SIGCOMM 2024

— SYDNEY —

eBPF

eBPF and Pefrormance

What, Why, How, and What's Next

Brendan Gregg

Statement from the heart

I’d like to begin by acknowledging the Traditional Owners of this land and pay my respects to Elders past and present.

This Keynote is about Performance Engineering

Australia typically does:

- Load Testing
- Performance Monitoring

Buy product, run product
Capacity planning
Basic SW/HW evaluations

...but not:

- Performance Engineering

Root-cause analysis (incl. runtimes, kernel, metal)
Custom perf tool development
Perf feature/fix development (incl. open source)
Vendor design collaboration (incl. pre-silicon)

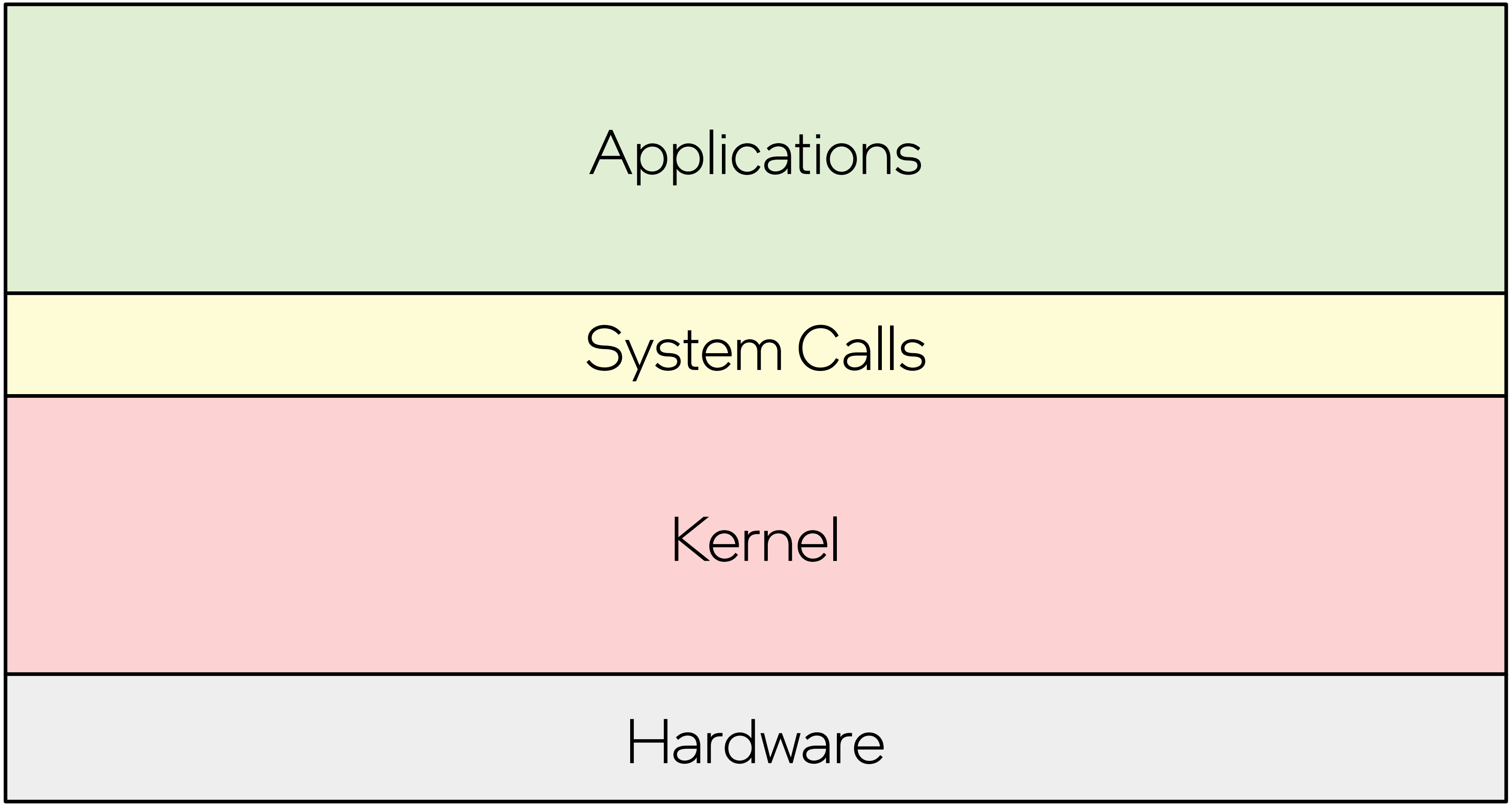
There are many companies with significant performance engineering teams, including:
Intel, AMD, Nvidia, Meta, Google, X (Twitter), Amazon,
Netflix, eBay, Salesforce, Pinterest, etc.

Agenda

1) **What: A type of software**
2) Why: Case Study
3) How: History, Internals, Usage, Recommendations
4) What's Next: Challenges, Future
5) Discussion & Q&A

(This section is my type-of-software summary; Daniel Borkmann went deeper in parts for SIGCOMM 2023)

50 Years, one (dominant) OS model



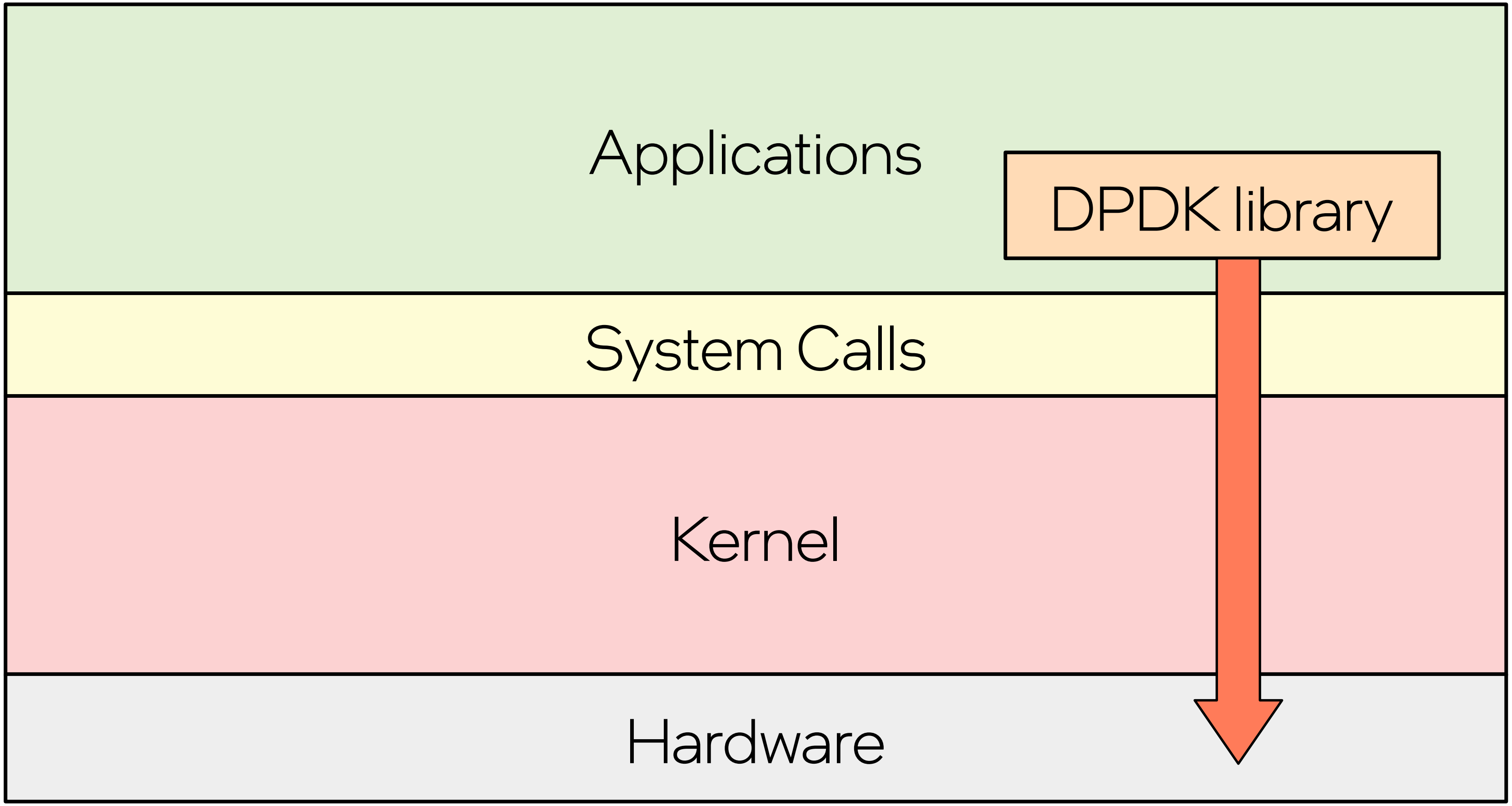
Applications

System Calls

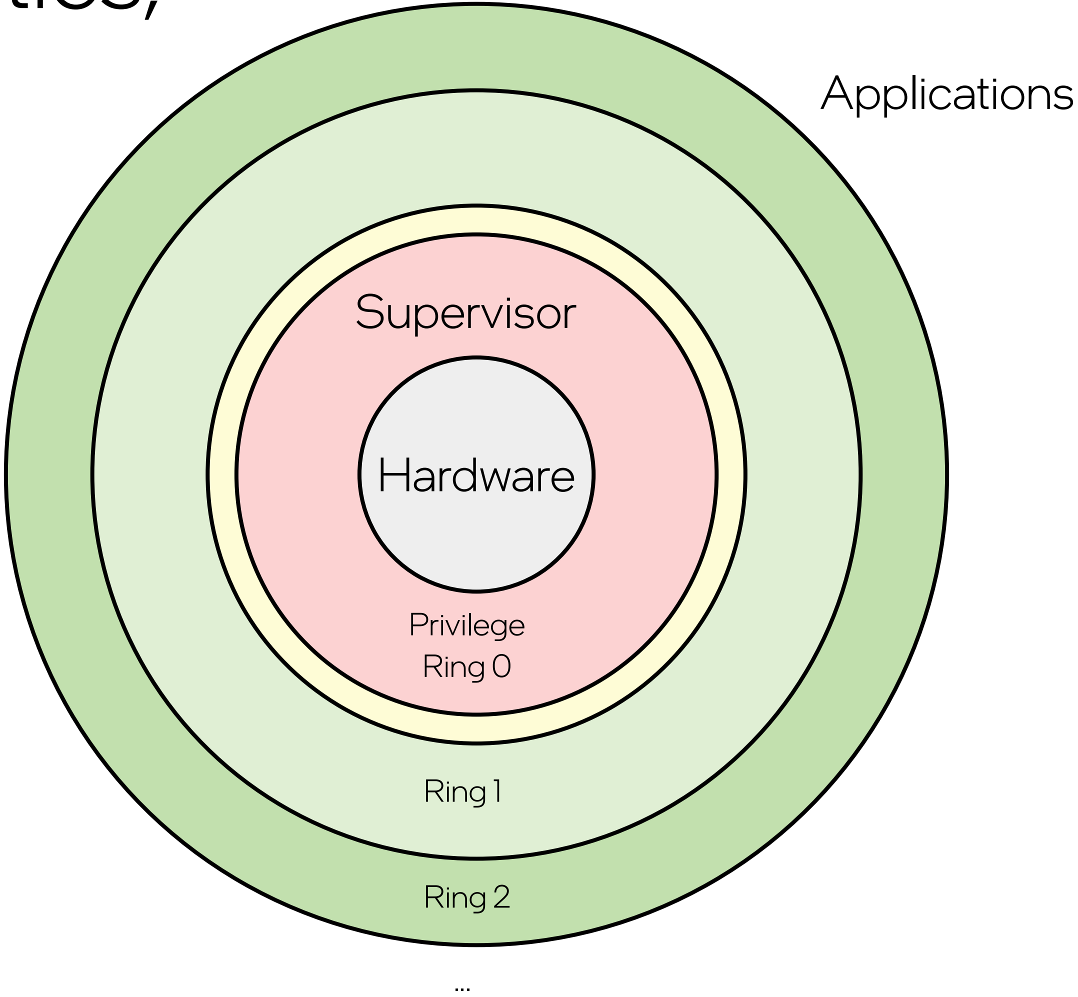
Kernel

Hardware

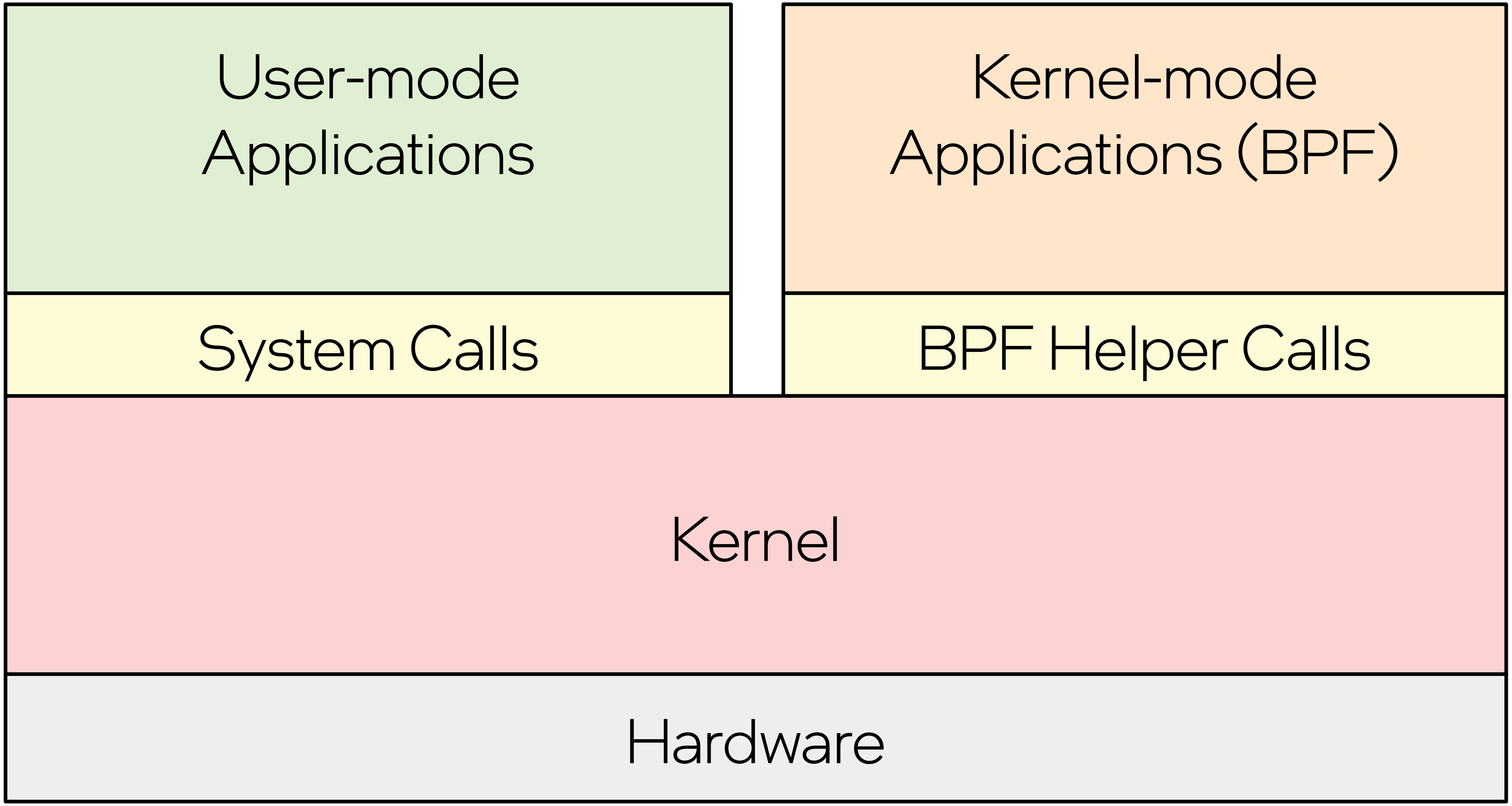
…requiring workarounds for high performance



Origins: Multics, 1960s



Modern day: A new OS model



User-mode Applications

Kernel-mode Applications (BPF)

