eBPF super powers on ARM64 and Android

Powerful Linux Tracing for Android

Joel Fernandes <joelaf@google.com>
LinuxInternals.org
IRC: joel_ on OFTC
Twitter: joel_linux

Google

About me:

Kernel team

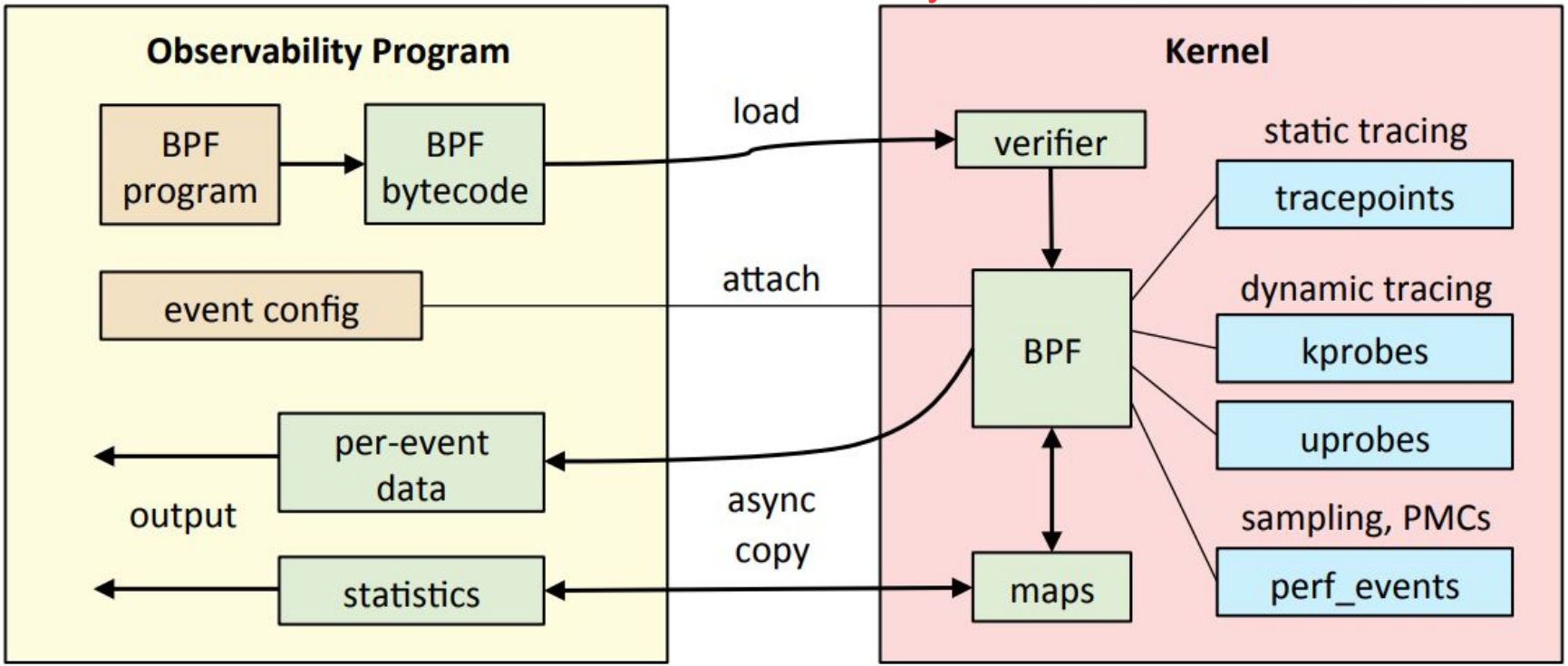
- scheduler
- tracing

Signals of interesting things in the kernel:

- static trace points (kernel trace events)
- dynamic trace points (kprobe)
- userspace dynamic trace points (uprobes)
- userspace static tracepoints (usdt+uprobes)
- perf HW events – PMC counters (cycles, cache misses)
- perf SW events (Ex: Sampling)

BPF for Tracing, Internals

BPF lets you attach and observe them



Enhanced BPF is also now used for SDNs, DDOS mitigation, intrusion detection, container security, ...

