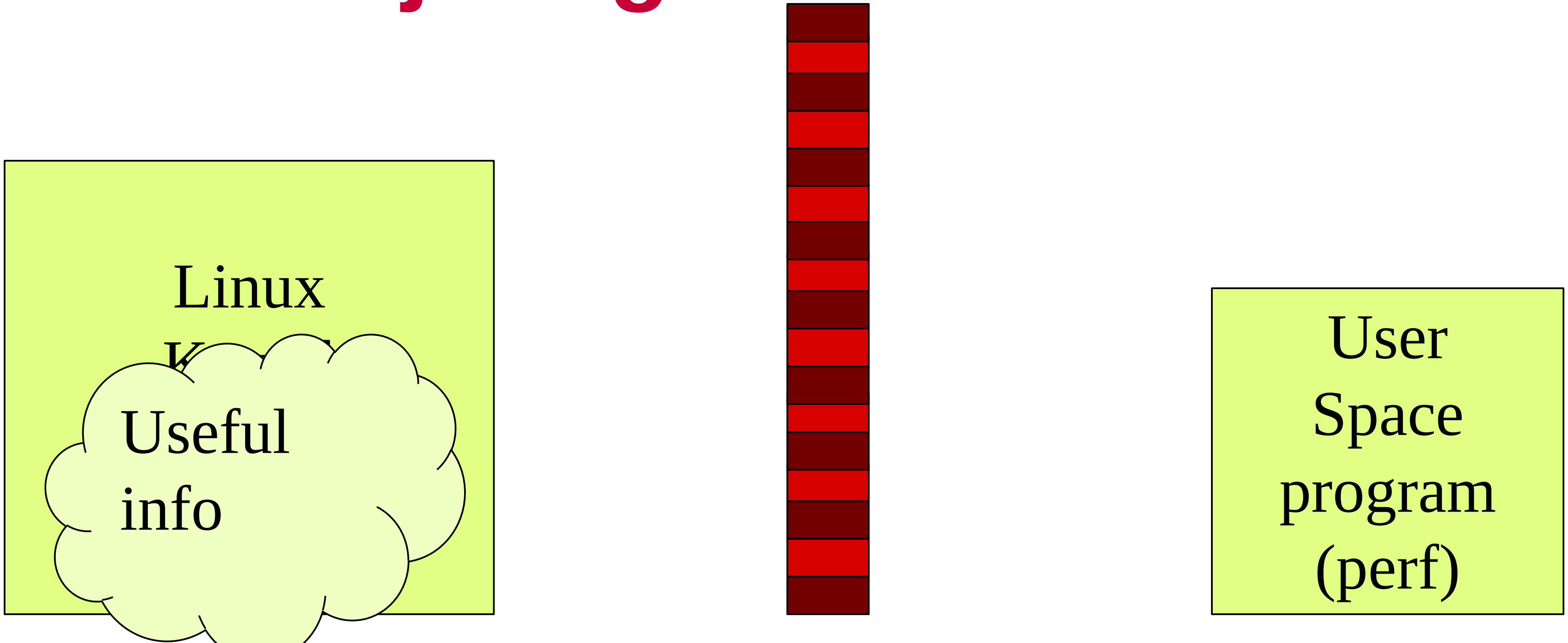
```
# time condor_userlog GlobalEventLog > /dev/null

real    0m35.446s
user    0m17.486s
sys     0m17.960s

# time TZ=GMT condor_userlog GlobalEventLog >
/dev/null
real    0m28.592s
user    0m18.112s
sys     0m10.480s
```