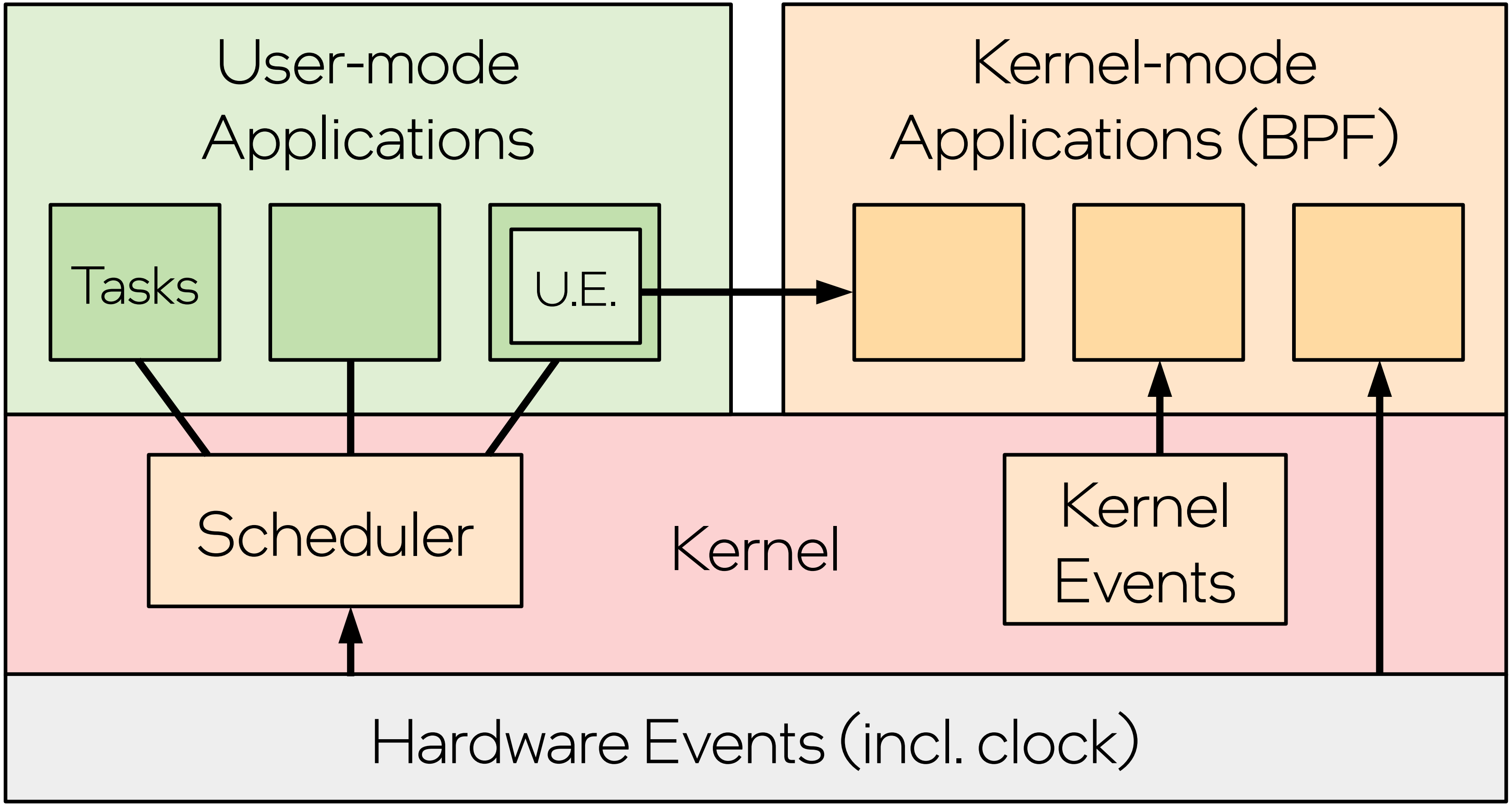
System Calls

BPF Helper Calls

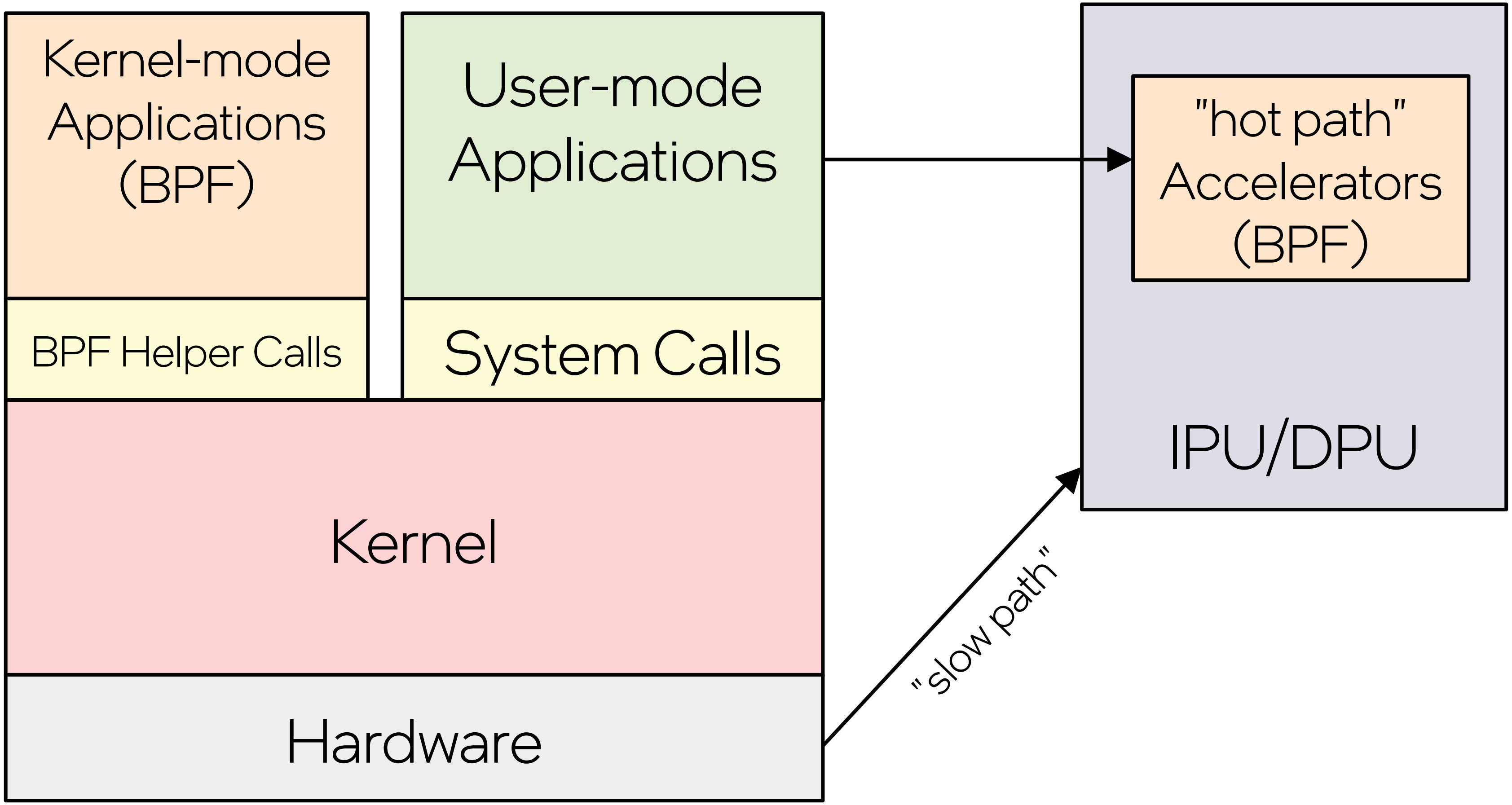
Kernel

Hardware

A different execution model



Future: A new take on "edge computing"

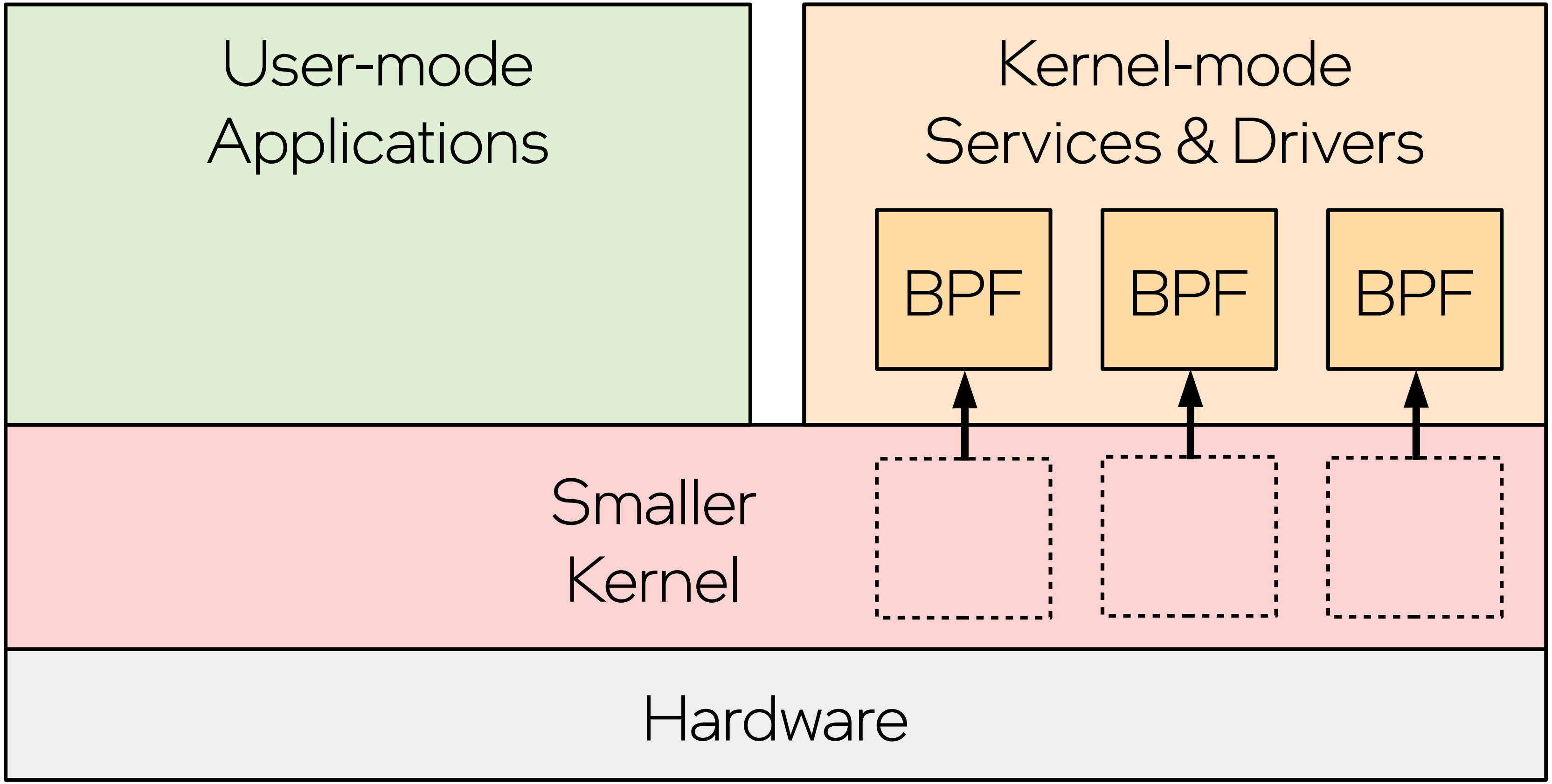


A New Type of Software

	Execution model	User defined	Compilation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none (code reviews)	panic	direct
BPF	event	yes	JIT, CO-RE	verified, JIT	error message	restricted helpers, kfuncs

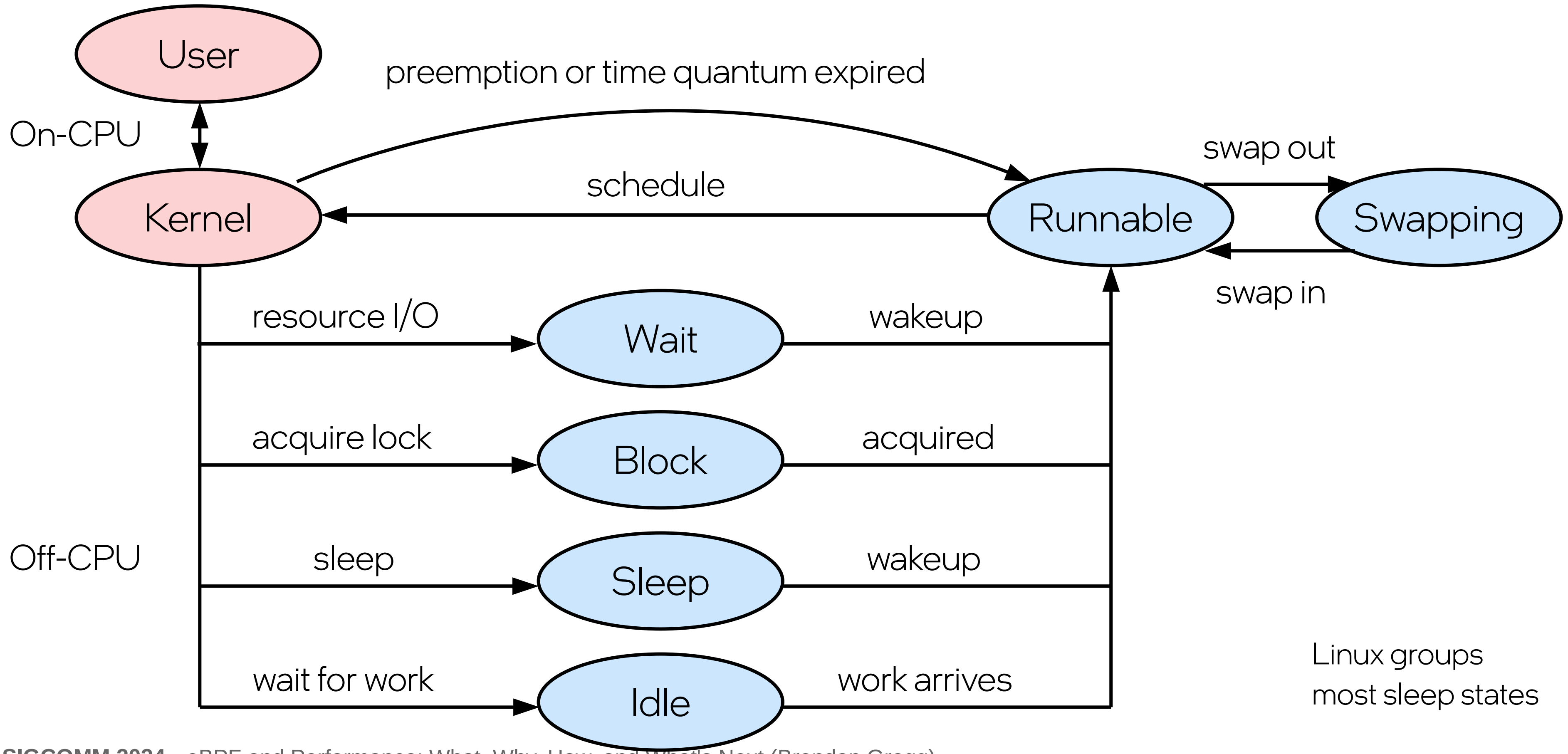
Updated by Daniel Borkmann, SIGCOMM 2023

Modern Linux is becoming Microkernel-ish



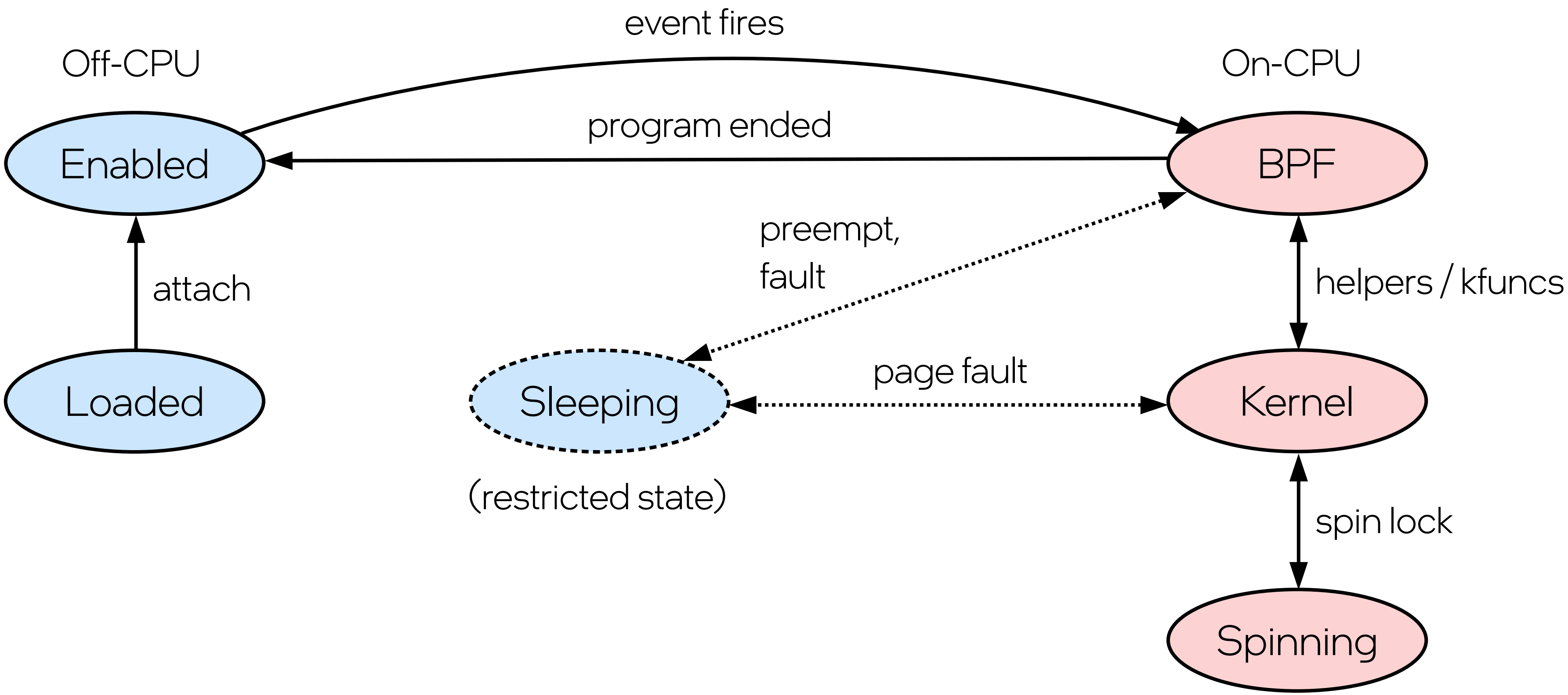
The word "microkernel" has already been invoked by Jonathan Corbet, Thomas Graf, Greg Kroah-Hartman, ...

