```
# biolatency.bt
Attaching 3 probes...
Tracing block device I/O... Hit Ctrl-C to end.
^C

@usecs:
[256, 512)             2 |                                                    |
[512, 1K)             10 |@                                                   |
[1K, 2K)             426 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[2K, 4K)             230 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                    |
[4K, 8K)               9 |@                                                   |
[8K, 16K)            128 |@@@@@@@@@@@@@@@@                                    |
[16K, 32K)            68 |@@@@@@@@                                            |
[32K, 64K)             0 |                                                    |
[64K, 128K)            0 |                                                    |
[128K, 256K)          10 |@                                                   |
```

bpftrace

Your name here

Conference name
Month Year

These are sample bpftrace slides that I (Brendan Gregg) have created, and I give you (anyone) permission to use some or all of them to give bpftrace conference/meetup talks. The LibreOffice source is at https://github.com/iovisor/bpf-docs Just edit/delete the red text!

Experience: *Your story here*

Best way to start is to tell your own story of using bpftrace to solve a problem. Don't have a good story to tell yet? You could try:

- running execsnoop.bt on a production server for 60 minutes. What did you see?
- running opensnoop.bt during application startup. Anything surprising?
- running tcpconnect.bt: any unexpected connections?
- running tcpretrans.bt on a busy server: any pattern to the retransmits?
- running biosnoop.bt, biolatency.bt, and bitesize.bt to examine disk workloads.
- working through docs/tutorial_one_liners.md with some workloads.
- developing your own one-liners and tools to solve something.

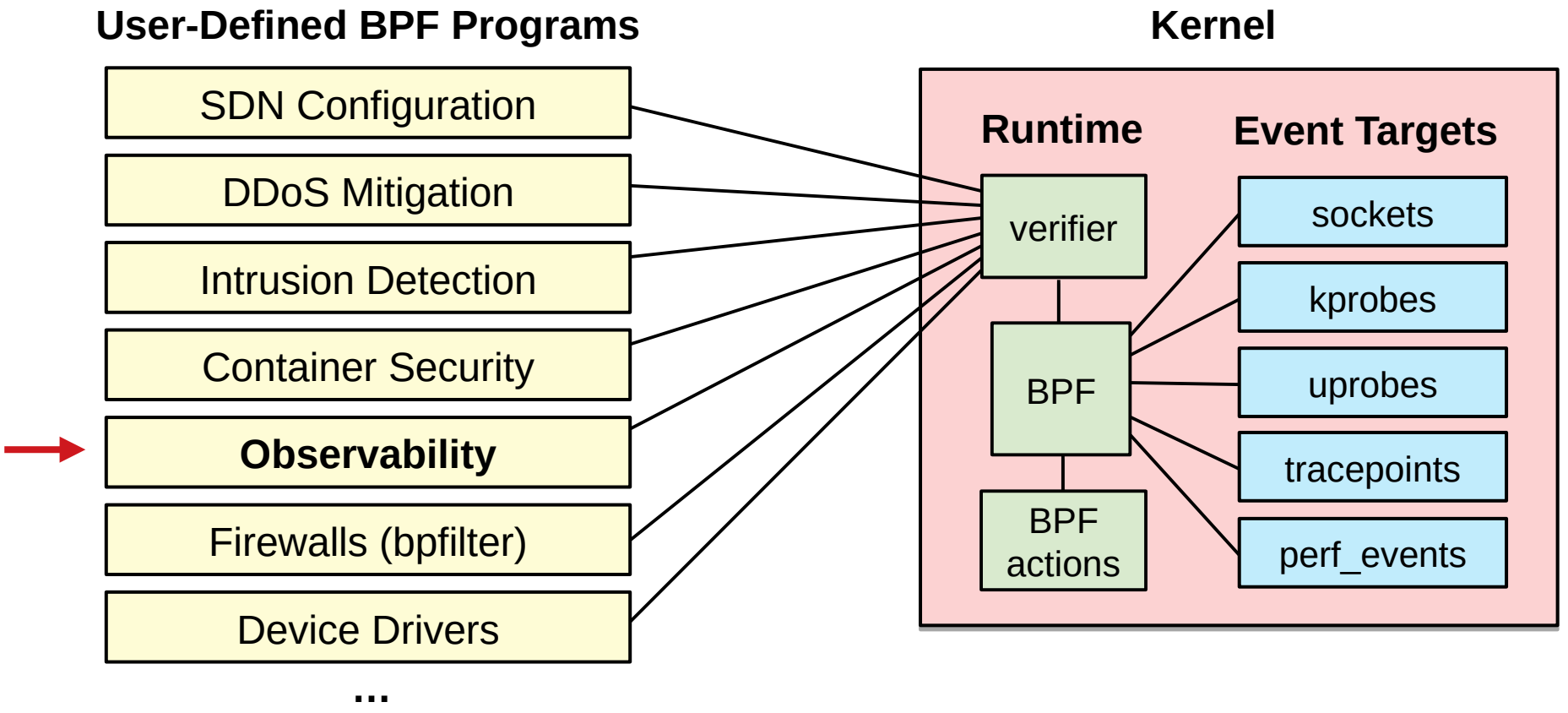
Also consider starting with a live demo.