Without sending all info over wall for userspace to reduce

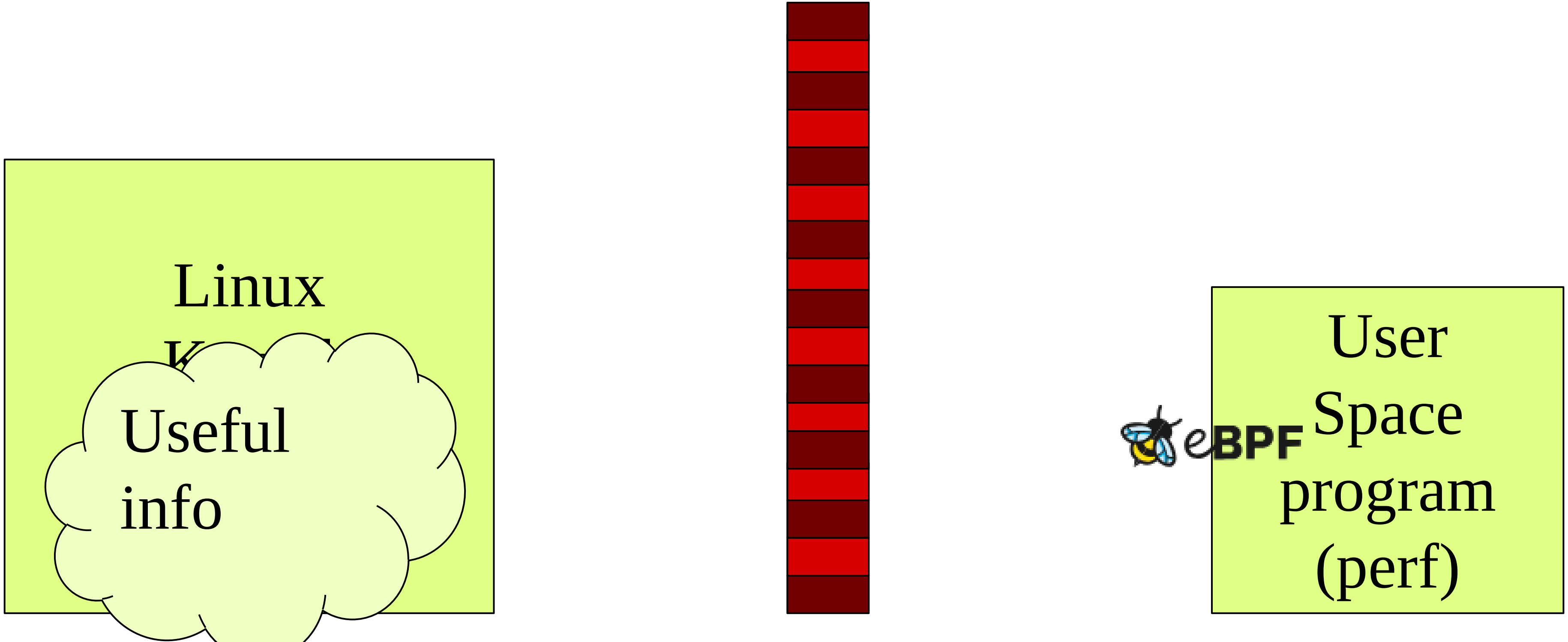
How to get at all info in kernel?