50 Years, one process state model



Linux groups
most sleep states

BPF program state model



Agenda

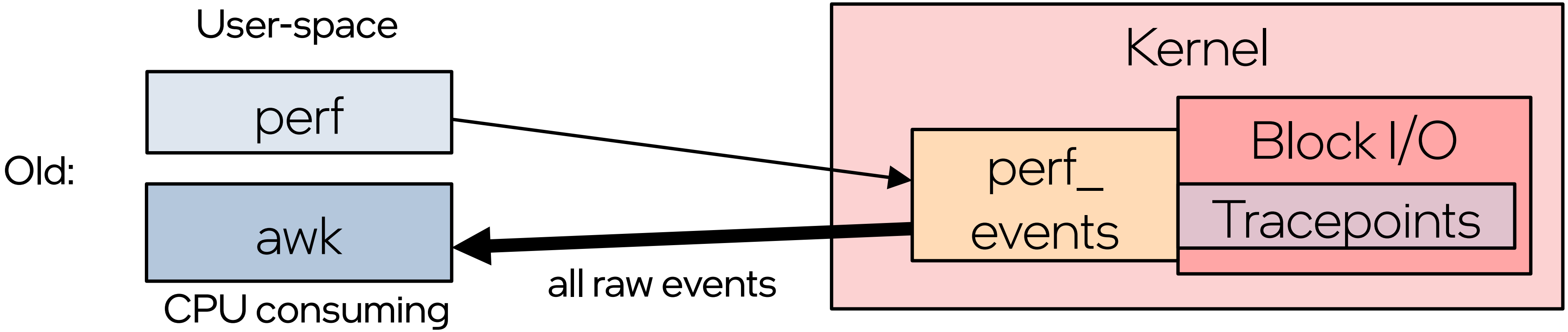
1) What: A type of software
2) **Why: Case Study**
3) How: History, Internals, Usage, Recommendations
4) What's Next: Challenges, Future
5) Discussion & Q&A

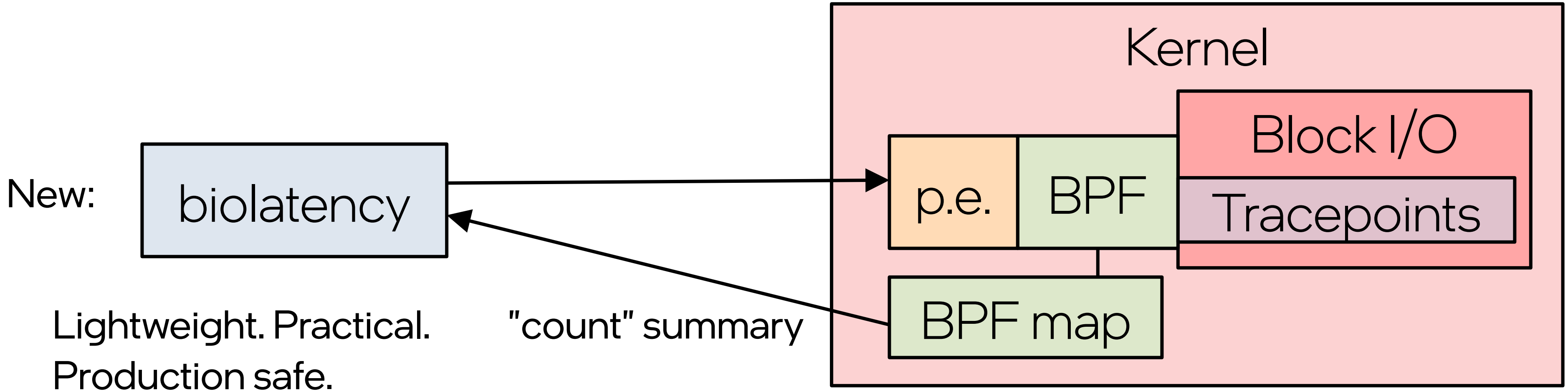
Example BPF perf tool: biolatency

What is the distribution of disk I/O latency? Per second?

```
# ./biolatency -mT 1 5
Tracing block device I/O... Hit Ctrl-C to end.

06:20:16
     msecs           : count     distribution
       0 -> 1        : 36       |************************************** |
       2 -> 3        : 1        |*                                      |
       4 -> 7        : 3        |***                                    |
       8 -> 15       : 17       |*****************                      |
      16 -> 31       : 33       |**********************************     |
      32 -> 63       : 7        |*******                                |
      64 -> 127      : 6        |******                                 |

06:20:17
     msecs           : count     distribution
       0 -> 1        : 96       |************************************   |
       2 -> 3        : 25       |*********                              |
       4 -> 7        : 29       |***********                            |
[...]
```

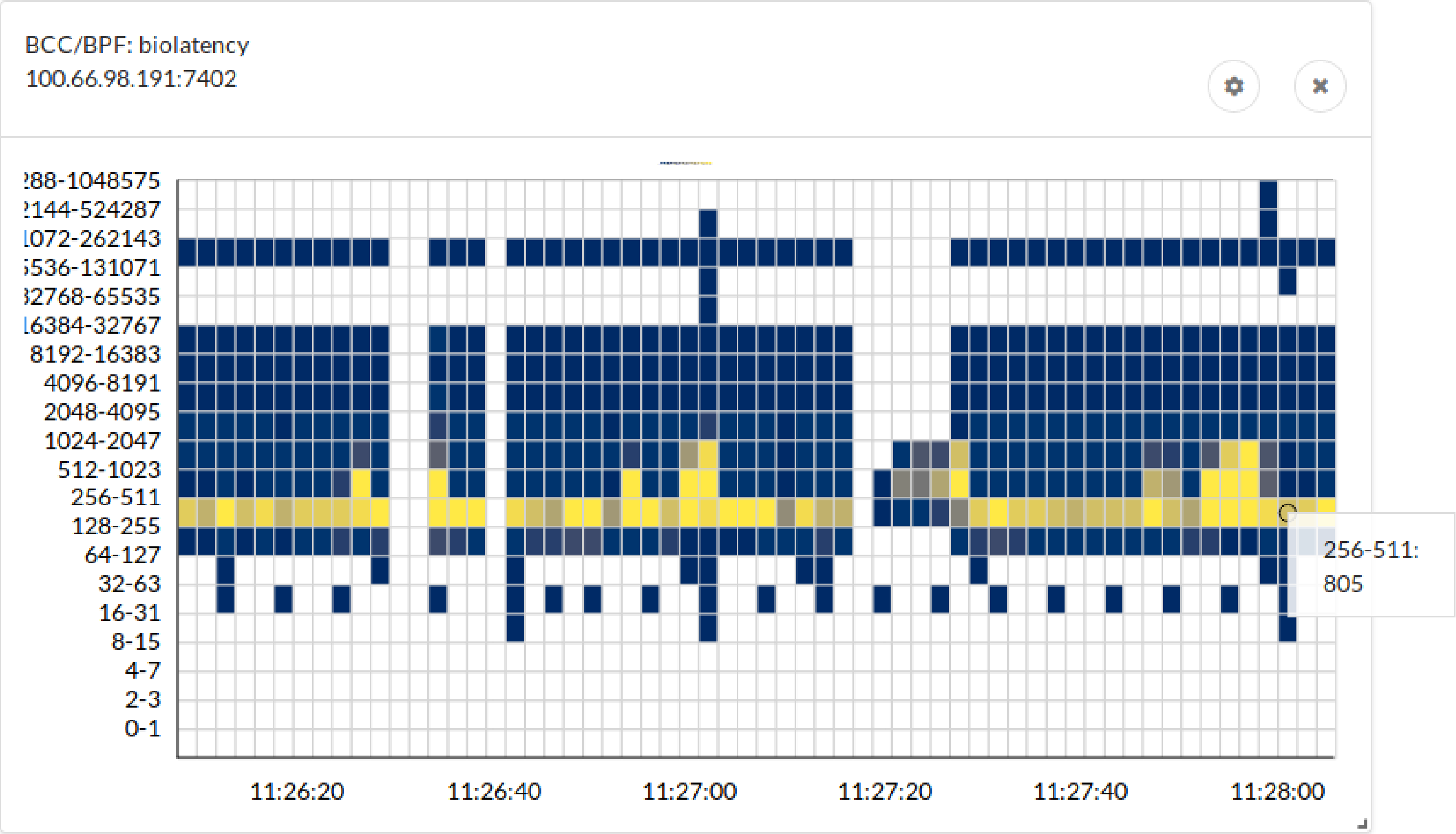
Why biolatency is efficient





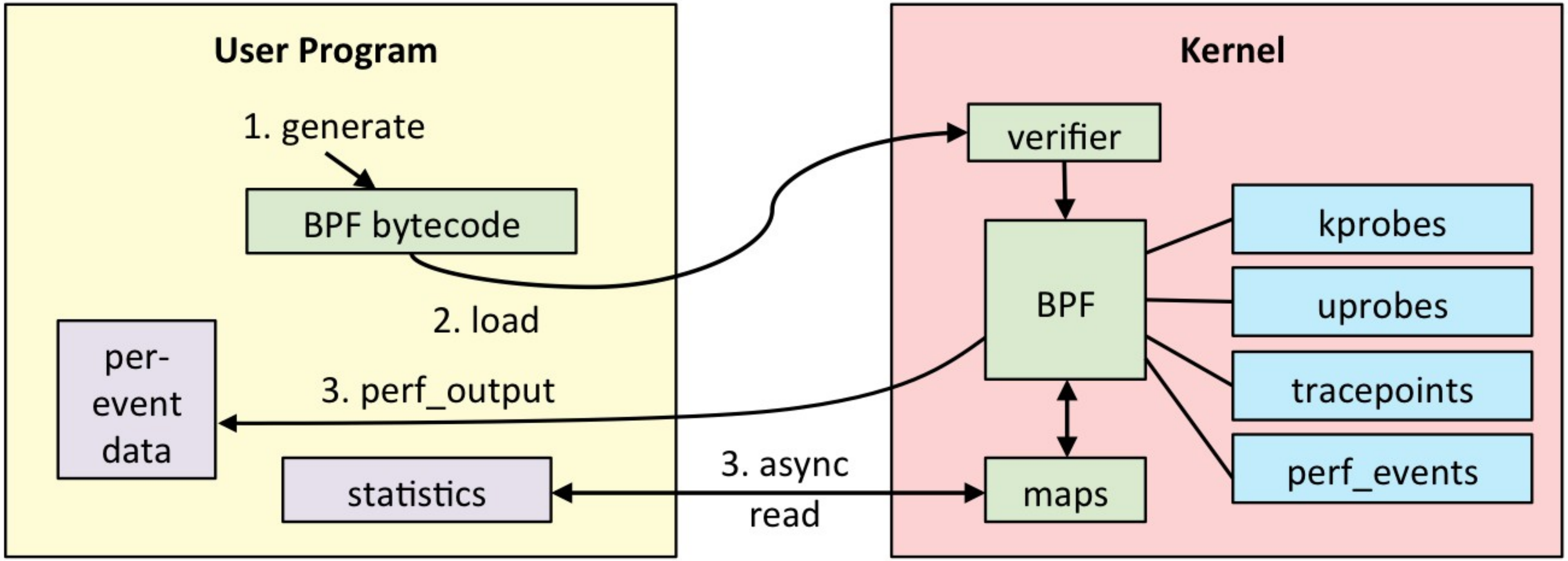
Lightweight. Practical.
Production safe.

Real-time custom histograms now practical



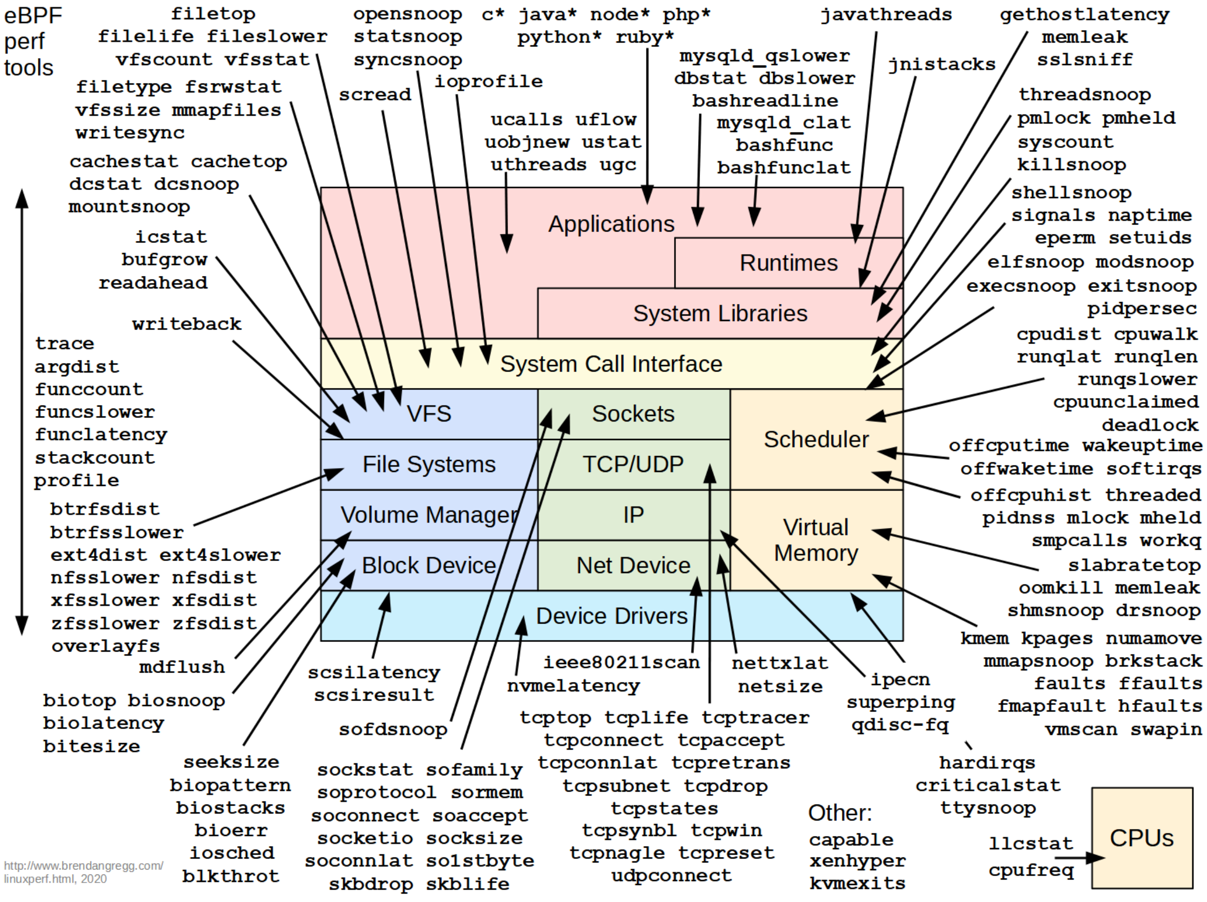
BCC/BPF: biolatency
100.66.98.191:7402

Two ways to return data to user-space



New tools more practical: Filling in blind spots

Whatever you can imagine!



eBPF perf tools

filetop
filelife fileslower
vfscount vfsstat
filetype fsrwstat
vfssize mmapfiles
writesync
cachestat cachetop
dcstat dcsnoop
mountsnoop
icstat
bufgrow
readahead
writeback

trace
argdist
funccount
funcslower
funclatency
stackcount
profile

btrfsdist
btrfsslower
ext4dist ext4slower
nfsslower nfsdist
xfsslower xfsdist
zfsslower zfsdist
overlayfs
mdflush

biotop biosnoop
biolatency
bitesize

seeksize
biopattern
biostacks
bioerr
iosched
blkthrot

opensnoop
statsnoop
syncsnoop
scread
ioprofile

c* java* node* php*
python* ruby*

ucalls uflow
uobjnew ustat
uthreads ugc

mysqld_qslower
dbstat dbslower
bashreadline
mysqld_clat
bashfunc
bashfunclat

javathreads
jnistacks

gethostlatency
memleak
sslsniff

threadsnoop
pmlock pmheld
syscount
killsnoop
shellsnoop
signals naptime
eperm setuids
elfsnoop modsnoop
execsnoop exitsnoop
pidpersec
cpudist cpuwalk
runqlat runqlen
runqslower
cpuunclaimed
deadlock
offcputime wakeuptime
offwaketime softirqs
offcpuhist threaded
pidnss mlock mheld
smpcalls workq
slabratetop
oomkill memleak
shmsnoop drsnoop
kmem kpages numamove
mmapsnoop brkstack
faults ffaults
fmapfault hfaults
vmscan swapin

Applications
Runtimes
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device
Net Device
Device Drivers

scsilatency
scsiresult
sofdsnoop

sockstat sofamily
soprotocol sormem
soconnect soaccept
socketio socksize
soconnlat so1stbyte
skbdrop skblife

ieee80211scan
nvmelatency
nettxlat
netsize

tcptop tcplife tcptracer
tcpconnect tcpaccept
tcpconnlat tcpretrans
tcpsubnet tcpdrop
tcpstates
tcpsynbl tcpwin
tcpnagle tcpreset
udpconnect

ipecn
superping
qdisc-fq

Other:
capable
xenhyper
kvmexits

hardirqs
criticalstat
ttysnoop

llcstat
cpufreq

CPUs

http://www.brendangregg.com/linuxperf.html, 2020

Agenda

1) What: A type of software
2) Why: Case Study
3) **How: History**, Internals, Usage
4) What's Next: Challenges, Future
5) Discussion & Q&A

(This history section focuses on "tracing": performance analysis using eBPF)

eBPF, The Early Years (2014-2017)

PLUMgrid

redhat

NETFLIX

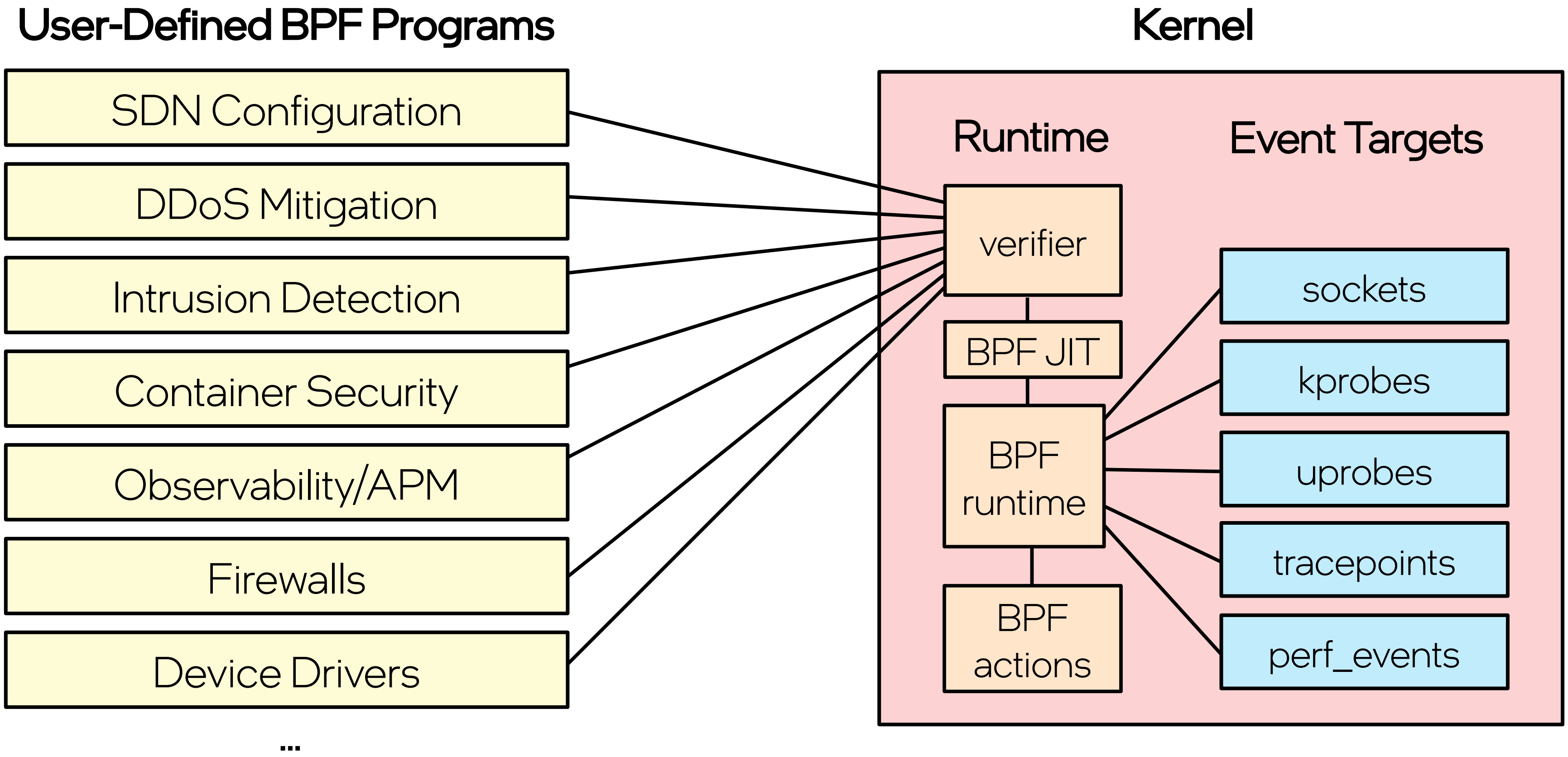
https://www.youtube.com/watch?v=Wb_vD3XZYOA