Experience: *your story here*

```
# opensnoop.bt
Attaching 3 probes...
Tracing open syscalls... Hit Ctrl-C to end.
PID    COMM               FD ERR PATH
2440   snmp-pass           4   0 /proc/cpuinfo
2440   snmp-pass           4   0 /proc/stat
25706  ls                  3   0 /etc/ld.so.cache
25706  ls                  3   0 /lib/x86_64-linux-gnu/libselinux.so.1
25706  ls                  3   0 /lib/x86_64-linux-gnu/libc.so.6
25706  ls                  3   0 /lib/x86_64-linux-gnu/libpcre.so.3
25706  ls                  3   0 /lib/x86_64-linux-gnu/libdl.so.2
25706  ls                  3   0 /lib/x86_64-linux-gnu/libpthread.so.0
25706  ls                  3   0 /proc/filesystems
25706  ls                  3   0 /usr/lib/locale/locale-archive
25706  ls                  3   0 .
1744   snmpd               8   0 /proc/net/dev
1744   snmpd              21   0 /proc/net/if_inet6
[...]
```

Highlight the interesting bits

eBPF: extended Berkeley Packet Filter



bpftrace: BPF observability front-end

https://github.com/iovisor/bpftrace
Built from the ground-up for BPF and Linux
Used in production at Netflix, Facebook, etc

Custom one-liners
Tools

Linux 4.9+

bpftrace One-liners

```
# Files opened by process
bpftrace -e 't:syscalls:sys_enter_open { printf("%s %s\n", comm,
    str(args->filename)) }'

# Read size distribution by process
bpftrace -e 't:syscalls:sys_exit_read { @[comm] = hist(args->ret) }'

# Count VFS calls
bpftrace -e 'kprobe:vfs_* { @[func]++ }'

# Show vfs_read latency as a histogram
bpftrace -e 'k:vfs_read { @[tid] = nsecs }
    kr:vfs_read /@[tid]/ { @ns = hist(nsecs - @[tid]); delete(@tid) }’

# Trace user-level function
bpftrace -e 'uretprobe:bash:readline { printf(“%s\n”, str(retval)) }’
…
```