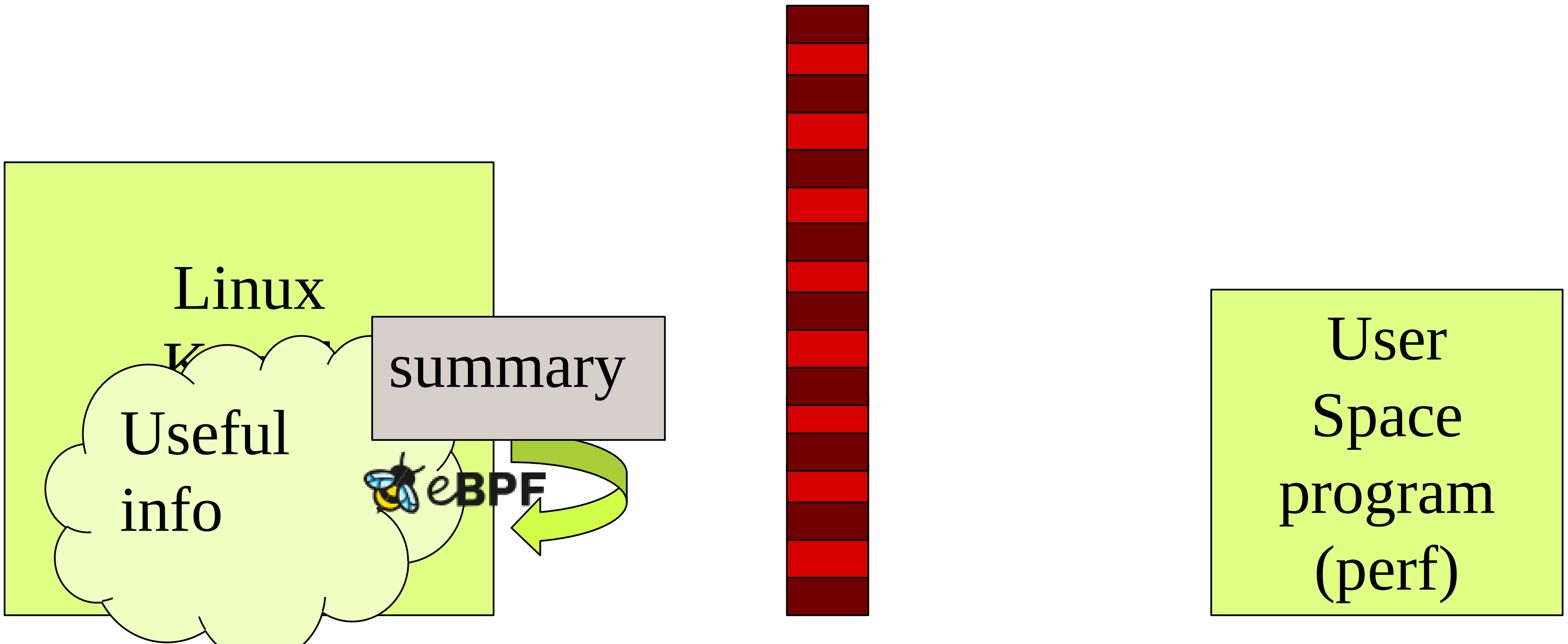
Summary is tiny percentage.How do we just get it from kernel?