BPF 1992: Berkeley Packet Filter

```
# tcpdump -d host 127.0.0.1 and port 80
(000) ldh      [12]
(001) jeq      #0x800           jt 2   jf 18
(002) ld       [26]
(003) jeq      #0x7f000001      jt 6   jf 4
(004) ld       [30]
(005) jeq      #0x7f000001      jt 6   jf 18
(006) ldb      [23]
(007) jeq      #0x84            jt 10  jf 8
(008) jeq      #0x6             jt 10  jf 9
(009) jeq      #0x11            jt 10  jf 18
(010) ldh      [20]
(011) jset     #0x1fff          jt 18  jf 12
(012) ldxb     4*([14]&0xf)
(013) ldh      [x + 14]
(014) jeq      #0x50            jt 17  jf 15
(015) ldh      [x + 16]
(016) jeq      #0x50            jt 17  jf 18
(017) ret      #262144
(018) ret      #0
```

A limited
virtual machine for
efficient packet filters

eBPF 2014+


User-Defined BPF Programs
SDN Configuration
DDoS Mitigation
Intrusion Detection
Container Security
Observability/APM
Firewalls
Device Drivers
...
Kernel
Runtime
Event Targets
verifier
BPF JIT
BPF runtime
BPF actions
sockets
kprobes
uprobes
tracepoints
perf_events

Called eBPF at first, then:

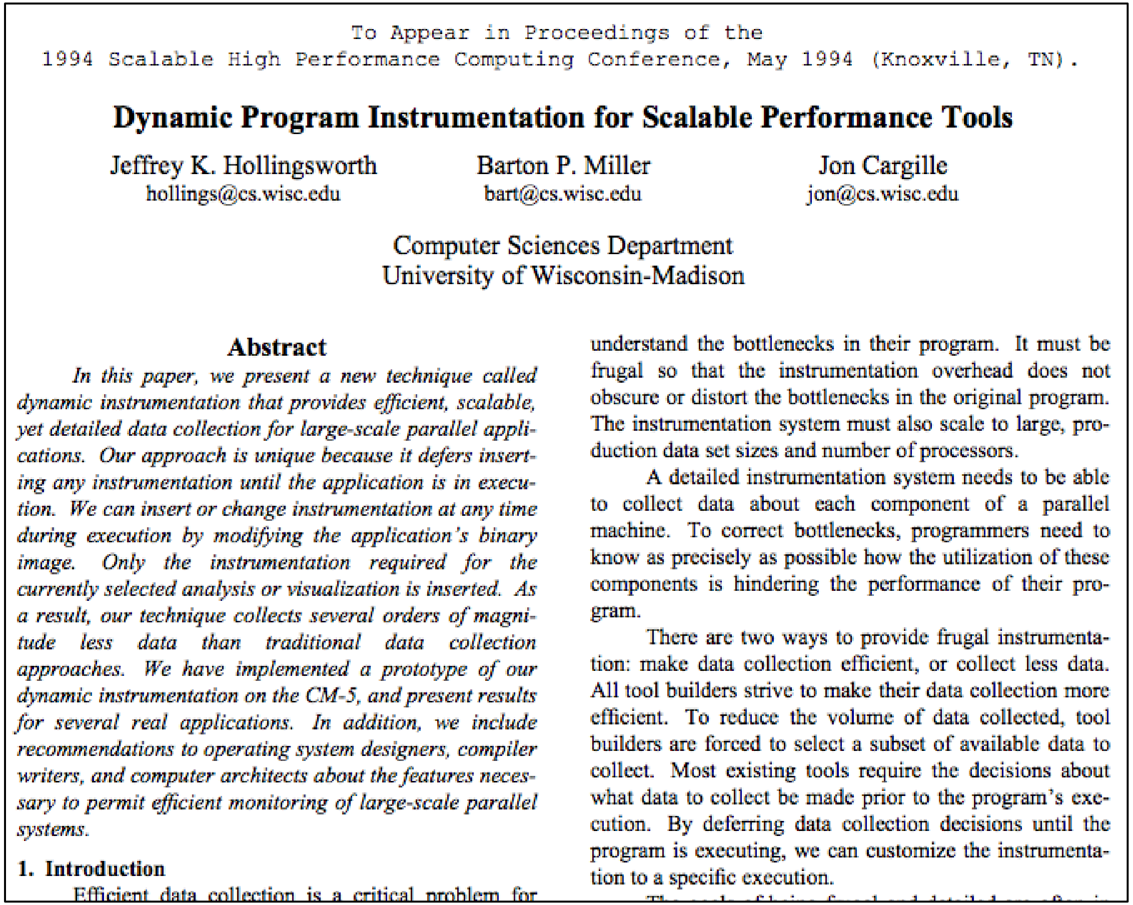
BPF (kernel engineers)

eBPF (marketing)

Modern logo

BPF is no longer an acronym

1994: Origin of Dynamic Instrumentation / Tracing



To Appear in Proceedings of the
1994 Scalable High Performance Computing Conference, May 1994 (Knoxville, TN).

Dynamic Program Instrumentation for Scalable Performance Tools

Jeffrey K. Hollingsworth
hollings@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Jon Cargille
jon@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison

Abstract

In this paper, we present a new technique called dynamic instrumentation that provides efficient, scalable, yet detailed data collection for large-scale parallel applications. Our approach is unique because it defers inserting any instrumentation until the application is in execution. We can insert or change instrumentation at any time during execution by modifying the application's binary image. Only the instrumentation required for the currently selected analysis or visualization is inserted. As a result, our technique collects several orders of magnitude less data than traditional data collection approaches. We have implemented a prototype of our dynamic instrumentation on the CM-5, and present results for several real applications. In addition, we include recommendations to operating system designers, compiler writers, and computer architects about the features necessary to permit efficient monitoring of large-scale parallel systems.

1. Introduction

Efficient data collection is a critical problem for

understand the bottlenecks in their program. It must be frugal so that the instrumentation overhead does not obscure or distort the bottlenecks in the original program. The instrumentation system must also scale to large, production data set sizes and number of processors.

A detailed instrumentation system needs to be able to collect data about each component of a parallel machine. To correct bottlenecks, programmers need to know as precisely as possible how the utilization of these components is hindering the performance of their program.

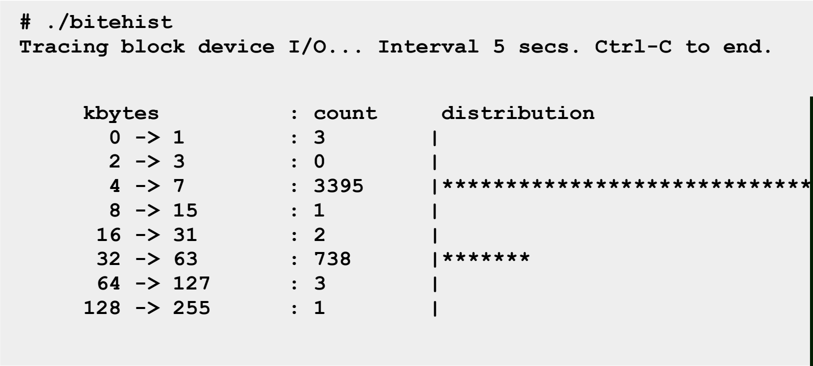
There are two ways to provide frugal instrumentation: make data collection efficient, or collect less data. All tool builders strive to make their data collection more efficient. To reduce the volume of data collected, tool builders are forced to select a subset of available data to collect. Most existing tools require the decisions about what data to collect be made prior to the program's execution. By deferring data collection decisions until the program is executing, we can customize the instrumentation to a specific execution.

Source: https://www.cs.umd.edu/users/hollings/papers/shpcc94.pdf

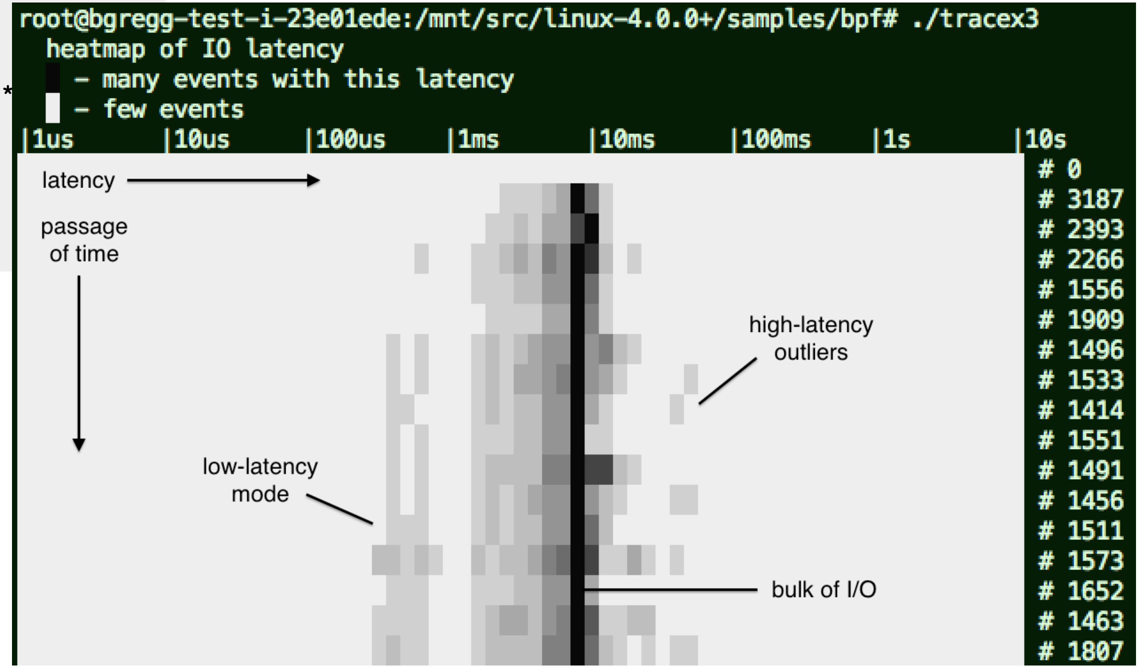
<2014: Linux Tracing was a mess (with cute ponies)

The tracing ponies were created by a marketing professional from Sun Microsystems, Deirdre Straughan, now my wife

eBPF 2015: First perf/tracing tool use case



```
# ./bitehist
Tracing block device I/O... Interval 5 secs. Ctrl-C to end.

     kbytes          : count     distribution
       0 -> 1        : 3        |
       2 -> 3        : 0        |
       4 -> 7        : 3395     |*****************************
       8 -> 15       : 1        |
      16 -> 31       : 2        |
      32 -> 63       : 738      |*******
      64 -> 127      : 3        |
     128 -> 255      : 1        |
```



```
root@bgregg-test-i-23e01ede:/mnt/src/linux-4.0.0+/samples/bpf# ./tracex3
  heatmap of IO latency
  ■ - many events with this latency
  □ - few events
|1us      |10us     |100us    |1ms      |10ms     |100ms    |1s       |10s
                                                                     # 0
                                                                     # 3187
                                                                     # 2393
                                                                     # 2266
                                                                     # 1556
                                                                     # 1909
                                                                     # 1496
                                                                     # 1533
                                                                     # 1414
                                                                     # 1551
                                                                     # 1491
                                                                     # 1456
                                                                     # 1511
                                                                     # 1573
                                                                     # 1652
                                                                     # 1463
                                                                     # 1807
```

Dynamic instrumentation of block I/O functions, custom timing, and custom in-kernel histograms.

https://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html

2015-2016+: New BPF-based tracers

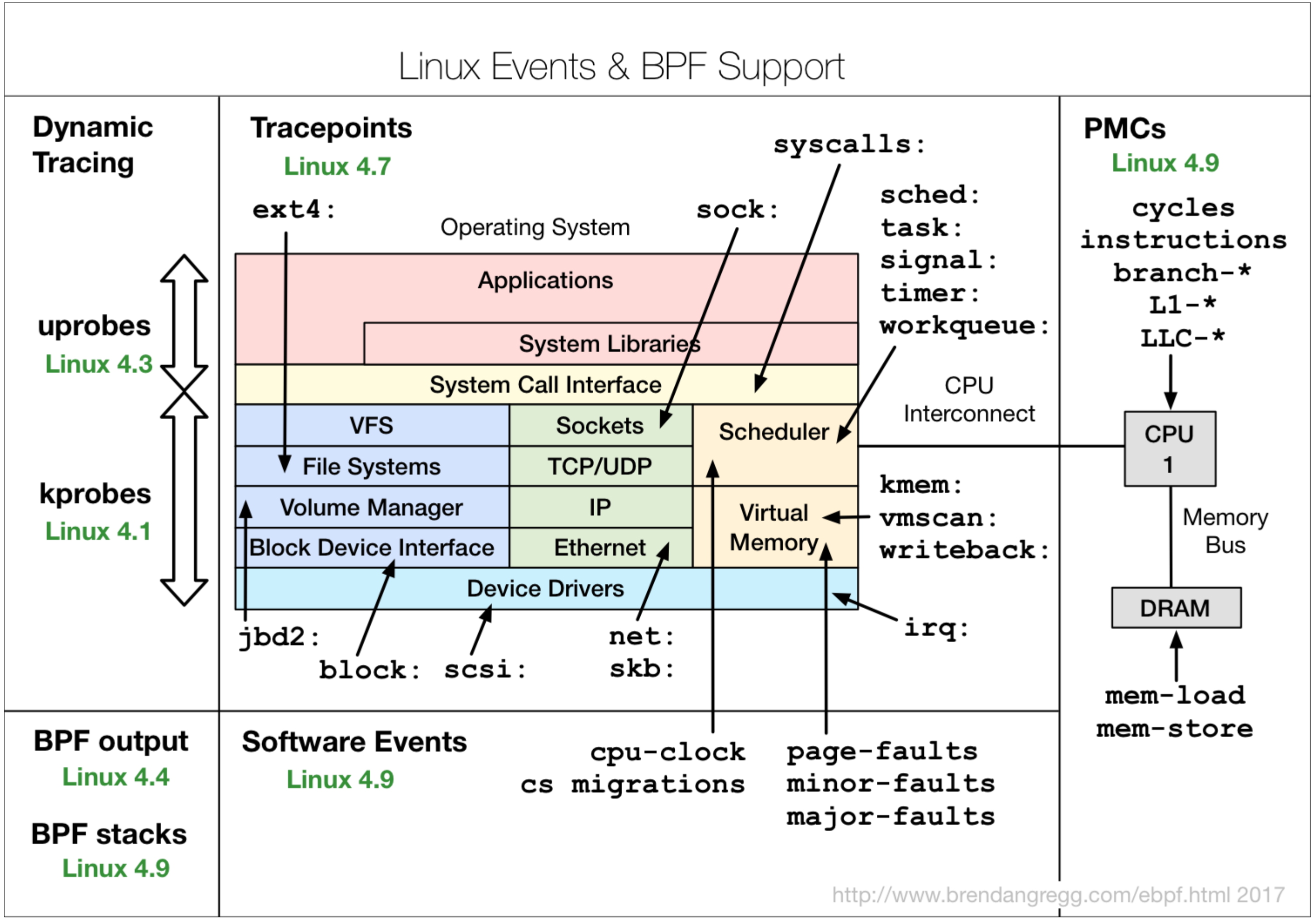
bcc

bpftrace

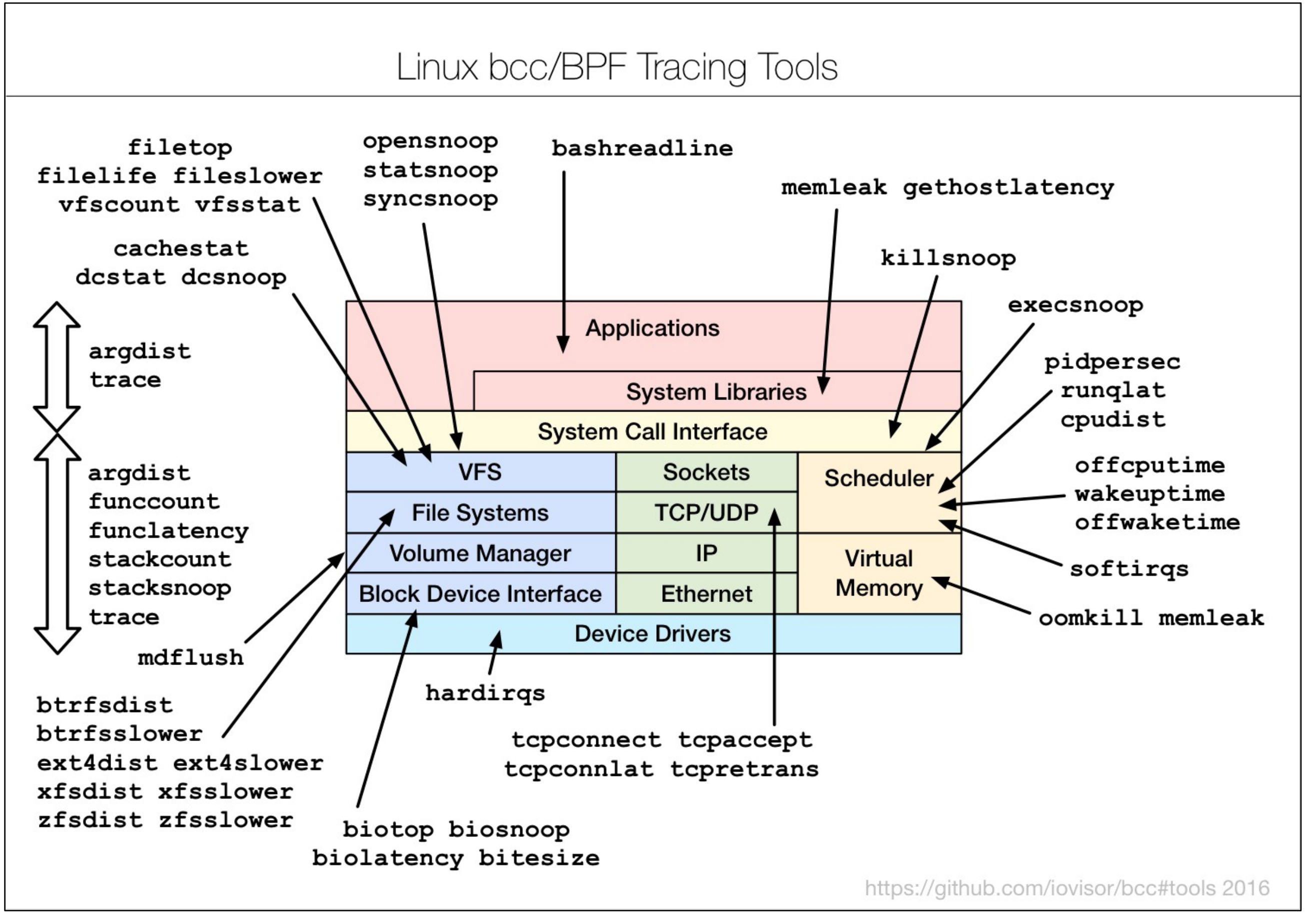
2015-2016: More event sources

Instrument anything, safely, in production.