Credit: Brendan Gregg

What's BCC?

- BPF Compiler Collection
  - Compile, load, parse.

Kernel
lots of events -> ebpf program -> maps/rb

Userspace
bpf maps -> userspace

Why BCC ?

- In kernel aggregation : No return to userspace or trace Postprocessing

Kernel
lots of events -> ebpf program -> maps/rb

Userspace
bpf maps -> userspace

Why BCC ?

- More efficient sometimes compared to other techniques
  - For example, get a count of stacks

```
stackcount submit_bio
  submit_bio
  __block_write_full_page
  block_write_full_page
  blkdev_writepage
  __writepage
  write_cache_pages
  generic_writepages
  ...
  wb_workfn
  process_one_work
  worker_thread
  kthread
  ret_from_fork
    --
    kworker/u16:0 [16007]
    60
```

Why BCC ?

- More efficient sometimes compared to other techniques
  - Perf way:
    - Record every stack trace to disk using perf record
    - Count them as a second stage
  - eBPF / BCC way
    - Build in-kernel histogram of stack trace.
    - Discard record
    - Return hist to userspace

Note: BCC uses the ever-amazing perf_events framework where it can.

Note: Overhead does exist for high freq. events

This tool kprobes the finish_task_switch function

```
# cpudist &

# perf bench sched pipe -l 100000
     Total time: 4.288 [sec]

# perf bench sched pipe -l 100000
     Total time: 4.020 [sec]
```

~6.6% Overhead

Why BCC ?

- Stats that anyone can collect **without out-of-tree kernel changes**
- Large collection of tools
  - filetop
  - cachetop
  - cachestat
  - biosnoop
  - ext4slower
  - runqlen
  - runqlat
  - trace
- Open ecosystem of common recipes for linux tracing

Demos of BCC tools on Android

filetop: Displays File I/O summary every 5 seconds

```
# filetop 5

Monitor file read/writes (at VFS level).
While filetop is running, create a contact in Android, and see:

TID    COMM             READS  WRITES R_Kb    W_Kb    T FILE
6726   Binder:6152_8    29     0      112     0       R contacts2.db
6726   Binder:6152_8    26     44     104     88      R contacts2.db-wal
2107   servicemanager   16     0      63      0       R current
2107   servicemanager   14     0      55      0       R current
6166   Binder:6152_2    9      0      36      0       R contacts2.db-wal
6166   Binder:6152_2    8      0      32      0       R contacts2.db
5747   Profile Saver    3      0      16      0       R primary.prof
6479   Binder:6152_5    3      0      12      0       R contacts2.db
```

How to get BCC working for Android systems?

Problem:

- android device doesn’t have kernel headers, clang or python to run BCC for ARM64.

- bionic doesn’t have a lot of things needed.

How to get BCC working for Android systems?

Solution 1: Wrote a daemon to run on device and proxy any and all eBPF request. Works great!

Host:

BCC -> adb

Target:

adbd -> BPFd -> kernel

How to get BCC working for Android systems?

Solution 1: Wrote a daemon to run on device and proxy any and all eBPF request. Works great!

Details of project are at:
https://github.com/joelagnel/bpfd
https://lwn.net/Articles/744522/

How to get BCC working for Android systems?

Solution 2: Wrote a tool called androdeb (my current favorite!)

- Packages a full arm64 filesystem using debian tools
- Packages kernel headers from a local kernel tree
- Builds BCC master on device

How to get BCC working for Android systems?

Advantages of using androdeb instead of BPFd:

- Comes with trace-cmd, perf and all the open source friendly tools.
- Able to run BCC tools that can output lots of data (like bcc/trace!)
- Takes about 5 minutes to setup! (rootfs is downloaded from web).

Drawbacks of using androdeb instead of BPFd:

- Takes about 300MB space (can probably we trimmed to 200)
- Requires “adb root” to work.

How to get BCC working for Android systems?

Details of androdeb are at:

https://tinyurl.com/androdeb
(Run BCC on Android in 5 minutes!)

Status: Progress of BCC Journey on Android...