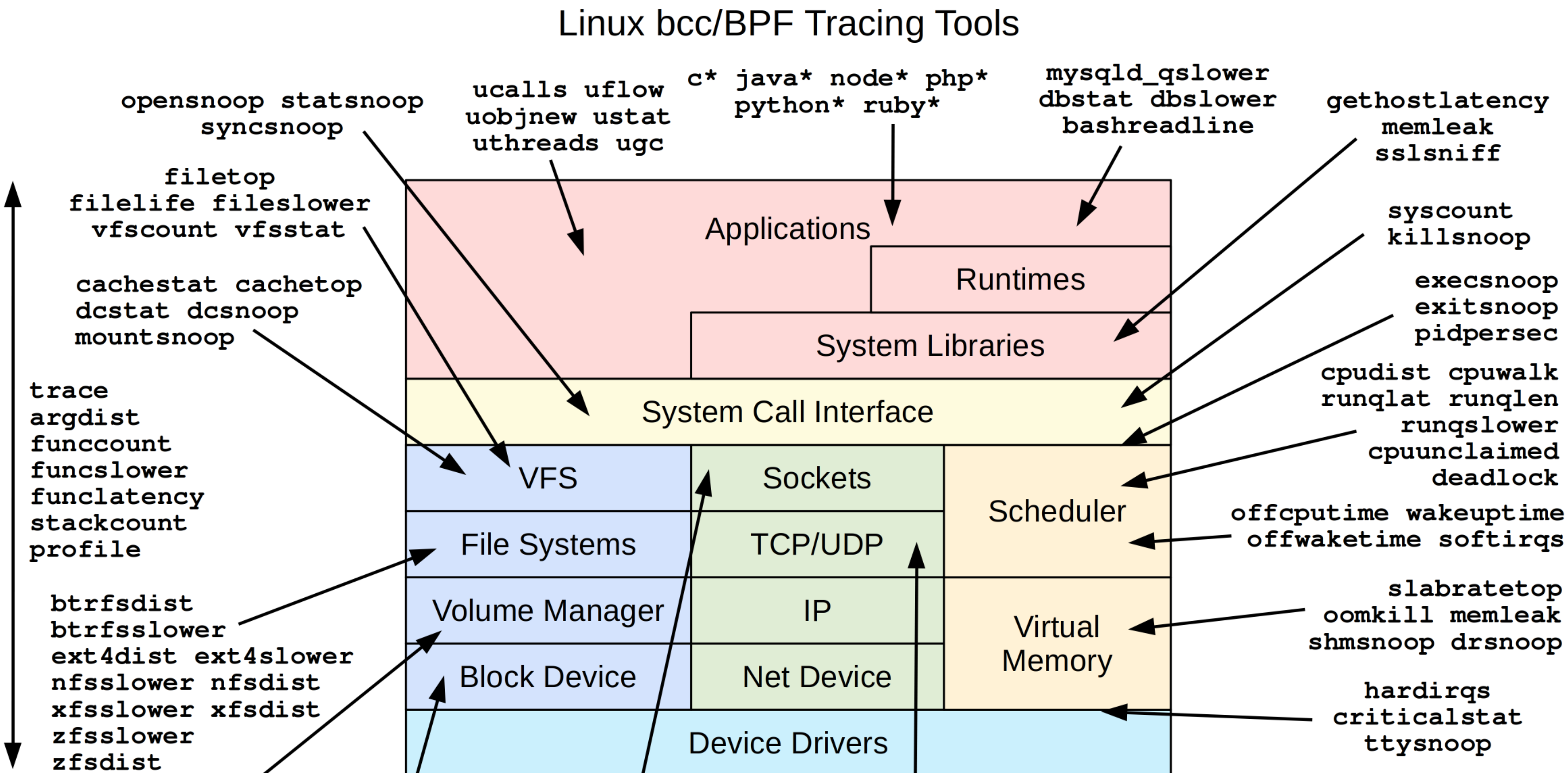
eBPF: send code to data



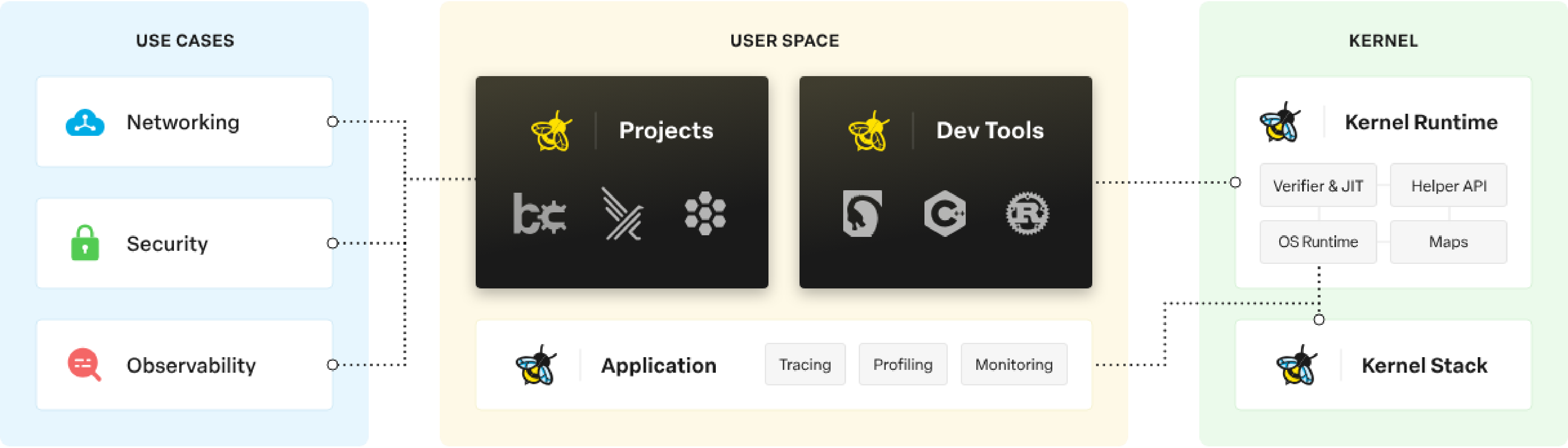
eBPF: send code to data



eBPF: big


Linux bcc/BPF Tracing Tools
opensnoop statsnoop
syncsnoop
ucalls uflow
uobjnew ustat
uthreads ugc
c* java* node* php*
python* ruby*
mysqld_qslower
dbstat dbslower
bashreadline
gethostlatency
memleak
sslsniff
filetop
filelife fileslower
vfscount vfsstat
cachestat cachetop
dcstat dcsnoop
mountsnoop
Applications
Runtimes
System Libraries
System Call Interface
syscount
killsnoop
execsnoop
exitsnoop
pidpersec
cpudist cpuwalk
runqlat runqlen
runqslower
cpuunclaimed
deadlock
trace
argdist
funccount
funcslower
funclatency
stackcount
profile
VFS
Sockets
Scheduler
File Systems
TCP/UDP
offcputime wakeuptime
offwaketime softirqs
Volume Manager
IP
Virtual
Memory
slabratetop
oomkill memleak
shmsnoop drsnoop
btrfsdist
btrfsslower
ext4dist ext4slower
nfsslower nfsdist
xfsslower xfsdist
zfsslower
zfsdist
Block Device
Net Device
hardirqs
criticalstat
ttysnoop
Device Drivers

eBPF: enormous




DON'T PANIC

(yet)

Historical Aside

eBPF (2014)

inspired by

SOLARIS DTrace

(2005)

inspired by

Paradyn

(1994)

Greg's (surely wrong) eBPF summary

A constrained 16 register assembly language

non-Turing complete – remind you of ???

With compilers for C, Python, other source

And built-in aggregating data structures

With a (jit) implementation in the kernel

Can be triggered on any kernel probe

Allow some (re) programing of the kernel

Two reactions:

OMG – This can't be good

1. Security?
2. Stability of kernel?
3. Complexity

Cool

1. Ultimate Power
2. I can think of 8 uses…
3. When can I start?

Start with pre-built tools…

Command	Action
execsnoop	Traces all exec pcall
opensnoop	Traces all file open
biolatency	Display block (disk) latency
biosnoop	Traces block access
tcpconnect	Traces all tcp connect
tcpaccept	Traces all incoming tcp
gethostlatency	Traces all DNS lookups

e.g. execsnoop

DaemonCoreDutyCyle

is > 0.95 on AP – Why?

(you DO watch DCDC?)

Aside:

HTCSS good at noticing, not so good at isolating

```