I started calling eBPF "**superpowers**."



2016: New perf tools using bcc



Linux bcc/BPF Tracing Tools

filetop
filelife fileslower
vfscount vfsstat

cachestat
dcstat dcsnoop

opensnoop
statsnoop
syncsnoop

bashreadline

memleak gethostlatency

killsnoop

execsnoop

pidpersec
runqlat
cpudist

offcputime
wakeuptime
offwaketime

softirqs

oomkill memleak

argdist
trace

argdist
funccount
funclatency
stackcount
stacksnoop
trace

mdflush

btrfsdist
btrfsslower
ext4dist ext4slower
xfsdist xfsslower
zfsdist zfsslower

hardirqs

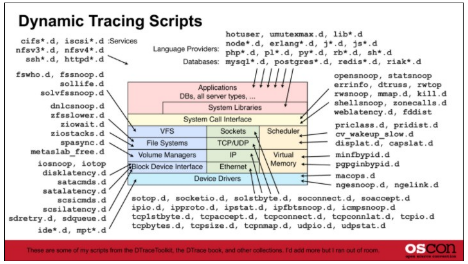
tcpconnect tcpaccept
tcpconnlat tcpretrans

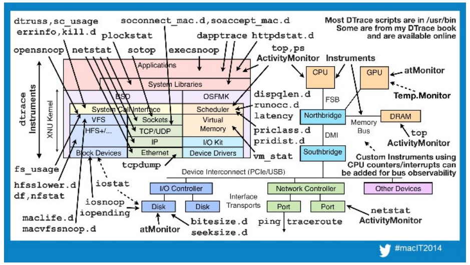
biotop biosnoop
biolatency bitesize

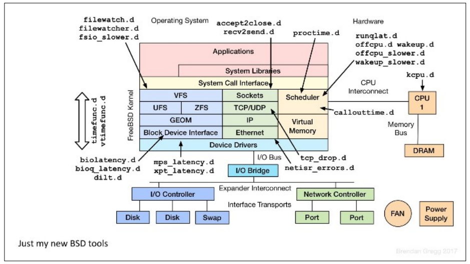
Applications
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device Interface
Ethernet
Device Drivers

https://github.com/iovisor/bcc#tools 2016

I'd done this a few times before,
Linux was my 4th kernel:





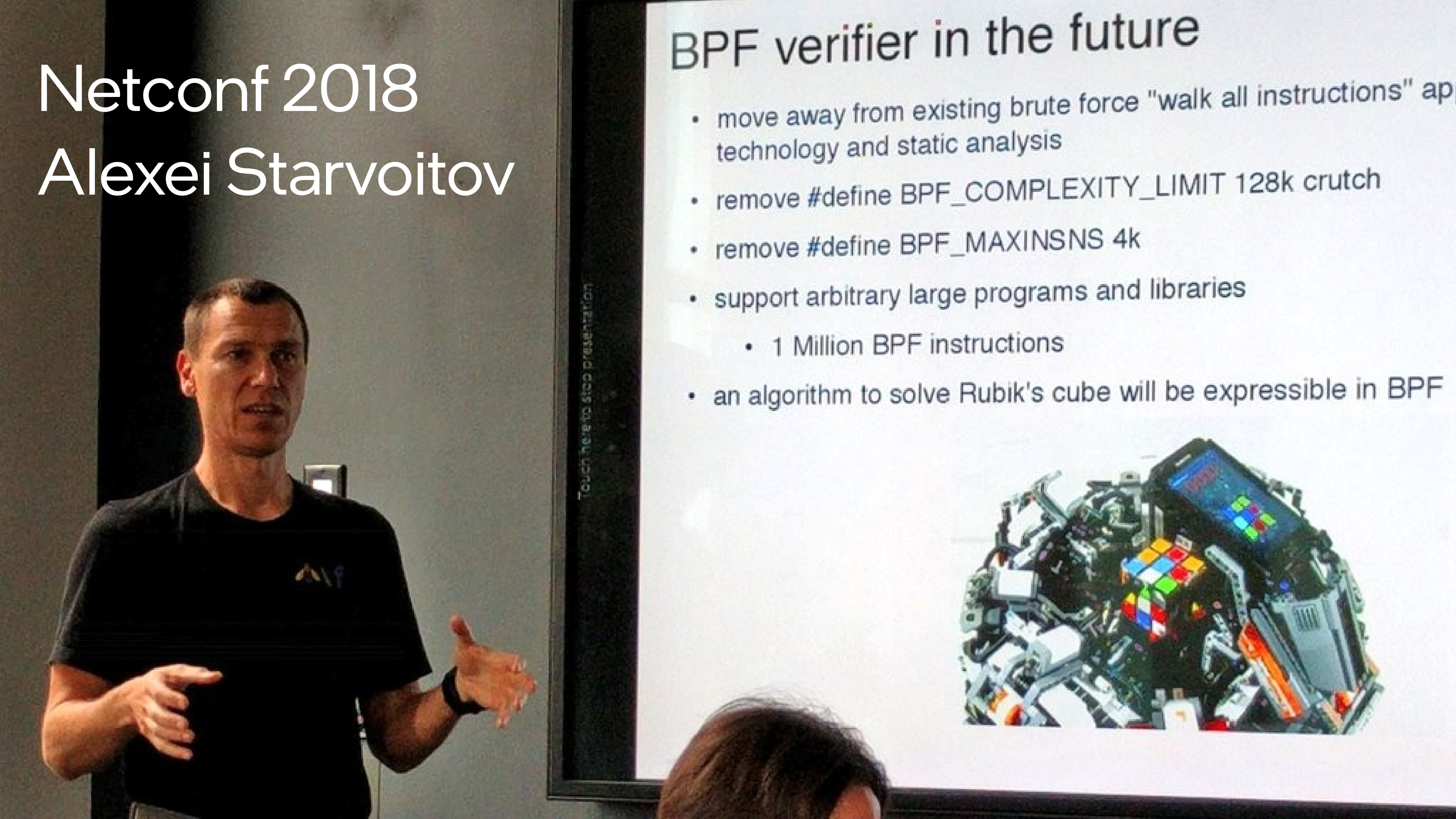


DockerCon 2017: BPF goes big

Meanwhile, I would sometimes hit instruction limits needing workarounds

Sessions Repeated Tomorrow
EXPO HALL 1
9:00 – Cilium: Network and Application Security with BPF and XDP by Thomas Graf
9:50 – Docker?!? But I am a Sysadmin by Mike Coleman
10:40
BALLROOM A
9:00 – Secure Substrate: L
Privilege Container Deplo
Diogo Mónica and Riyaz F
9:50 – Escape from VMs
Image2Docker by Elton S
Jeff koloff
hat Have Name
tely? by Liz R
eating Effectiv
dockercon17
NETFLIX

Netconf 2018
Alexei Starvoitov



BPF verifier in the future

- move away from existing brute force "walk all instructions" ap technology and static analysis
- remove #define BPF_COMPLEXITY_LIMIT 128k crutch
- remove #define BPF_MAXINSNS 4k
- support arbitrary large programs and libraries
  - 1 Million BPF instructions
- an algorithm to solve Rubik's cube will be expressible in BPF

2018-2019: bpftrace brings ease of use

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

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

tracepoint:block:block_bio_queue
{
        @start[args.sector] = nsecs;
}

tracepoint:block:block_rq_complete,
tracepoint:block:block_bio_complete
/@start[args.sector]/
{
        @usecs = hist((nsecs - @start[args.sector]) / 1000);
        delete(@start[args.sector]);
}