bpftrace Tools

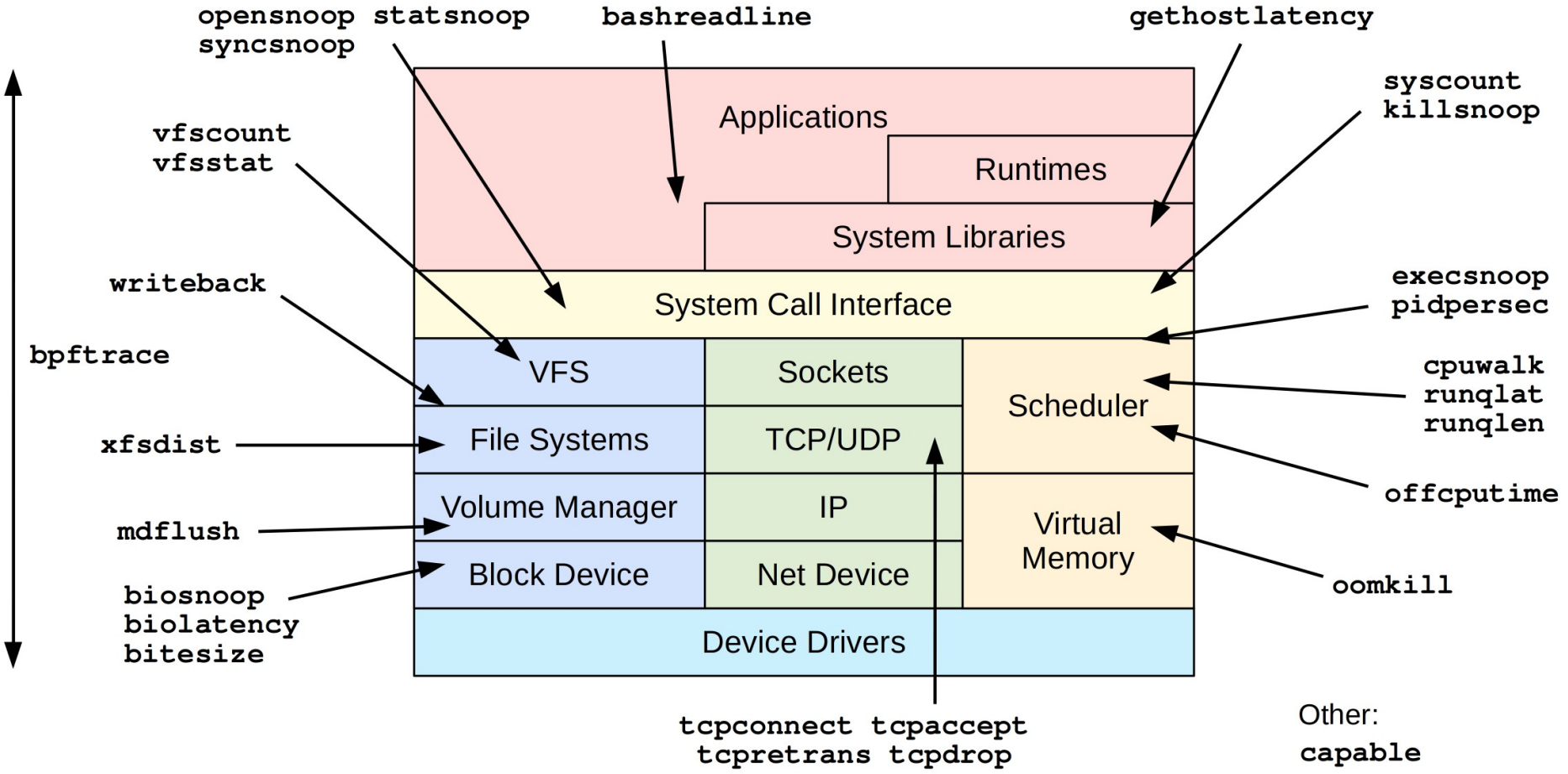


Diagram by Brendan Gregg, early 2019. https://github.com/iovisor/bpftrace

Tool: gethostlatency

```
# gethostlatency.bt
Attaching 7 probes...
Tracing getaddr/gethost calls... Hit Ctrl-C to end.
TIME      PID    COMM              LATms HOST
02:52:05  19105  curl                 81 www.netflix.com
02:52:12  19111  curl                 17 www.netflix.com
02:52:19  19116  curl                  9 www.facebook.com
02:52:23  19118  curl                  3 www.facebook.com
```

This is an example of user-level tracing, rather than kernel-level tracing: it's tracing the resolver calls from libc. You could include the source to show that.