What works in Upstream:

- BCC fixed for ARM64 platforms (Added October '17)
- Support to Compile for any architecture dynamically (Jan '18)
- BCC Support to compile eBPF on custom kernel tree path (Jan '18)
- Preliminary support for BCC communicating to remote targets (Jan '18)
- BPFd idea inception (https://lwn.net/Articles/744522/)
- Refactoring BCC to make it easier to add remote support merged (Feb ‘18).
- androdeb project created (March ‘18)
- Fixes to cachestat, and userspace sym lookup for Android (April ‘18)

Pending Upstream review:

- BCC remote support to talk to remote targets (Pushed April ‘18)

Demos of BCC tools on Android

hardirq: Total time spent in hard interrupt handlers

Example. Start and minimize an app a lot, watch the mali interrupts total time:

```
# ./tools/hardirqs.py 10
Tracing hard irq event time... Hit Ctrl-C to end.
HARDIRQ                    TOTAL_usecs
wl18xx                             181
ufshcd                             243
dw-mci                             409
hisi-asp-dma                      2671
mailbox-2                         2842
timer                             9978
xhci-hcd:usb1                    12468
kirin                            13720
e82c0000.mali                    60635
```

Demos of BCC tools on Android

cachestat: Page Cache Hits and Misses

```
# cachestat  1
   TOTAL   MISSES     HITS  DIRTIES   BUFFERS_MB  CACHED_MB
  165849        0   165849        0            2       1344
  114970        0   114970        0            2       1344
  269136        0   269136        0            2       1344
  253217        0   253217        0            2       1344
   14772        0    14772        0            2       1344
  280407        0   280407        0            2       1344
  268758        0   268758        0            2       1344
    8889        0     8889        0            2       1344
  264589        0   264589        0            2       1343
  276801       80   276721        0            2       1343
   18552    18552        0        0            0        387  <--- Did a “echo 1 > drop_caches”
  194915   183908    11007        0            0       1108
   68699    58350    10349        0            0       1327
  268413     3152   265261        0            0       1335
   13503      360    13143        0            0       1335
  267042      180   266862        0            0       1334
  269224      461   268763        0            0       1334
   12713        0    12713        0            0       1334
```

Demos of BCC tools on Android

runqlen: Per-CPU Histogram of run queue lengths

```
# runqlen -C
cpu = 4
     runqlen       : count     distribution
        0          : 68       |****************************************|

cpu = 5
     runqlen       : count     distribution
        0          : 49       |****************************************|

cpu = 6
     runqlen       : count     distribution
        0          : 0        |                                        |
        1          : 79       |********************                    |
        2          : 10       |**                                      |
        3          : 81       |********************                    |
        4          : 149      |****************************************|
```

Demos of BCC tools on Android

runqlen: Histogram of run queue lengths

```
# taskset -a -c 6 hackbench -P -g 2 -f 2 -l 10000000 &
   (Total of 8 tasks)

# runqlen

Sampling run queue length... Hit Ctrl-C to end.
^C
     runqlen       : count     distribution
        0          : 1080     |****************************************|
        1          : 98       |***                                     |
        2          : 11       |                                        |
        3          : 64       |**                                      |
        4          : 105      |***                                     |
        5          : 1        |                                        |
        6          : 0        |                                        |
```

Demos of BCC tools on Android

runqlat: show run queue latencies

```
# taskset -a -c 6 hackbench -P -g 14 -f 2 -l 10000000 &
# runqlat
     usecs               : count     distribution
         0 -> 1          : 22       |                                        |
         2 -> 3          : 68       |                                        |
         4 -> 7          : 166      |                                        |
         8 -> 15         : 23718    |****************************************|
        16 -> 31         : 19301    |********************************        |
        32 -> 63         : 2887     |****                                    |
        64 -> 127        : 1684     |**                                      |
       128 -> 255        : 2127     |***                                     |
       256 -> 511        : 2461     |****                                    |
       512 -> 1023       : 2927     |****                                    |
      1024 -> 2047       : 86       |                                        |
      2048 -> 4095       : 42       |                                        |
      4096 -> 8191       : 7        |                                        |
      8192 -> 16383      : 2        |                                        |
     16384 -> 32767      : 4        |                                        |
     32768 -> 65535      : 5        |                                        |
     65536 -> 131071     : 317      |                                        |
    131072 -> 262143     : 1        |                                        |
```