# execsnoop

TIME(s) PCOMM              PID     PPID    RET ARGS
0.602   condor_shadow      2421138 2903754   0 /usr/sbin/condor_shadow 7.15
0.940   sh                 2421145 2421144   0   condor_q
0.944   condor_q           2421145 2421144   0 /usr/bin/condor_q
1.828   sh                 2421159 2421158   0   condor_q
1.830   condor_q           2421159 2421158   0 /usr/bin/condor_q
2.268   sh                 2421165 2421164   0   condor_q
2.271   condor_q           2421165 2421164   0 /usr/bin/condor_q
2.618   sh                 2421170 2421169   0   condor_q
2.621   condor_q           2421170 2421169   0 /usr/bin/condor_q
3.023   sh                 2421178 2421177   0   condor_q
3.026   condor_q           2421178 2421177   0 /usr/bin/condor_q
```

$ watch –n 0.1 condor_q

Please don't do this. Can kill an AP

use "condor_watch_q" instead

"-n 0.1" means 10 Hz

We killed the "watch", AP returned to normal.

```
# execsnoop

TIME(s) PCOMM            PID     PP
0.602   condor_shadow    2421138 2                              .15
0.940   sh               2421145 24
0.944   condor_q         2421145 24
1.828   sh               2421159 24
1.830   condor_q         2421159 24
2.268   sh               2421165 24
2.271   condor_q
2.618   sh
2.621   condor_q         24211
3.023   sh               2421178 24
3.026   condor_q         2421178 24
```

Notice the time diffs here?

Not 10 Hz, at all

mean sched

Aside: why so common?

google "watch condor_q"…

Job Submission - Batch Docs

batchdocs.web.cern.ch/tutorial/exercise1a.html

CHTC Slack | HTCondor Design... | Jira | HTCondorWiki | Coverity Scan - Proj... | Condor Staff Sched... | Control Group APIs... | ops-summit-htcon...

Job Submission

Search | Source code

Monitoring the job

The command **condor_q** can be used curr

```
-- Schedd: bigbird04.cern.ch : <128.14
OWNER    BATCH_NAME        SUBMITTED
fprotops CMD: welcome.sh  12/6  15:08

1 jobs; 0 completed, 0 removed, 1 idle,
```

Who?

The condor_q command provides information regarding the current state of the jobs, the name of the schedd, the name of the owner, etc.