Tool: runqlen

```
# runqlen.bt
Attaching 2 probes...
Sampling run queue length at 99 Hertz... Hit Ctrl-C to end.
^C

@runqlen:
[0, 1)              1967 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1, 2)                 0 |                                                          |
[2, 3)                 0 |                                                          |
[3, 4)               306 |@@@@@@@@                                                  |
```

There's also runqlat, which shows the latency when waiting for a turn on-CPU, but runqlat traces every context switch which can add some overhead. runqlen, on the other hand, simply samples the length of the run queues. It's not as useful as runqlat, however, it has a tiny and fixed cost.

Tool: *more one-liner or tool screenshots*

```
# …
```

Also add more case studies if you have them

Also consider using bpftrace to dump data that you then visualize as graphs, scatter plots, line graphs, flame graphs, etc. Eg, save the output of biosnoop.bt to a file, then import it into Google spreadsheets and do a scatter plot of the TIME and LAT columns.

bpftrace Syntax

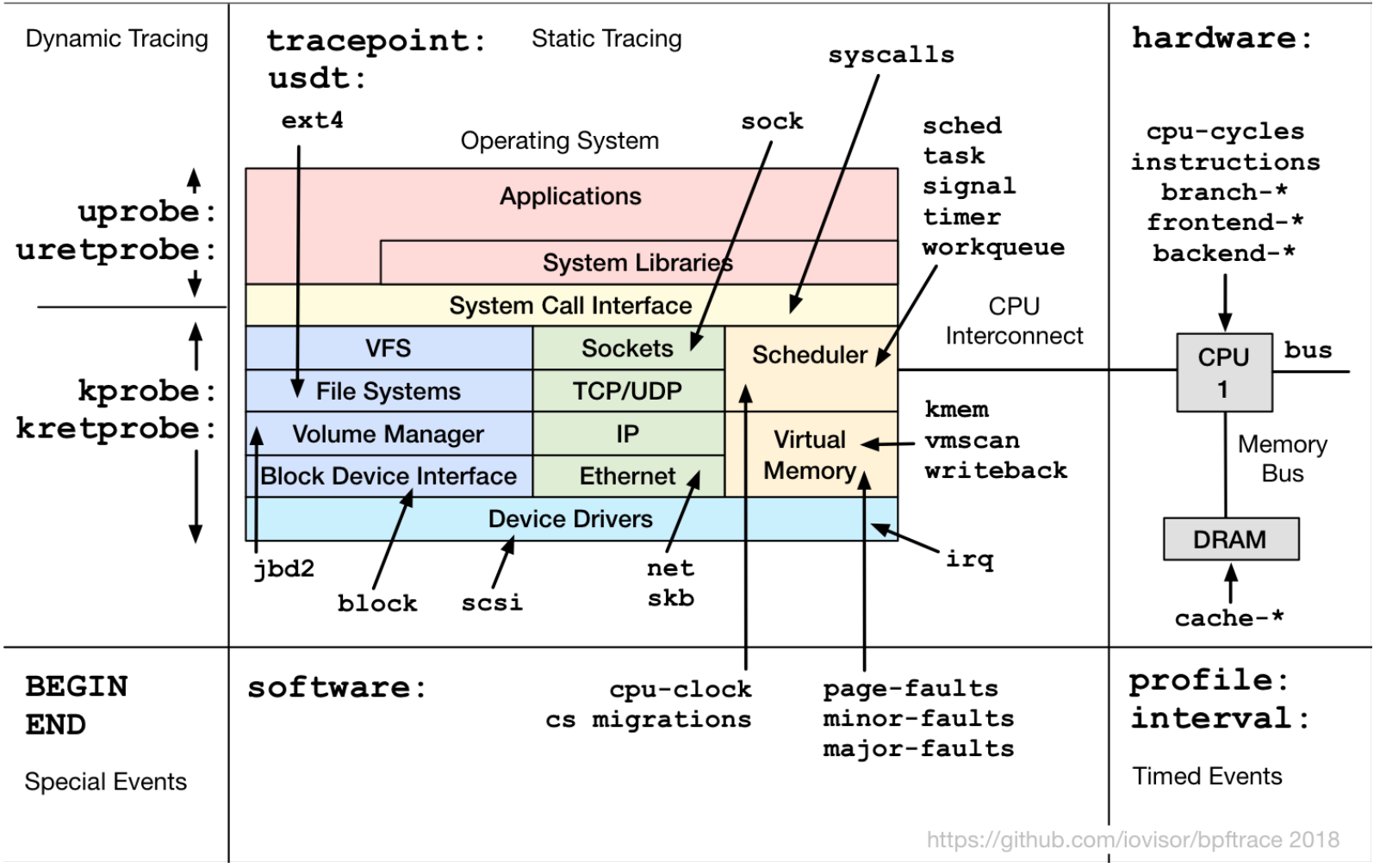
```
bpftrace -e 'k:do_nanosleep /pid > 100/ { @[comm]++ }'
```