END
{
        clear(@start);
}
```

https://github.com/bpftrace/bpftrace/blob/master/tools/biolatency.bt

ply was and is another good emerging option

BPF at Facebook

- ~40 BPF programs active on every server.
- ~100 BPF programs loaded on demand for short period of time.
- Mainly used by daemons that run on every server.
- Many teams are writing and deploying them.

8

ftrace: Where modifying a running ke
Analyzing changes to the binary inte
BPF at Facebook - Alexei Starovoitov

Kernel Recipes 2019, Alexei Starovoitov

~40 active BPF programs on every Facebook server

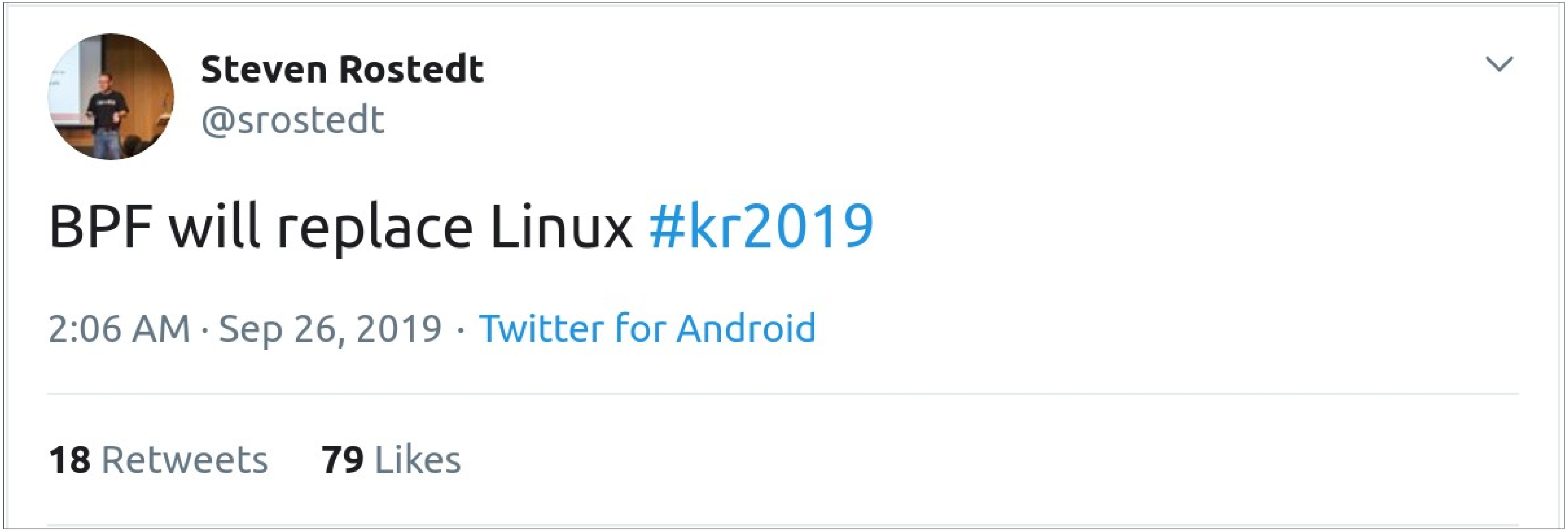
Extended BPF
A New Type of Software

Brendan Gregg

UbuntuMasters

UbuntuMasters 2019, Brendan Gregg

~14 active BPF programs on every Netflix cloud instance



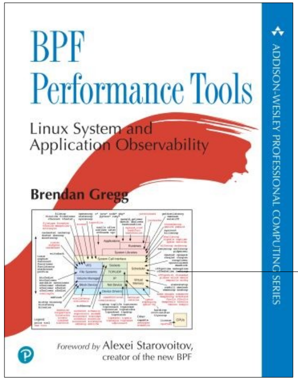
Steven Rostedt
@srostedt

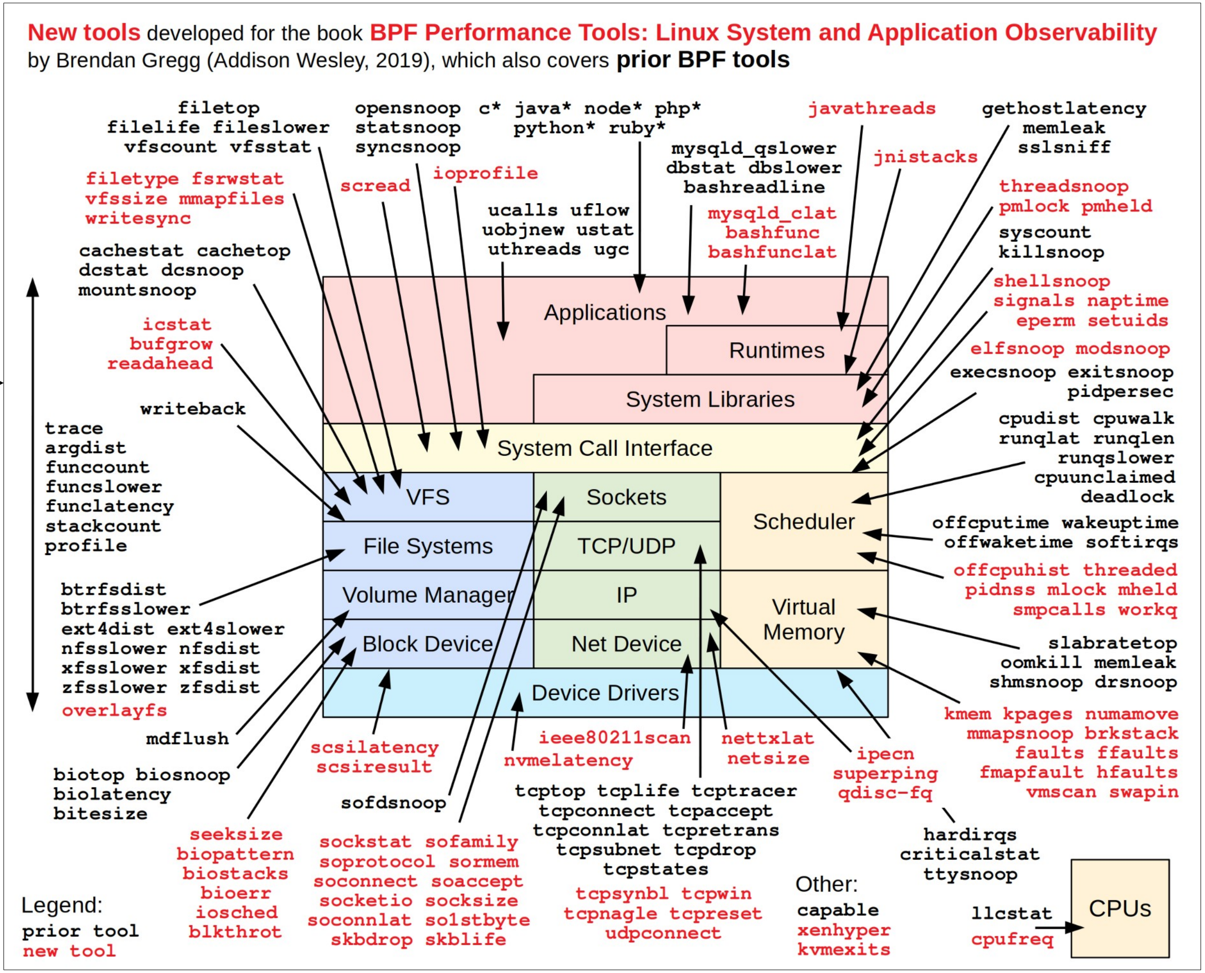
BPF will replace Linux #kr2019

2:06 AM · Sep 26, 2019 · Twitter for Android

18 Retweets **79** Likes

2019: BPF Perf Tools book, lots more tools


BPF
Performance Tools
Linux System and Application Observability
Brendan Gregg
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Foreword by Alexei Starovoitov, creator of the new BPF



New tools developed for the book **BPF Performance Tools: Linux System and Application Observability** by Brendan Gregg (Addison Wesley, 2019), which also covers **prior BPF tools**

Applications
Runtimes
System Libraries
System Call Interface
VFS
File Systems
Volume Manager
Block Device
Sockets
TCP/UDP
IP
Net Device
Scheduler
Virtual Memory
Device Drivers
CPUs

filetop
filelife fileslower
vfscount vfsstat

filetype fsrwstat
vfssize mmapfiles
writesync

cachestat cachetop
dcstat dcsnoop
mountsnoop

icstat
bufgrow
readahead

writeback

trace
argdist
funccount
funcslower
funclatency
stackcount
profile

btrfsdist
btrfsslower
ext4dist ext4slower
nfsslower nfsdist
xfsslower xfsdist
zfsslower zfsdist
overlayfs

mdflush

biotop biosnoop
biolatency
bitesize

seeksize
biopattern
biostacks
bioerr
iosched
blkthrot

opensnoop
statsnoop
syncsnoop

scread

ioprofile

c* java* node* php*
python* ruby*

ucalls uflow
uobjnew ustat
uthreads ugc

mysqld_qslower
dbstat dbslower
bashreadline

mysqld_clat
bashfunc
bashfunclat

javathreads

jnistacks

gethostlatency
memleak
sslsniff

threadsnoop
pmlock pmheld

syscount
killsnoop

shellsnoop
signals naptime
eperm setuids

elfsnoop modsnoop

execsnoop exitsnoop
pidpersec

cpudist cpuwalk
runqlat runqlen
runqslower
cpuunclaimed
deadlock

offcputime wakeuptime
offwaketime softirqs

offcpuhist threaded
pidnss mlock mheld
smpcalls workq

slabratetop
oomkill memleak
shmsnoop drsnoop

kmem kpages numamove
mmapsnoop brkstack
faults ffaults
fmapfault hfaults
vmscan swapin

scsilatency
scsiresult

sofdsnoop

sockstat sofamily
soprotocol sormem
soconnect soaccept
socketio socksize
soconnlat so1stbyte
skbdrop skblife

ieee80211scan
nvmelatency

nettxlat
netsize

ipecn
superping
qdisc-fq

tcptop tcplife tcptracer
tcpconnect tcpaccept
tcpconnlat tcpretrans
tcpsubnet tcpdrop
tcpstates

tcpsynbl tcpwin
tcpnagle tcpreset
udpconnect

hardirqs
criticalstat
ttysnoop

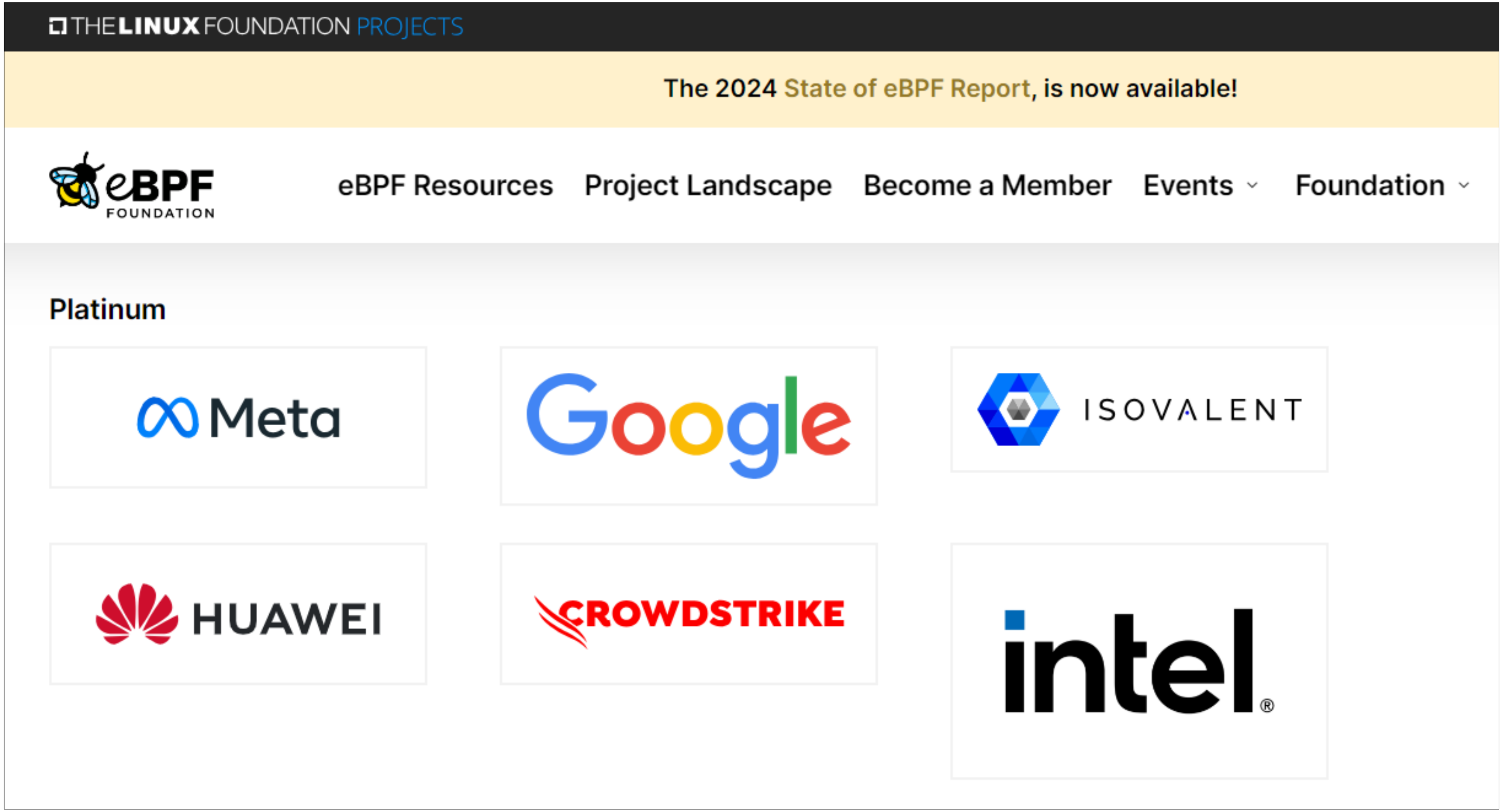
Other:
capable
xenhyper
kvmexits

llcstat
cpufreq

Legend:
prior tool
new tool

Many only possible thanks to dynamic instrumentation

2021: eBPF Foundation



THE LINUX FOUNDATION PROJECTS

The 2024 State of eBPF Report, is now available!

Platinum

https://ebpf.foundation/

2021+: BPF Technical Steering Committee (BSC)

Alexei Starovoitov (Meta)*

Daniel Borkmann (Isovalent/Cisco)*

Alan Jowett (Microsoft)

Andrii Nakryiko (Google)

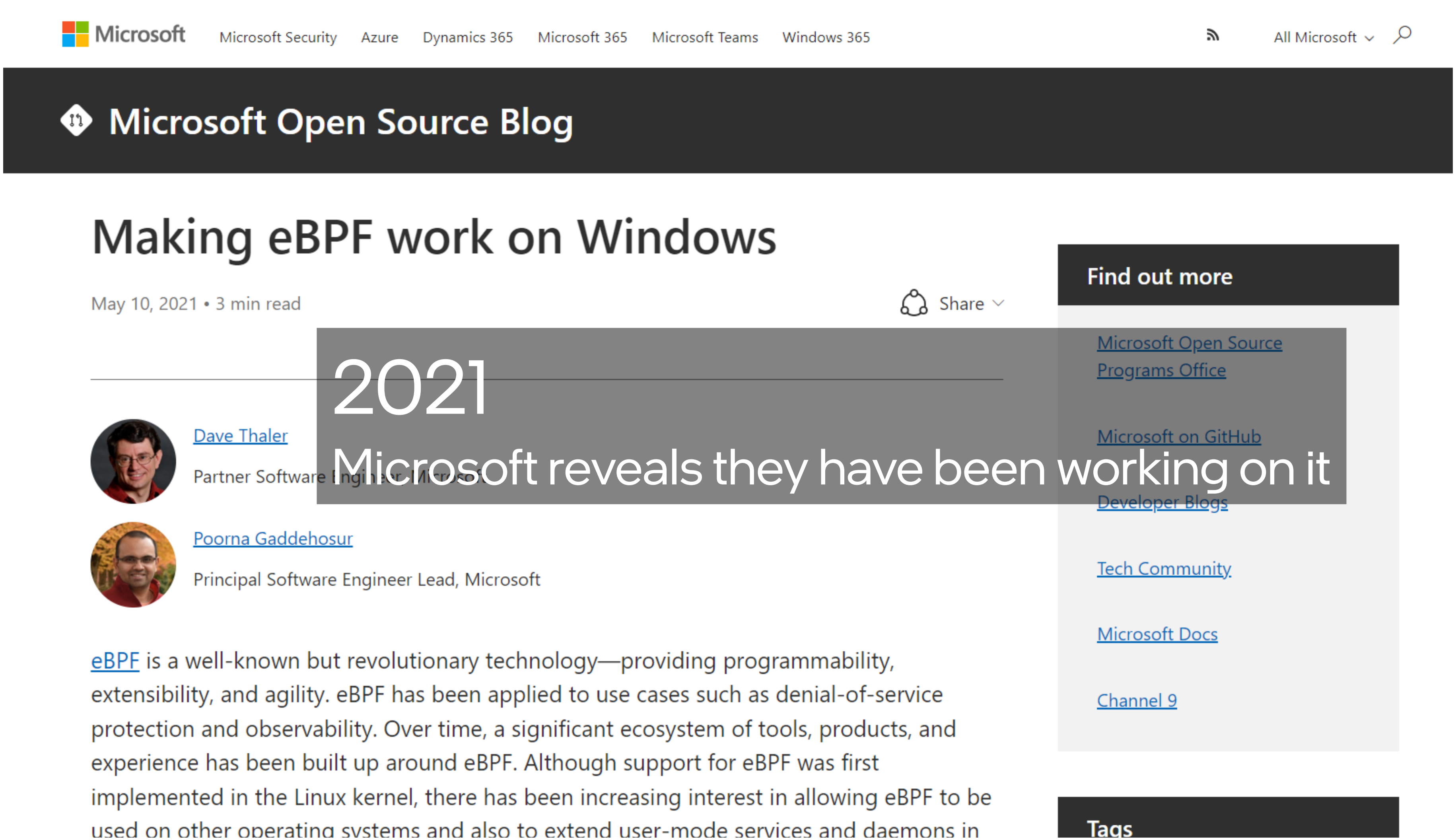
Brendan Gregg (Intel)

KP Singh (Google)

Joe Stringer (Isovalent/Cisco)

* Linux eBPF Maintaners

Perhaps the most crucial role of the BSC is to negotiate bytecode changes between implementations (Linux, Windows).
However, so far there have been no bytecode disagreements, and I'm not expecting any.



Making eBPF work on Windows

May 10, 2021 • 3 min read

Dave Thaler

Partner Software Engineer, Microsoft

Poorna Gaddehosur

Principal Software Engineer Lead, Microsoft

eBPF is a well-known but revolutionary technology—providing programmability, extensibility, and agility. eBPF has been applied to use cases such as denial-of-service protection and observability. Over time, a significant ecosystem of tools, products, and experience has been built up around eBPF. Although support for eBPF was first implemented in the Linux kernel, there has been increasing interest in allowing eBPF to be used on other operating systems and also to extend user-mode services and daemons in

Find out more

- Microsoft Open Source Programs Office
- Microsoft on GitHub
- Developer Blogs
- Tech Community
- Microsoft Docs
- Channel 9

Tags

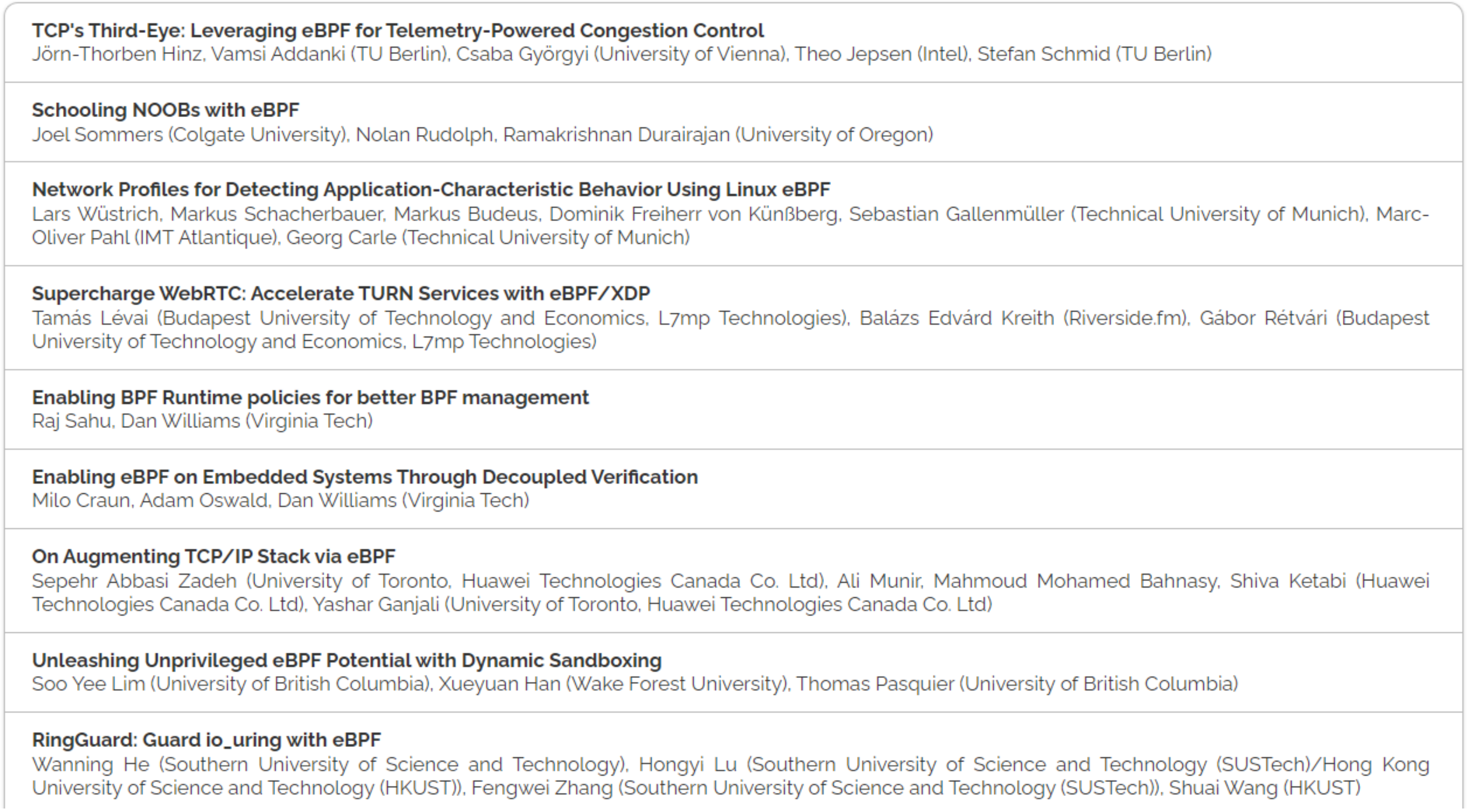
2021

Microsoft reveals they have been working on it

https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/

2021-2023: Powering new research and innovation

SIGCOMM 2023 New York City



TCP's Third-Eye: Leveraging eBPF for Telemetry-Powered Congestion Control
Jörn-Thorben Hinz, Vamsi Addanki (TU Berlin), Csaba Györgyi (University of Vienna), Theo Jepsen (Intel), Stefan Schmid (TU Berlin)

Schooling NOOBs with eBPF
Joel Sommers (Colgate University), Nolan Rudolph, Ramakrishnan Durairajan (University of Oregon)

Network Profiles for Detecting Application-Characteristic Behavior Using Linux eBPF
Lars Wüstrich, Markus Schacherbauer, Markus Budeus, Dominik Freiherr von Künßberg, Sebastian Gallenmüller (Technical University of Munich), Marc-Oliver Pahl (IMT Atlantique), Georg Carle (Technical University of Munich)

Supercharge WebRTC: Accelerate TURN Services with eBPF/XDP
Tamás Lévai (Budapest University of Technology and Economics, L7mp Technologies), Balázs Edvárd Kreith (Riverside.fm), Gábor Rétvári (Budapest University of Technology and Economics, L7mp Technologies)

Enabling BPF Runtime policies for better BPF management
Raj Sahu, Dan Williams (Virginia Tech)

Enabling eBPF on Embedded Systems Through Decoupled Verification
Milo Craun, Adam Oswald, Dan Williams (Virginia Tech)

On Augmenting TCP/IP Stack via eBPF
Sepehr Abbasi Zadeh (University of Toronto, Huawei Technologies Canada Co. Ltd), Ali Munir, Mahmoud Mohamed Bahnasy, Shiva Ketabi (Huawei Technologies Canada Co. Ltd), Yashar Ganjali (University of Toronto, Huawei Technologies Canada Co. Ltd)

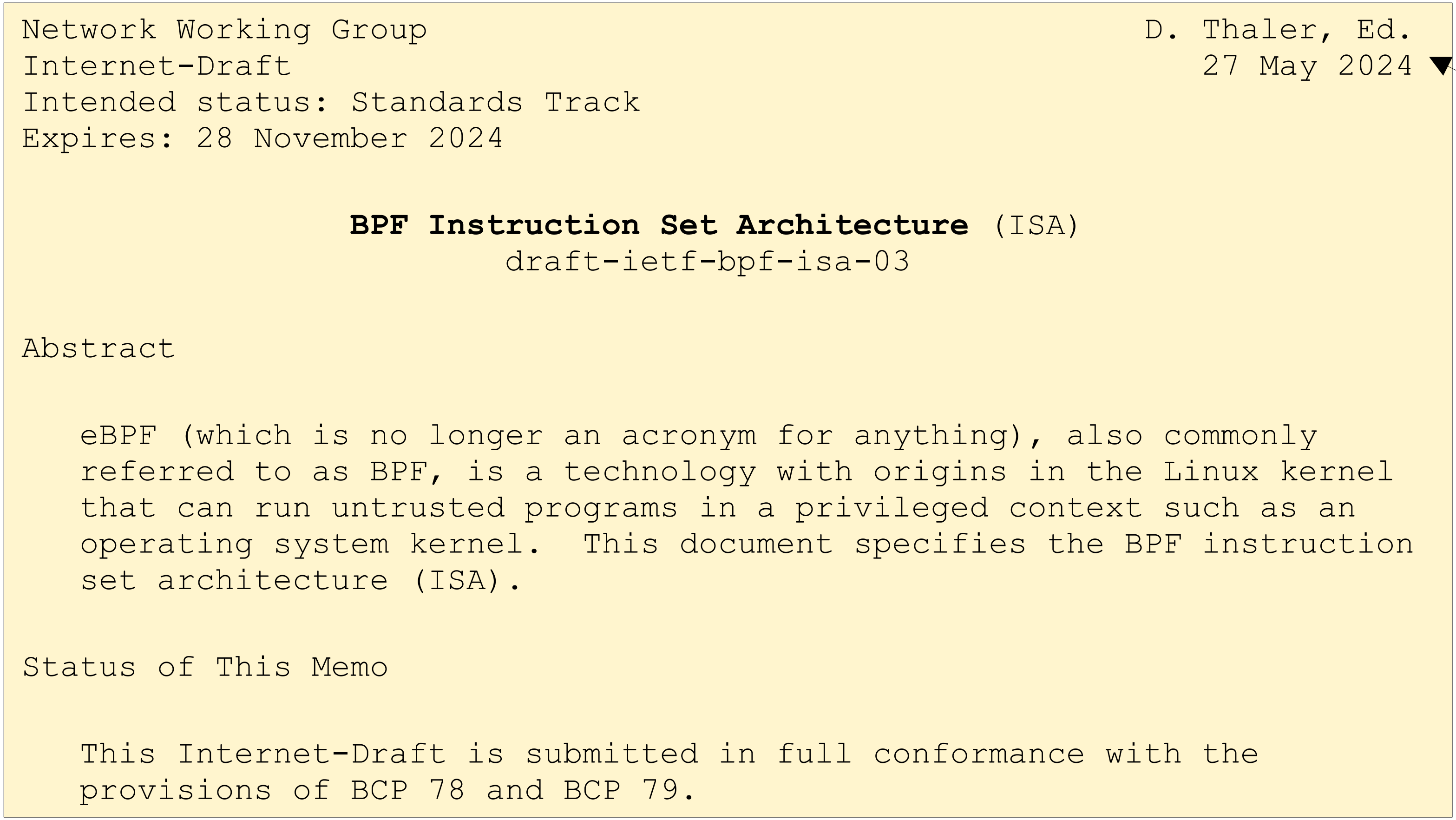
Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing
Soo Yee Lim (University of British Columbia), Xueyuan Han (Wake Forest University), Thomas Pasquier (University of British Columbia)

RingGuard: Guard io_uring with eBPF
Wanning He (Southern University of Science and Technology), Hongyi Lu (Southern University of Science and Technology (SUSTech)/Hong Kong University of Science and Technology (HKUST)), Fengwei Zhang (Southern University of Science and Technology (SUSTech)), Shuai Wang (HKUST)

...

https://conferences.sigcomm.org/sigcomm/2023/workshop-ebpf.html

2024: IETF standard draft

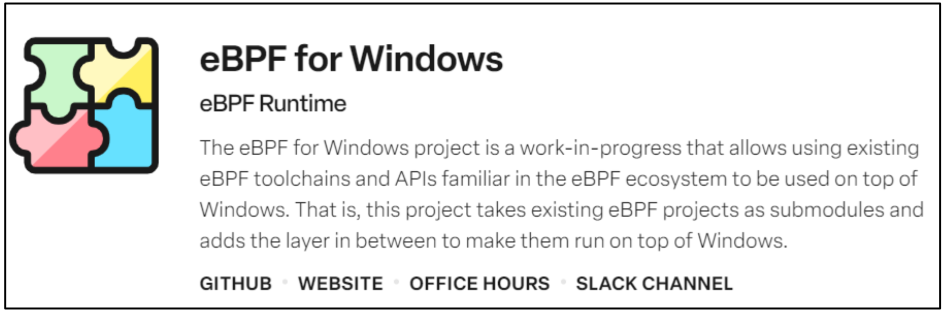


```
Network Working Group                                   D. Thaler, Ed.
Internet-Draft                                             27 May 2024
Intended status: Standards Track
Expires: 28 November 2024


                 BPF Instruction Set Architecture (ISA)
                         draft-ietf-bpf-isa-03

Abstract

   eBPF (which is no longer an acronym for anything), also commonly
   referred to as BPF, is a technology with origins in the Linux kernel
   that can run untrusted programs in a privileged context such as an
   operating system kernel.  This document specifies the BPF instruction
   set architecture (ISA).