The progress of a job can be followed by executing:

```
watch condor_q
```

The -nobatch option can be used to the status of each individual job rather the cluster summary.

```
condor_q -nobatch

 -- Schedd: bigbird04.cern.ch : <128.142.194.115:9618?... @
 ID       OWNER          SUBMITTED     RUN_TIME ST PRI SIZE
21847.0   fprotops       3/28 17:13   0+00:00:00 I  0   0.0
```

What ?

You can also get status on a specific job cluster:

```
$ condor_q -nobatch 1144.0 -- Schedd: training.osgconnect.net : <192.170.227.119:9419?... ID OWNER SUBMITTED RUN_
```

Note the ST (state) column. Your job will be in the `I` state (idle) if it hasn't started yet. If it's currently scheduled and running, it will have state `R` (running). If it has completed already, it will not appear in `condor_q`.

Let's wait for your job to finish – that is, for `condor_q` not to show the job in its output. A useful tool for this is watch – it runs a program repeatedly, letting you see how the output differs at fixed time intervals. Let's submit the job again, and watch condor_q output at two-second intervals:

```
$ condor_submit tutorial01.submit Submitting job(s). 1 job(s) submitted to cluste 1145 $ watch -n2 condor_q usern
```

When your job has completed, it will disappear from the list. To close w... hold down Ctrl and press C.

sigh

... about its execution from the `condor_history` command:

```
_TIME ST COMPLETED CMD 1144.0 username 3/6 09:46 0+00:00:12 C 3/6 09:4
```

Quickstart-Submit Example HTCo ×

portal.path-cc.io/documentation/htc_workloads/submitting_workloads/quickstart/

CHTC Slack · HTCondor Design... · JIRA · HTCondorWiki · Coverity Scan - Proj... · Condor Staff Sched... · Control Group APIs... · ops-summit-htcon...

PATh **Quickstart-Submit Example HTCondor Jobs** Search

Managing HTC Workloads On the PATh Facility

- Submitting HTC Workloads With HTCondor
  - Quickstart-Submit Example HTCondor Jobs
  - Easily Submit Multiple Jobs
  - Checkpointing Jobs
- Specific Resource Needs
- Software
- Containers
- Using Data and Job Files
- Automated Workflows

Note the `DONE`, `RUN`, and `IDLE` columns. Your job will be listed in the `IDLE` column if it hasn't started yet. If it's currently scheduled and running, it will appear in the `RUN` column. As it finishes up, it will then show in the `DONE` column. Once the job completes completely, it will not appear in `condor_q`.

Let's wa your job h – that is, for `con q` not to show the job in its output. A useful tool for is watch – it a program repeatedly g you see how the output differs at fixed time int Let's su he job again, and wat

```
$ condor_submit tutorial01.submit
Submitting job(s).
1 job(s) submitted to cluster 1441272
$ condor_watch_q
...
```

When your job has completed, it will disappear fr

Note: To close watch, hold down Ctrl and press C

Job history

Once your job has finished, you can get informati `condor_history` command:

Table of contents

ents to

TCondor's

Why is "watch condor_q" so bad?

And how can we get some insight?

```
# perf trace -p pid_of_schedd

1268.509 ( 0.002 ms): getpid() 2903754
1268.520 ( 0.011 ms): write(fd: 5<SchedLog>) =79
1268.534 ( 0.002 ms): rt_sigprocmask() = 0
1268.540 (147.908 ms): clone(flags: VFORK) = 301
1416.507 ( 0.008 ms)  close(fd: 55) = 0
```

Condor_q forks schedd (clone)
Here speed-of-light is ~ 8Hz

Why is clone/fork slow?

Roughly linear in memory size of schedd

(what happened to CoW? – page tables)

Why is schedd big?

```
condor_q -all -tot

-- Schedd: submit-1.chtc.wisc.edu : <1.2.3.4:5>

40713 jobs; 0 completed, 0 removed, 19281 idle,
3321 running, 18111 held, 0 suspended
```

Held jobs aren't free

Maybe don't keep them forever

Back to eBPF

bpftrace – easy mode to eBPF

Using bcc, even python is hard – why?

bpftrace is a much easier to use language

Modelled on AWK (!)