Probe

Filter
(optional)

Action

Probes


Dynamic Tracing
tracepoint:
usdt:
Static Tracing
syscalls
hardware:
ext4
sock
Operating System
sched
task
signal
timer
workqueue
cpu-cycles
instructions
branch-*
frontend-*
backend-*
uprobe:
uretprobe:
Applications
System Libraries
System Call Interface
CPU
Interconnect
CPU
1
bus
VFS
Sockets
Scheduler
kprobe:
kretprobe:
File Systems
TCP/UDP
kmem
vmscan
writeback
Volume Manager
IP
Virtual
Memory
Memory
Bus
Block Device Interface
Ethernet
DRAM
Device Drivers
irq
jbd2
block
scsi
net
skb
cache-*
BEGIN
END
Special Events
software:
cpu-clock
cs migrations
page-faults
minor-faults
major-faults
profile:
interval:
Timed Events
https://github.com/iovisor/bpftrace 2018

Probe Type Shortcuts

`tracepoint`	`t`	Kernel static tracepoints
`usdt`	`U`	User-level statically defined tracing
`kprobe`	`k`	Kernel function tracing
`kretprobe`	`kr`	Kernel function returns
`uprobe`	`u`	User-level function tracing
`uretprobe`	`ur`	User-level function returns
`profile`	`p`	Timed sampling across all CPUs
`interval`	`i`	Interval output
`software`	`s`	Kernel software events
`hardware`	`h`	Processor hardware events

Filters

- **`/pid == 181/`**
- **`/comm != “sshd”/`**
- **`/@ts[tid]/`**

Actions

- Per-event output
  - **`printf()`**
  - **`system()`**
  - **`join()`**
  - **`time()`**
- Map Summaries
  - **`@ = count()`** or **`@++`**
  - **`@ = hist()`**
  - `...`

The following is in the https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md

Functions

- **`hist(n)`** Log2 histogram
- **`lhist(n, min, max, step)`** Linear hist.
- **`count()`** Count events
- **`sum(n)`** Sum value
- **`min(n)`** Minimum value
- **`max(n)`** Maximum value
- **`avg(n)`** Average value
- **`stats(n)`** Statistics
- **`str(s)`** String
- **`ksym(p)`** Resolve kernel addr
- **`usym(p)`** Resolve user addr
- **`kaddr(n)`** Resolve kernel symbol
- **`uaddr(n)`** Resolve user symbol
- **`printf(fmt, ...)`** Print formatted
- **`print(@x[,top[,div]])`** Print map
- **`ntop([af,]addr)`** IP address to string
- **`delete(@x)`** Delete map element
- **`clear(@x)`** Delete all keys/values
- **`reg(n)`** Register lookup
- **`join(a)`** Join string array
- **`time(fmt)`** Print formatted time
- **`system(fmt)`** Run shell command
- **`cat(file)`** Print file contents
- **`exit()`** Quit bpftrace

Variable Types

- Basic Variables
  - **`@global`**
  - **`@thread_local[tid]`**
  - **`$scratch`**
- Associative Arrays
  - **`@array[key] = value`**
- Buitins
  - **`pid`**
  - **`...`**

Builtin Variables

- **`pid`** Process ID (kernel tgid)
- **`tid`** Thread ID (kernel pid)
- **`cgroup`** Current Cgroup ID
- **`uid`** User ID
- **`gid`** Group ID
- **`nsecs`** Nanosecond timestamp
- **`cpu`** Processor ID
- **`comm`** Process name
- **`kstack`** Kernel stack trace
- **`ustack`** User stack trace