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.
```

Driven by Dave Thaler (Microsoft) while he was on the BSC.
Thanks, Dave!

https://datatracker.ietf.org/doc/draft-ietf-bpf-isa/

2024: Major Projects Include

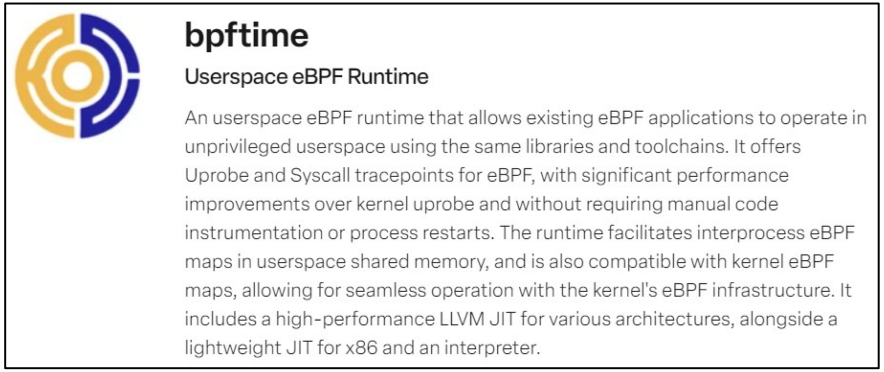


eBPF for Windows

eBPF Runtime

The eBPF for Windows project is a work-in-progress that allows using existing eBPF toolchains and APIs familiar in the eBPF ecosystem to be used on top of Windows. That is, this project takes existing eBPF projects as submodules and adds the layer in between to make them run on top of Windows.

GITHUB · WEBSITE · OFFICE HOURS · SLACK CHANNEL

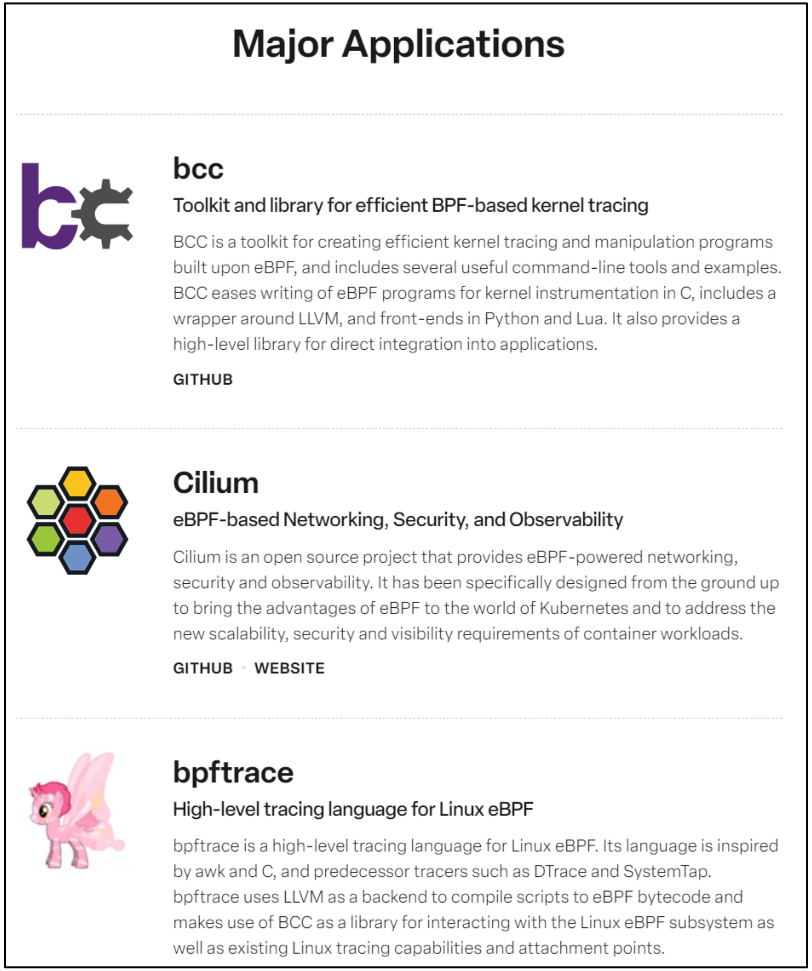


bpftime

Userspace eBPF Runtime

An userspace eBPF runtime that allows existing eBPF applications to operate in unprivileged userspace using the same libraries and toolchains. It offers Uprobe and Syscall tracepoints for eBPF, with significant performance improvements over kernel uprobe and without requiring manual code instrumentation or process restarts. The runtime facilitates interprocess eBPF maps in userspace shared memory, and is also compatible with kernel eBPF maps, allowing for seamless operation with the kernel's eBPF infrastructure. It includes a high-performance LLVM JIT for various architectures, alongside a lightweight JIT for x86 and an interpreter.

Source: https://ebpf.io/infrastructure/



Major Applications

bcc

Toolkit and library for efficient BPF-based kernel tracing

BCC is a toolkit for creating efficient kernel tracing and manipulation programs built upon eBPF, and includes several useful command-line tools and examples. BCC eases writing of eBPF programs for kernel instrumentation in C, includes a wrapper around LLVM, and front-ends in Python and Lua. It also provides a high-level library for direct integration into applications.

GITHUB

Cilium

eBPF-based Networking, Security, and Observability

Cilium is an open source project that provides eBPF-powered networking, security and observability. It has been specifically designed from the ground up to bring the advantages of eBPF to the world of Kubernetes and to address the new scalability, security and visibility requirements of container workloads.

GITHUB · WEBSITE

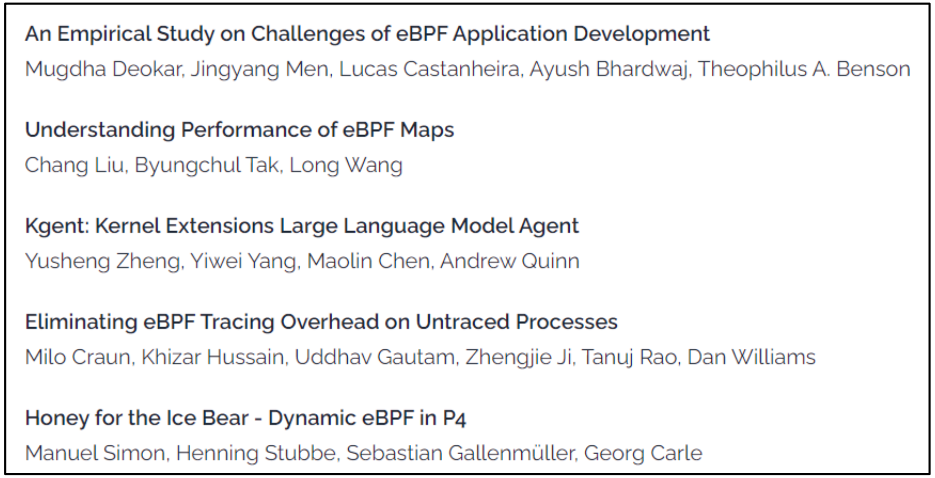
bpftrace

High-level tracing language for Linux eBPF

bpftrace is a high-level tracing language for Linux eBPF. Its language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap. bpftrace uses LLVM as a backend to compile scripts to eBPF bytecode and makes use of BCC as a library for interacting with the Linux eBPF subsystem as well as existing Linux tracing capabilities and attachment points.

Source: https://ebpf.io/applications/

2024+: More innovation



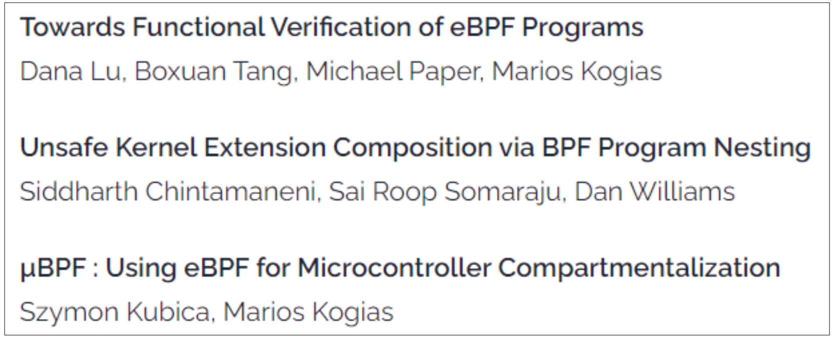
An Empirical Study on Challenges of eBPF Application Development
Mugdha Deokar, Jingyang Men, Lucas Castanheira, Ayush Bhardwaj, Theophilus A. Benson

Understanding Performance of eBPF Maps
Chang Liu, Byungchul Tak, Long Wang

Kgent: Kernel Extensions Large Language Model Agent
Yusheng Zheng, Yiwei Yang, Maolin Chen, Andrew Quinn

Eliminating eBPF Tracing Overhead on Untraced Processes
Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, Dan Williams

Honey for the Ice Bear - Dynamic eBPF in P4
Manuel Simon, Henning Stubbe, Sebastian Gallenmüller, Georg Carle

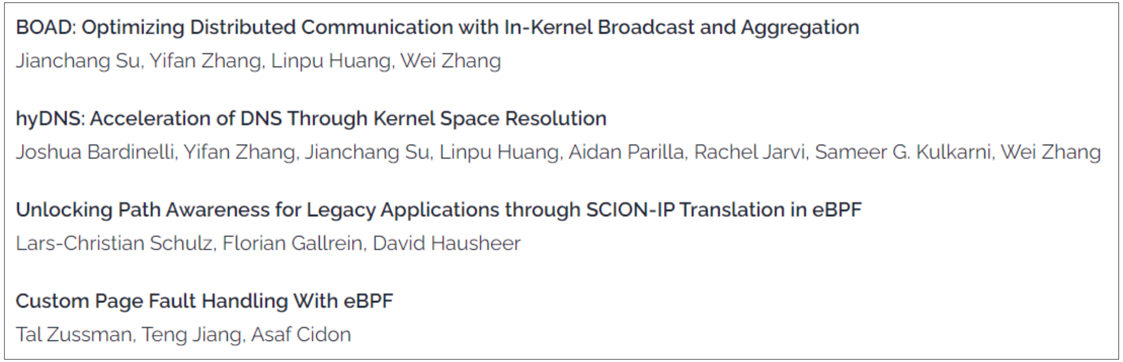


Towards Functional Verification of eBPF Programs