Bpftrace programs have…

Begin with block of kernel #include files…
BEGIN/END tag with block of source code

probe tag with block of source code

some magic globals blocks can use

Global maps/HashTables, printed on exit

Kind of like AWK!

Aside: What's a probe?

Place to attach code

Many different kinds, more being added…

For now, three probe types:

kprobe:func	**On entry to kernel function named func**
kretprobe:func	On any return from kernel function named func
tracepoint::syscall:open	On entry to syscall open, even if name changes

```
#!/usr/bin/bpftrace
#include <net/sock.h>

BEGIN {printf("Tracing network traffic.");}

kretprobe:sock_recvmsg
{
    @recv_bytes[pid, comm] = sum(retval);
}
```

```
Attaching 2 probes...
Tracing network traffic.

@recv_bytes[1614012, condor_shadow]: 38
@recv_bytes[1135048, condor_shadow]: 38
@recv_bytes[1499055, condor_shadow]: 38
@recv_bytes[2023650, condor_shadow]: 38
@recv_bytes[861103, condor_shadow]: 593
@recv_bytes[2336929, condor_shadow]: 596
@recv_bytes[2263702, condor_shadow]: 599
@recv_bytes[2263433, condor_shadow]: 599
@recv_bytes[1459336, condor_shadow]: 606
@recv_bytes[1065538, condor_shadow]: 607
@recv_bytes[1808916, condor_shadow]: 610
```

What's the best thing about ~~AWK~~ bpftrace?

One Liners!

A sampler platter of them

Stolen from:

https://github.com/iovisor/bpftrace/blob/master/doc

print file, proc for all opens

```
# bpftrace -e \
'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'
snmp-pass /proc/cpuinfo
snmp-pass /proc/stat
snmpd /proc/net/dev
snmpd /proc/net/if_inet6
```

syscall counts by process

```
# bpftrace -e 'tracepoint:raw_syscalls:sys_enter {
@[comm] = count(); }'
Attaching 1 probe...
^C

@[bpftrace]: 6
@[systemd]: 24
@[snmp-pass]: 96
@[sshd]: 125
```

syscall counts by process

```
# bpftrace -e 'tracepoint:syscalls:sys_exit_read /pid ==
18644/ { @bytes = hist(args.ret); }'

@bytes:
[0, 1]          12 |@@@@@@@@                                |
[2, 4)          18 |@@@@@@@@@@@@@@@@@@@                     |
[4, 8)           0 |                                        |
[8, 16)          0 |                                        |
[16, 32)         0 |                                        |
[32, 64)        30 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64, 128)       19 |@@@@@@@@@@@@@@@@@@@@                    |
```

Histogram of bytes read

```
# bpftrace -e 'tracepoint:syscalls:sys_exit_read
   /pid == 18644/ { @bytes = hist(args.ret); }'
@bytes:
[0, 1]        12 |@@@@@@@@                                |
[2, 4)        18 |@@@@@@@@@@@@@@@@@@@                     |
[4, 8)         0 |                                        |
[8, 16)        0 |                                        |
[16, 32)       0 |                                        |
[32, 64)      30 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64, 128)     19 |@@@@@@@@@@@@@@@@@@@@                    |
```

Histogram of bytes read

```
# bpftrace -e 'tracepoint:syscalls:sys_exit_read
   /pid == 18644/ { @bytes = hist(args.ret); }'
@bytes:
[0, 1]        12 |@@@@@@@@                                |
[2, 4)        18 |@@@@@@@@@@@@@@@@@@@                     |
[4, 8)         0 |                                        |
[8, 16)        0 |                                        |
[16, 32)       0 |                                        |
[32, 64)      30 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[64, 128)     19 |@@@@@@@@@@@@@@@@@@@@                    |
```

Final use case -- IGWN

IGWN had network overload, but hard time tracking down to single job

Pretty sure it was file xfer (or maybe sched?)