- **`arg0`**, **`arg1`**, ... Function args
- **`retval`** Return value
- **`args`** Tracepoint args
- **`func`** Function name
- **`probe`** Full probe name
- **`curtask`** Curr task_struct (u64)
- **`rand`** Random number (u32)

Tool: biolatency

```
# biolatency.bt
Attaching 3 probes...
Tracing block device I/O... Hit Ctrl-C to end.
^C

@usecs:
[256, 512)             2 |                                                    |
[512, 1K)             10 |@                                                   |
[1K, 2K)             426 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[2K, 4K)             230 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4K, 8K)               9 |@                                                   |
[8K, 16K)            128 |@@@@@@@@@@@@@@@@                                    |
[16K, 32K)            68 |@@@@@@@@                                            |
[32K, 64K)             0 |                                                    |
[64K, 128K)            0 |                                                    |
[128K, 256K)          10 |@                                                   |
```

Tool: biolatency

```
#!/usr/local/bin/bpftrace

BEGIN
{
    printf("Tracing block device I/O... Hit Ctrl-C to end.\n");
}

kprobe:blk_account_io_start
{
    @start[arg0] = nsecs;
}

kprobe:blk_account_io_done
/@start[arg0]/
{
    @usecs = hist((nsecs - @start[arg0]) / 1000);
    delete(@start[arg0]);
}
```

Advanced tool: runqlat

Just an example to show that yes, we can walk structs

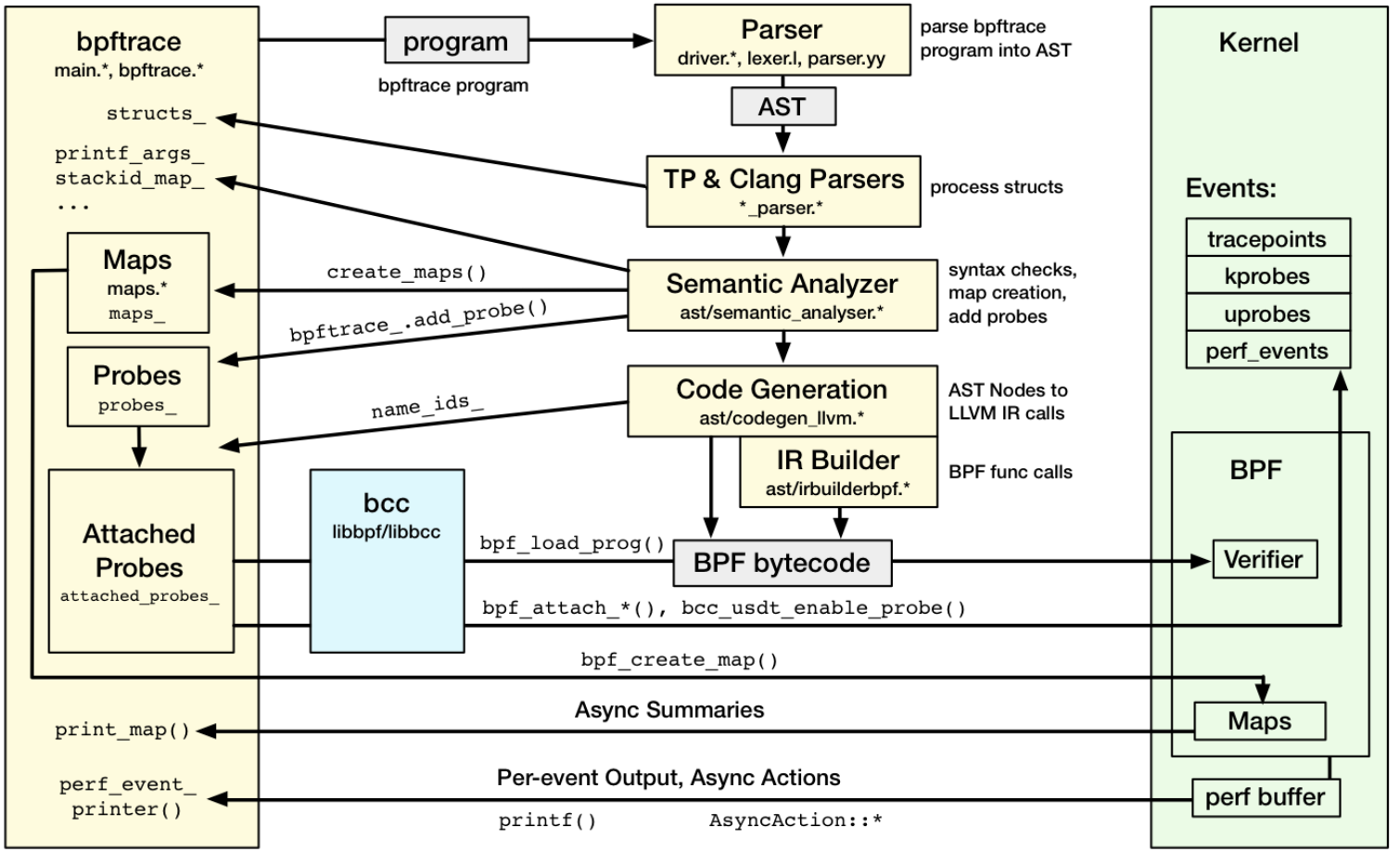
```
#!/usr/local/bin/bpftrace
#include <linux/sched.h>

// Until BTF is available, we'll need to declare some of this struct manually,
// since it isn't avaible to be #included. This will need maintenance to match
// your kernel version. It is from kernel/sched/sched.h:
struct cfs_rq_partial {
     struct load_weight load;
     unsigned long runnable_weight;
     unsigned int nr_running;
     unsigned int h_nr_running;
};

BEGIN {
     printf("Sampling run queue length at 99 Hertz... Hit Ctrl-C to end.\n");
}

profile:hz:99 {
     $task = (struct task_struct *)curtask;
     $my_q = (struct cfs_rq_partial *)$task->se.cfs_rq;
     $len = $my_q->nr_running;
     $len = $len > 0 ? $len - 1 : 0;    // subtract currently runing task
     @runqlen = lhist($len, 0, 100, 1);
}
```