Trace Multitool : A swiss army knife

Usecase: Using dynamic tracepoints (kprobes)

Function we'd like to trace has prototype:

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode);
```

```
# trace 'do_sys_open "%s", arg2' -T

TIME     PID     TID     COMM            FUNC             -
19:45:44 2220    2250    storaged        do_sys_open      /sys/block/sda/stat
19:45:44 2220    2250    storaged        do_sys_open      /sys/block/sda/stat
19:45:48 2132    2132    servicemanager  do_sys_open      /proc/4113/attr/current
19:45:49 2352    2437    DeviceStorageMo do_sys_open      /system/framework/arm/boot.art
19:45:49 2352    2437    DeviceStorageMo do_sys_open      ../system@framework@boot.art
19:45:49 2352    2437    DeviceStorageMo do_sys_open      /system/framework/arm64/boot.art
19:45:49 2352    2437    DeviceStorageMo do_sys_open      ../system@framework@boot.art
19:45:55 2132    2132    servicemanager  do_sys_open      /proc/2480/attr/current
19:45:55 2132    2132    servicemanager  do_sys_open      /proc/2480/attr/current
```

Trace Multitool : A swiss army knife

Usecase: kernel tracepoints (although I'd stick to trace-cmd for TPs)

```
# trace 't:block:block_rq_complete "sectors=%d", args→nr_sector'

PID     TID     COMM            FUNC             -
0       0       swapper/0       block_rq_complete sectors=64
0       0       swapper/0       block_rq_complete sectors=0
0       0       swapper/0       block_rq_complete sectors=8
0       0       swapper/0       block_rq_complete sectors=0
0       0       swapper/0       block_rq_complete sectors=80
0       0       swapper/0       block_rq_complete sectors=0
```

Demos of tools: argdist

Example: Get a historgram of size parameter passed to __kmalloc

```
# argdist -i 1 -H 'p::__kmalloc(size_t size):size_t:size'

     size               : count     distribution
         0 -> 1         : 0        |                                        |
         2 -> 3         : 0        |                                        |
         4 -> 7         : 1        |                                        |
         8 -> 15        : 217      |****************************************|
        16 -> 31        : 21       |***                                     |
        32 -> 63        : 178      |*******************************         |
        64 -> 127       : 20       |***                                     |
       128 -> 255       : 5        |                                        |
       256 -> 511       : 7        |*                                       |
       512 -> 1023      : 8        |*                                       |
      1024 -> 2047      : 2        |                                        |
      2048 -> 4095      : 0        |                                        |
      4096 -> 8191      : 12       |**                                      |
```

Open discussion: Lock Contention detection tool

Current solution: Monitor futex functions in kernel and identify sleepers/wakers

Problem: futex isn't only used for locking.

- How to detect futex is a lock?
  - Analyzing the userspace stack
    - Problem: Very userspace-specific
  - Can we monitor anything about futex usage?
    - timing?
    - parameters?

- How to provide more information about which lock?

Open ideas for new tools relevant to Scheduler/Power

- Write new tools relevant to OSPM … Ideas ?
  - Calculating average power calculated from EM and cpufreq events
  - Identifying other common problems..
    - Is load balancing running often enough and doing the right thing?
    - Is EAS not hurting performance sensitive tasks?
    - Are we wasting too much power by not going to deeper idle enough?
  - Scheduler workload characterization for unit tests (BCC has a ‘sched-time’)
    - Seems to be “ballpark” characterization (doesn’t account for every sleep/wakeup, just models dependencies correctly….)

Resources

- Androdeb: https://tinyurl.com/androdeb
- BPFd project: https://github.com/joelagnel/bpfd
- LWN article: https://lwn.net/Articles/744522/
- Brendan Gregg's eBPF page: http://brendangregg.com/perf.html#eBPF

Thanks

- Brendan Gregg, Alexei Staravoitov and Sasha Goldstein for encouragement.
- Todd Kjos for help with androdeb.
- Android kernel team for encouragement and ideas.
- OSPM team

Questions?