HTCondor keeps stats in history file

But only after xfer completes – too late

Great Programers copy

bpftrace ships with "tcpsnoop"

- Almost does what I wanted
- But per user, not per process

```
#!/bin/bpftrace
#include <net/sock.h>
#include <linux/cred.h>
#include <linux/sched.h>
#include <linux/uidgid.h>

BEGIN
{
  printf("Per User shadow network usage. Ctrl-C to
stop\n");
  clear(@recv_bytes);
  clear(@send_bytes);
}
```

```
kprobe:sock_recvmsg,
kprobe:sock_sendmsg
{
  $sock = (struct socket *)arg0;
  $family = $sock->sk->__sk_common.skc_family;
  /* Set a flag to ignore non-IP (unix domain sockets) */
  if ($family == AF_INET || $family == AF_INET6) {
    @inetsocket[tid] = 1;
  } else {
    @inetsocket[tid] = 0;
  }
}
```

```
kretprobe:sock_recvmsg
{
  if (( (comm == "condor_schedd") || (comm ==
"condor_shadow")) && (@inetsocket[tid] && retval
< 4294967000)) {
        $ct   = (struct task_struct *)curtask;
        $cred = (struct cred *)$ct->cred;
        $euid = $cred->euid.val;
        @recv_bytes[$euid, comm] = sum(retval);
  }
  delete(@inetsocket[tid])
}
```

```
kretprobe:sock_sendmsg
{
  if ((comm == "condor_schedd") || (comm ==
"condor_shadow")) &&
   (@inetsocket[tid] && retval < 4294960000)) {
        $ct   = (struct task_struct *)curtask;
        $cred = (struct cred *)$ct->cred;
        $euid = $cred->euid.val;
        @send_bytes[$euid, comm] = sum(retval);
  }
  delete(@inetsocket[tid])
}
```

```
@recv_bytes[1000, condor_schedd]: 1297
@send_bytes[1000, condor_schedd]: 296
@send_bytes[24755, condor_shadow]: 799
@send_bytes[21454, condor_shadow]: 799
@send_bytes[21046, condor_shadow]: 1566
@send_bytes[23265, condor_shadow]: 3026
@send_bytes[20589, condor_shadow]: 15856
@send_bytes[21506, condor_shadow]: 6954623
@send_bytes[23201, condor_shadow]: 12239630
```

eBPF futures: mutation

Originally read-only

Some limited mutation

- Replacing k8s networking sidecars
- Device limiting (see tomorrow)

Future ???

eBPF: Ultimate POSIX intervention?

Should HTCondor have 1st class bpf?

If so, who controls? Submitter? Admin?

Usually need root/CAP_BPF – worthwhile

What tracing info wanted from jobs?

all file opens? User selects from menu?

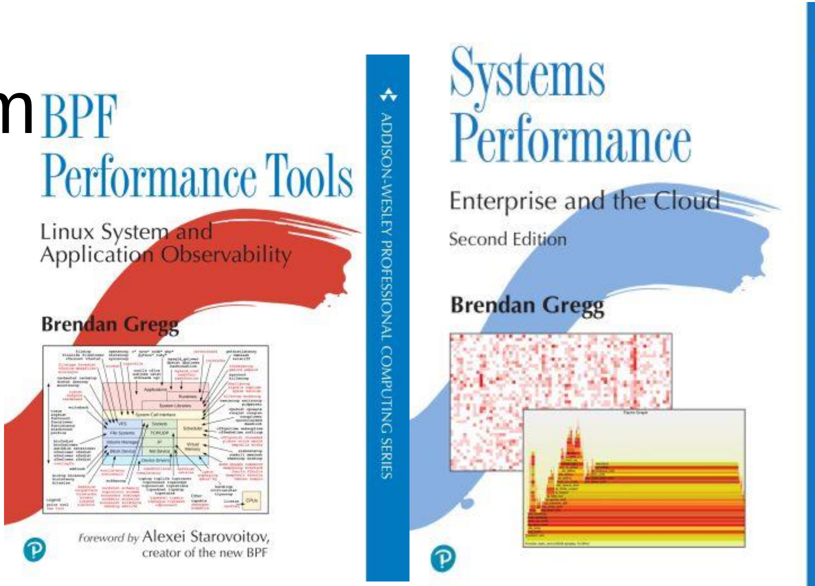
References

bpftrace

https://github.com/iovisor/bpftrace/blob/master/doc

Perf testing in general

https://www.brendangregg.com



Conclusion

This was not a HTCSS talk – is that ok?

eBPF/perf tools are powerful and under used

Bpftrace is an easy entry

This is just the beginning…