Many kernel structs are in the kernel headers package, which bpftrace will use. But sometimes a struct is missing, and we declare some here. BTF (BPF Type Format) will solve this problem in the future by adding a lightweight debuginfo to the kernel. See: https://www.kernel.org/doc/Documentation/bpf/btf.rst

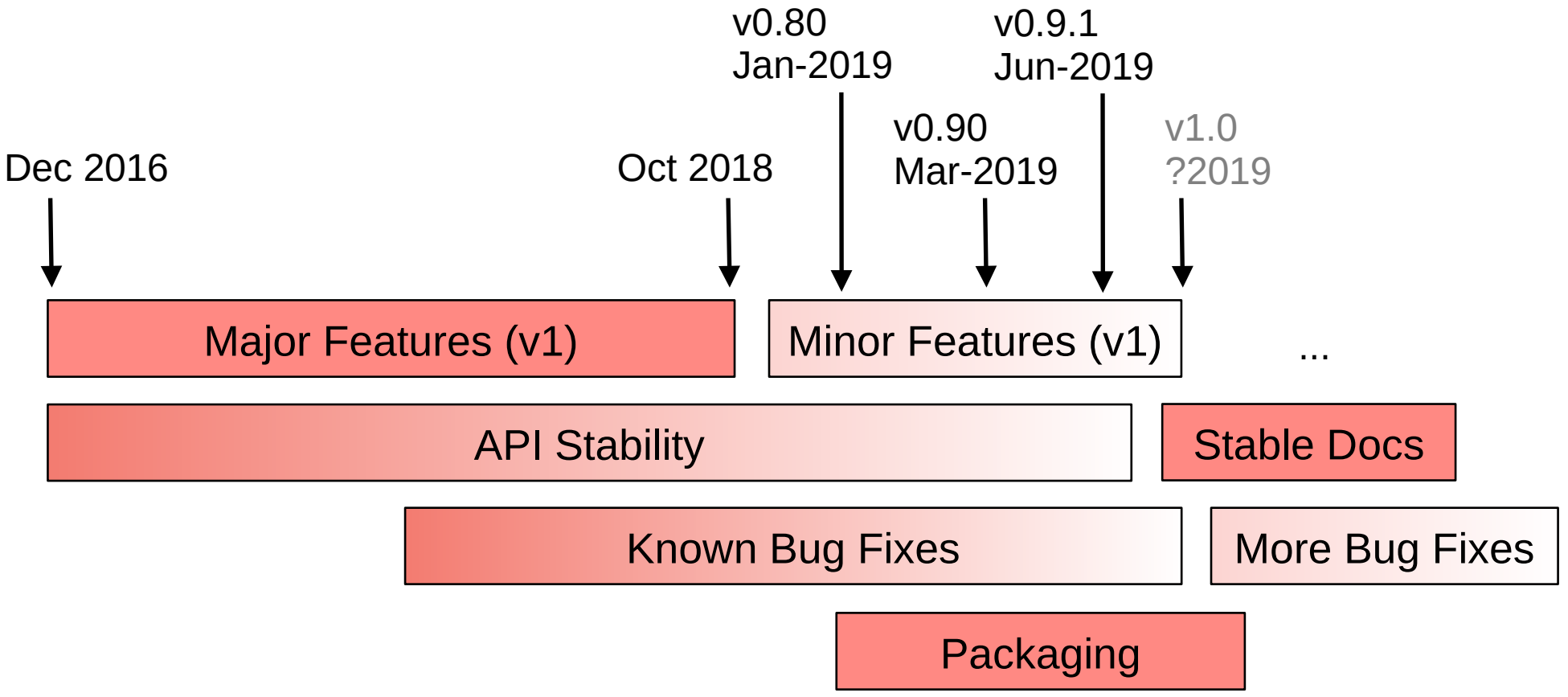
Lots of info can be found from task_struct

bpftrace Internals

Future versions may use a builtin BPF compiler, avoiding the LLVM/Clang dependencies



bpftrace Development



ply: a lightweight BPF-based front-end

```
# ply 'tracepoint:syscalls/sys_enter_open {
    printf("PID: %d (%s) opening: %s\n", pid, comm, str(data->filename)); }'
ply: active
PID: 22737 (Chrome_IOThread) opening: /dev/shm/.org.chromium.Chromium.dh4msB
PID: 22737 (Chrome_IOThread) opening: /dev/shm/.org.chromium.Chromium.dh4msB
PID: 22737 (Chrome_IOThread) opening: /dev/shm/.org.chromium.Chromium.2mIlx4
[...]
```

Can be worth mentioning, as it is lightweight (little dependencies, although bpftrace may head in that direction too) and suited for some environments.
It is also built from the ground-up for BPF and Linux.
It is more limited in functionality. Syntax very similar to bpftrace.

https://github.com/iovisor/ply

Takeaway:

bpftrace all the things!

URLs

- https://github.com/iovisor/bpftrace
  - https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners.md
  - https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md
- http://www.brendangregg.com/ebpf.html#bpftrace
- http://www.brendangregg.com/bpfperftools.html
- https://github.com/iovisor/kubectl-trace
- https://tracker.debian.org/pkg/bpftrace

Thanks

- bpftrace
  - Alastair Robertson (creator)
  - Netflix: Brendan Gregg, Matheus Marchini
  - Sthima: Willian Gaspar
  - Facebook: Jon Haslam, Dan Xu
  - Augusto Mecking Caringi, Dale Hamel, ...
- eBPF & bcc
  - Facebook: Alexei Starovoitov, Teng Qin, Yonghong Song, Martin Lau, Mark Drayton, …
  - Netflix: Brendan Gregg
  - VMware: Brenden Blanco
  - Daniel Borkmann, David S. Miller, Sasha Goldsthein, Paul Chaignon, …
- Slides from Brendan Gregg, used with permission