Dana Lu, Boxuan Tang, Michael Paper, Marios Kogias

Unsafe Kernel Extension Composition via BPF Program Nesting
Siddharth Chintamaneni, Sai Roop Somaraju, Dan Williams

μBPF : Using eBPF for Microcontroller Compartmentalization
Szymon Kubica, Marios Kogias

SIGCOMM 2024
— SYDNEY —



BOAD: Optimizing Distributed Communication with In-Kernel Broadcast and Aggregation
Jianchang Su, Yifan Zhang, Linpu Huang, Wei Zhang

hyDNS: Acceleration of DNS Through Kernel Space Resolution
Joshua Bardinelli, Yifan Zhang, Jianchang Su, Linpu Huang, Aidan Parilla, Rachel Jarvi, Sameer G. Kulkarni, Wei Zhang

Unlocking Path Awareness for Legacy Applications through SCION-IP Translation in eBPF
Lars-Christian Schulz, Florian Gallrein, David Hausheer

Custom Page Fault Handling With eBPF
Tal Zussman, Teng Jiang, Asaf Cidon

https://conferences.sigcomm.org/sigcomm/2024/workshop/ebpf/

2024 Reality

2015: I knew of every company, person, and significant development.

2024: eBPF is too big. I don't even know all the companies.

eBPF

Agenda

1) What: A type of software
2) Why: Case Study
3) **How:** History, **Internals**, Usage
4) What's Next: Challenges, Future
5) Discussion & Q&A

Some updated highlights for my tracing internals talk

BPF Internals

(eBPF)

Tracing Examples

Brendan Gregg

USENIX

LISA21

Jun, 2021

NETFLIX

https://www.brendangregg.com/blog/2021-06-15/bpf-internals.html

(Dan used a couple of these slides for SIGCOMM 2023)

eBPF

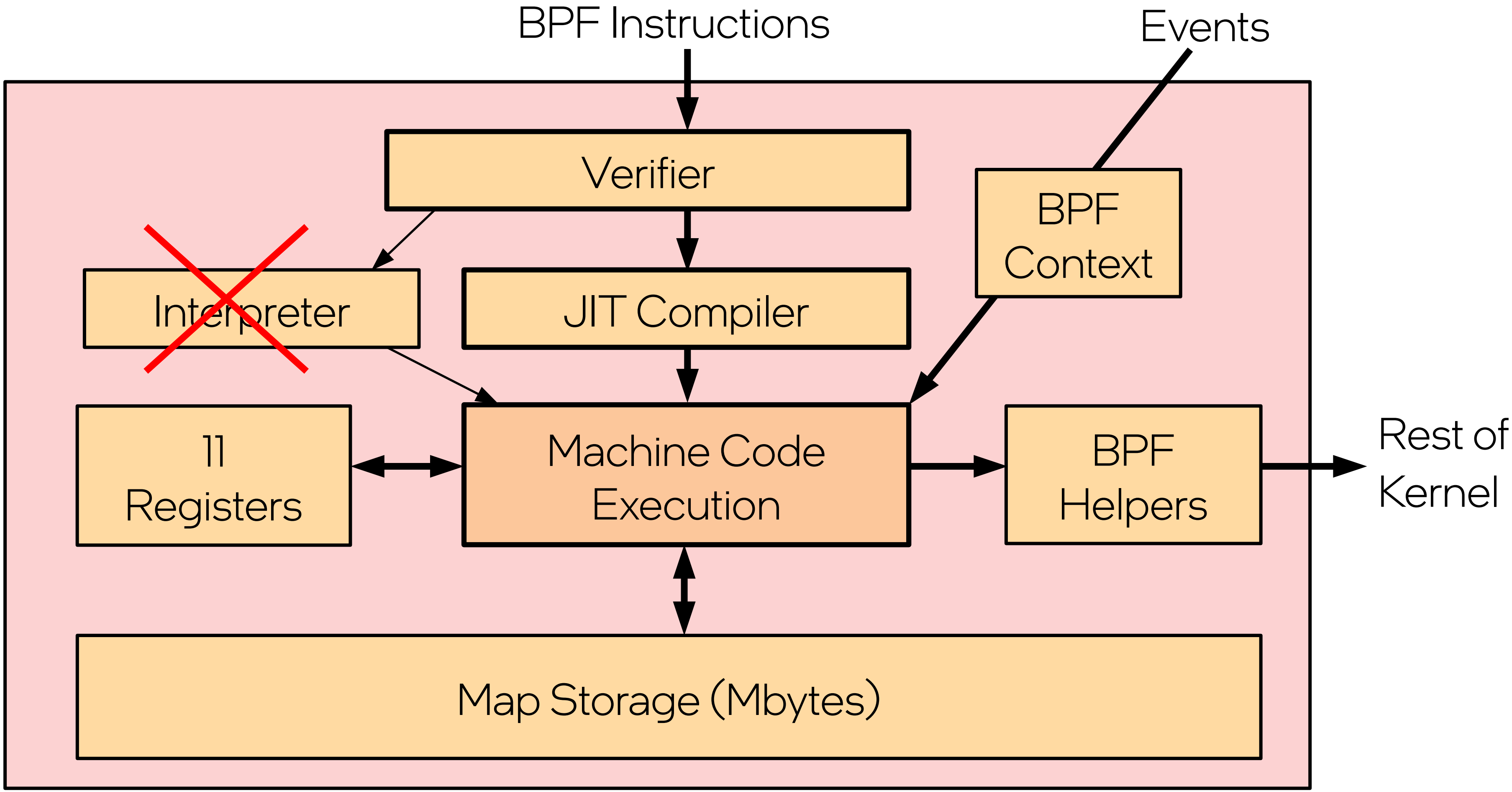
Extended BPF (eBPF) modernized BPF

	Classic BPF	Extended BPF
Word size	32-bit	64-bit
Registers	2	10+1
Storage	16 slots	512 byte stack + infinite map storage
Events	packets	many event sources

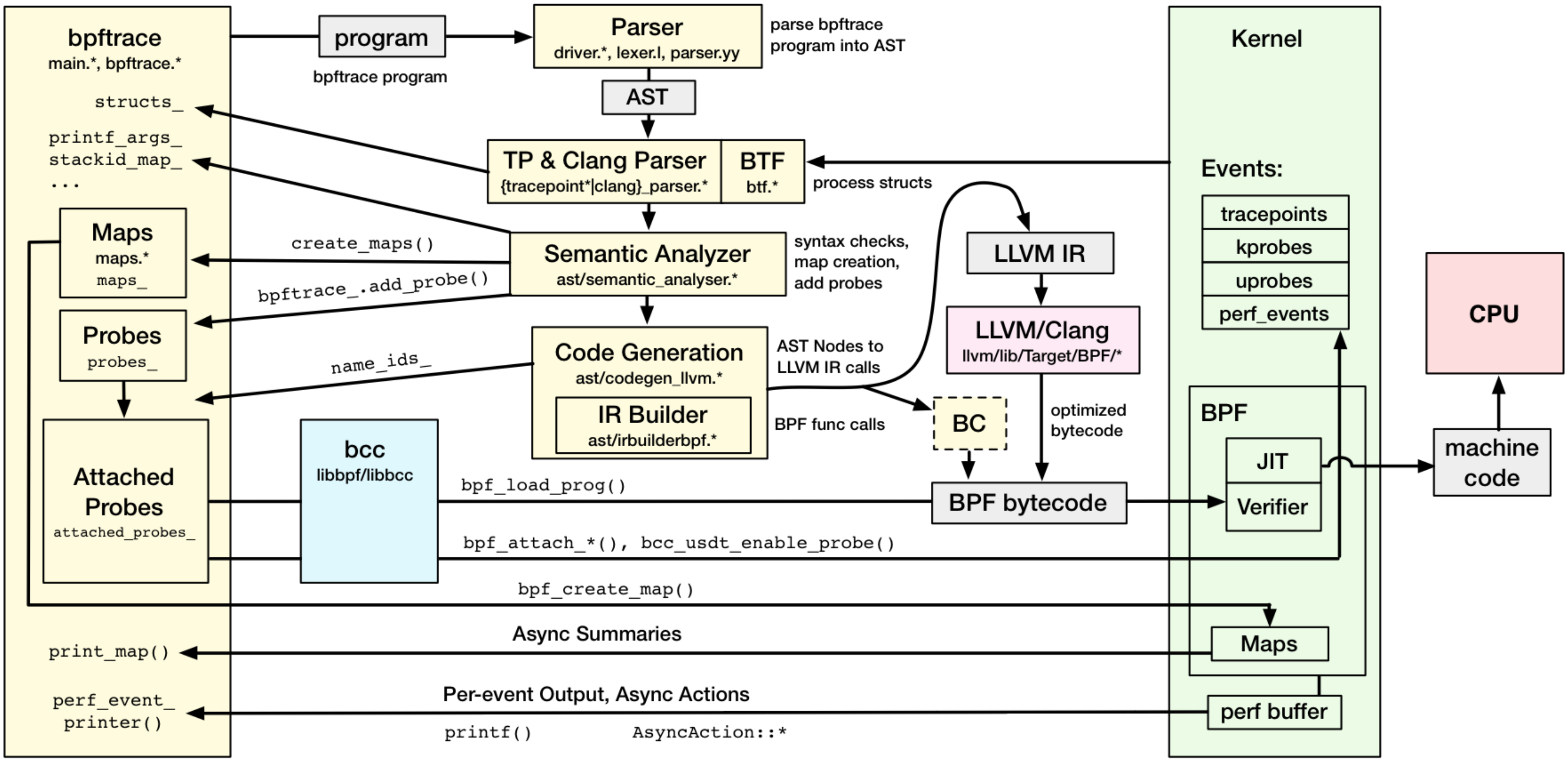
Maintainers/creators: Alexei Starovoitov & Daniel Borkmann

Old BPF is now “Classic BPF,” and eBPF is usually just “BPF”

BPF Internals

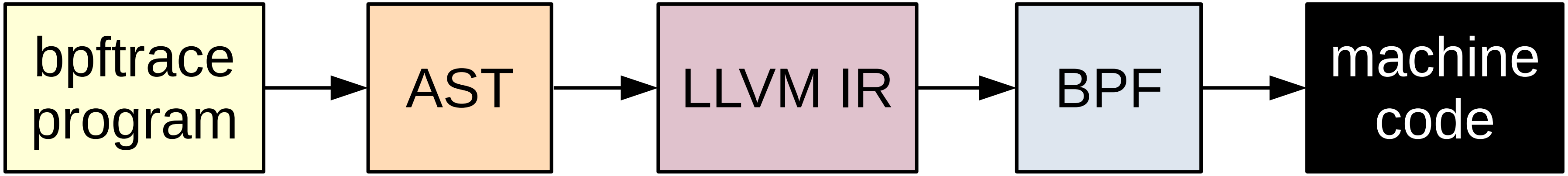


bpftrace Internals

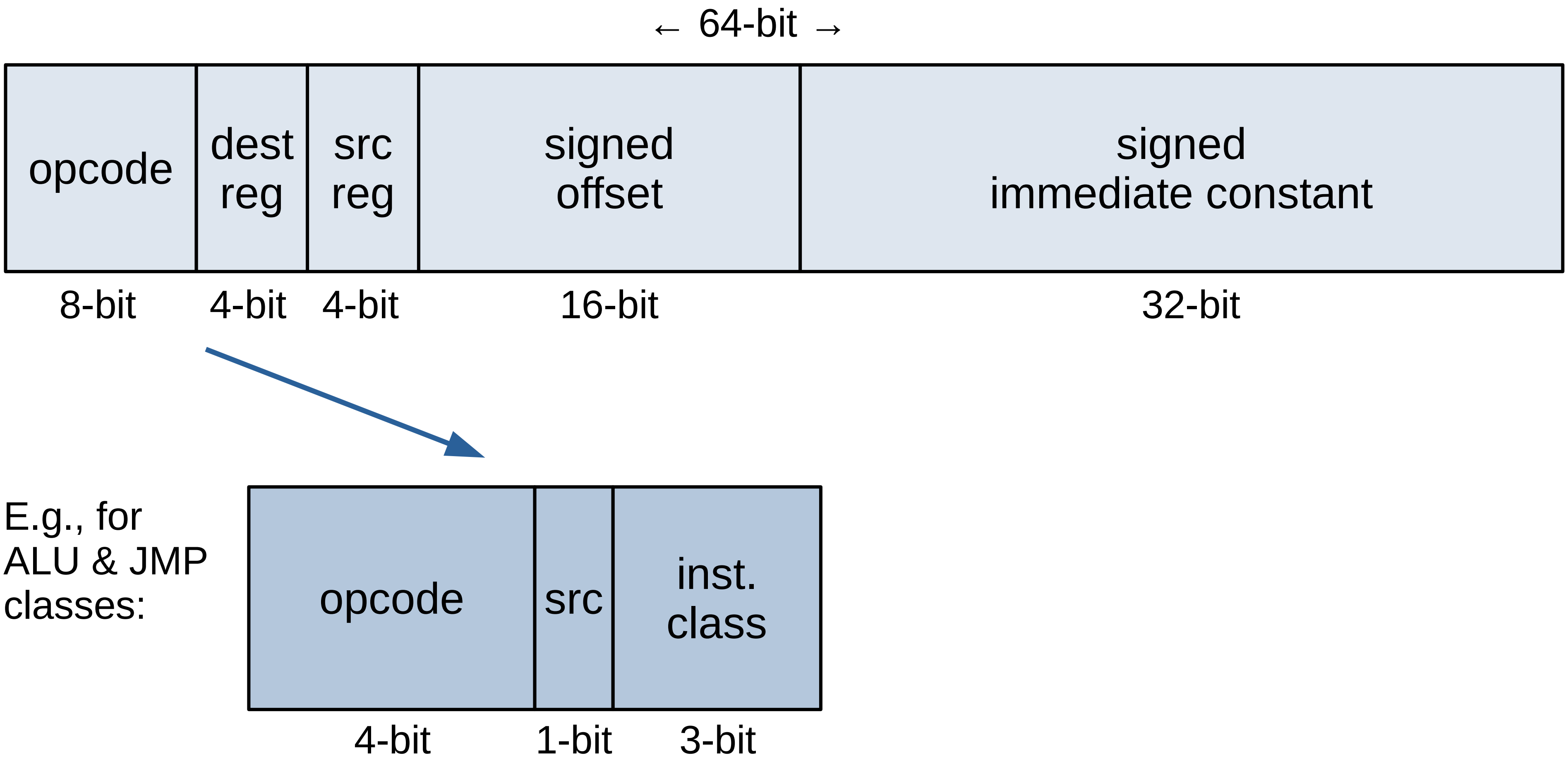


https://github.com/iovisor/bpftrace 2021, diagram by Brendan Gregg

bpftrace program transformations



Extended BPF instruction (bytecode) format



← 64-bit →

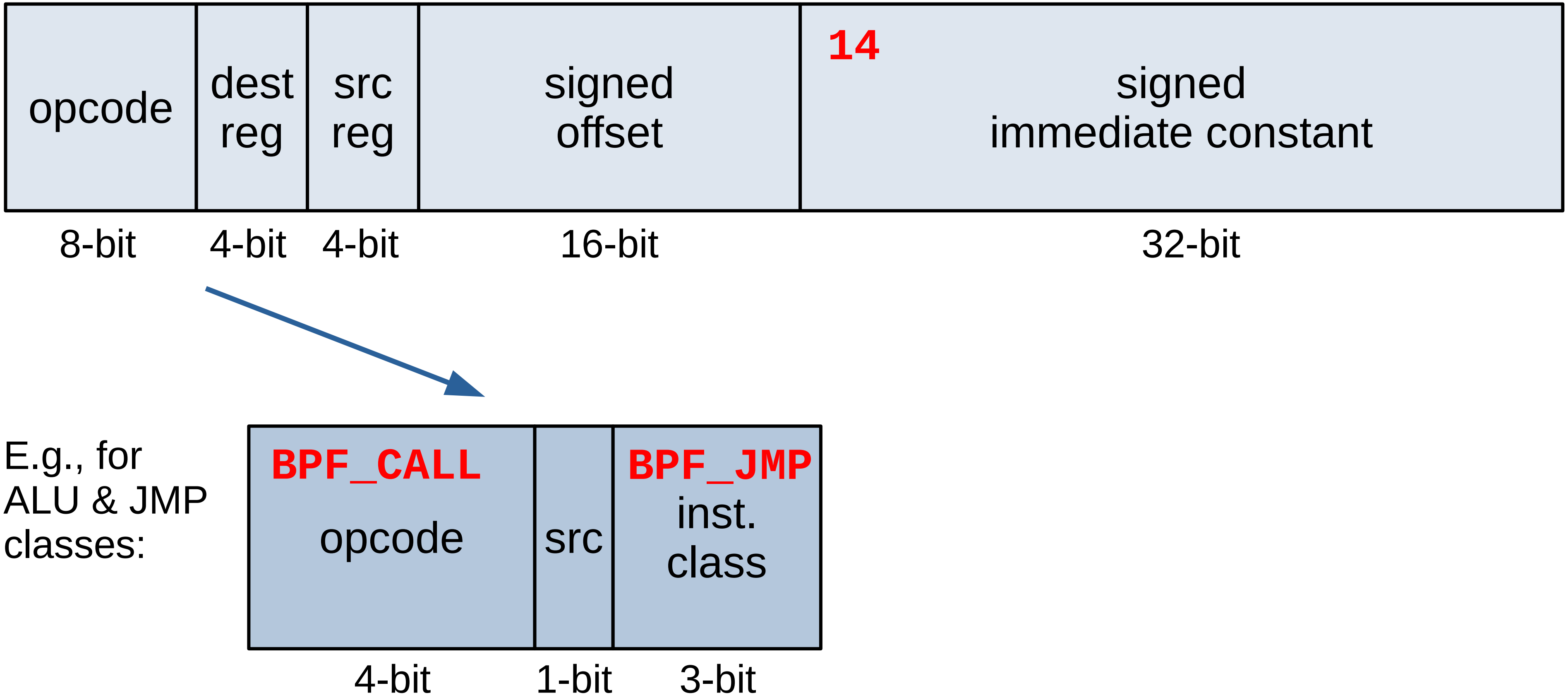
opcode	dest reg	src reg	signed offset	signed immediate constant
8-bit	4-bit	4-bit	16-bit	32-bit

E.g., for ALU & JMP classes:

opcode	src	inst. class
4-bit	1-bit	3-bit

Extended BPF instruction (bytecode) format (2)

E.g., call get_current_pid_tgid



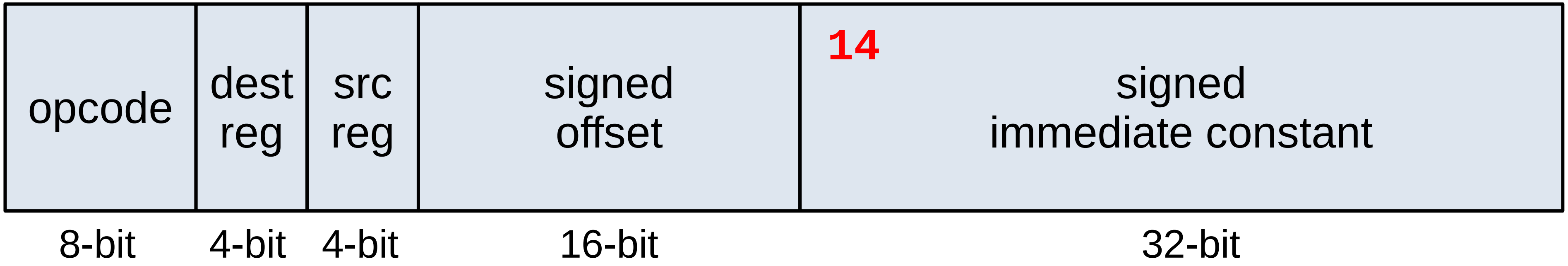
opcode	dest reg	src reg	signed offset	**14** signed immediate constant
8-bit	4-bit	4-bit	16-bit	32-bit

E.g., for ALU & JMP classes:

BPF_CALL opcode	src	**BPF_JMP** inst. class
4-bit	1-bit	3-bit

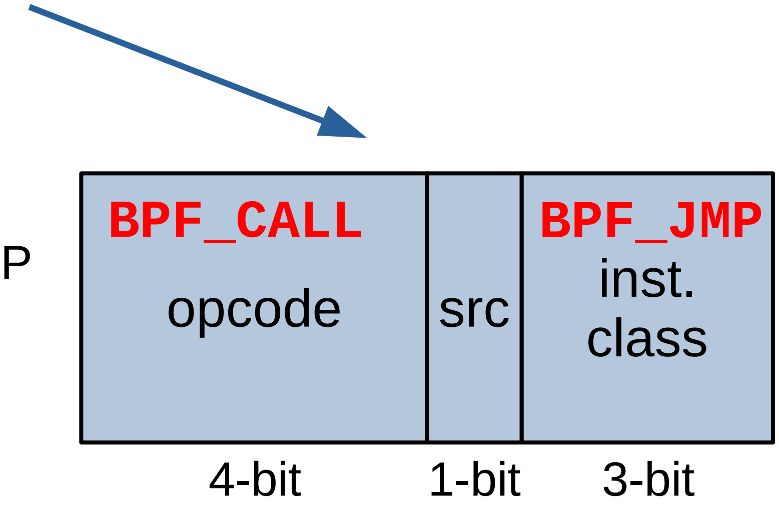
Extended BPF instruction (bytecode) format (3)

E.g., call get_current_pid_tgid



opcode	dest reg	src reg	signed offset	**14** signed immediate constant
8-bit	4-bit	4-bit	16-bit	32-bit

E.g., for ALU & JMP classes:



BPF_CALL opcode	src	**BPF_JMP** inst. class
4-bit	1-bit	3-bit

Linux include/uapi/linux/bpf.h

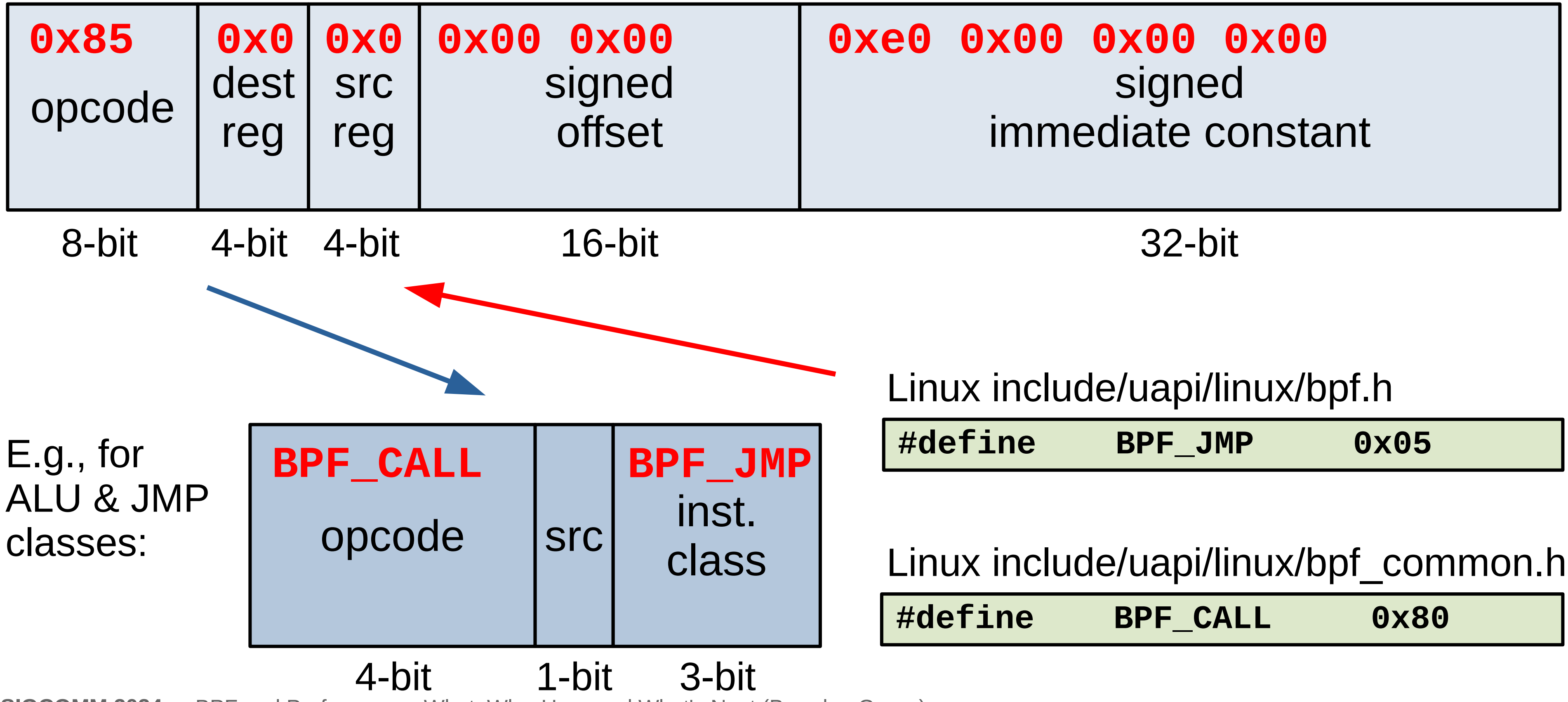
```
#define     BPF_JMP      0x05
```

Linux include/uapi/linux/bpf_common.h

```
#define     BPF_CALL      0x80
```

Extended BPF instruction (bytecode) format (4)

E.g., call get_current_pid_tgid



Linux include/uapi/linux/bpf.h

```
#define     BPF_JMP       0x05
```

Linux include/uapi/linux/bpf_common.h

```
#define     BPF_CALL      0x80
```

Extended BPF instruction (bytecode) format (5)

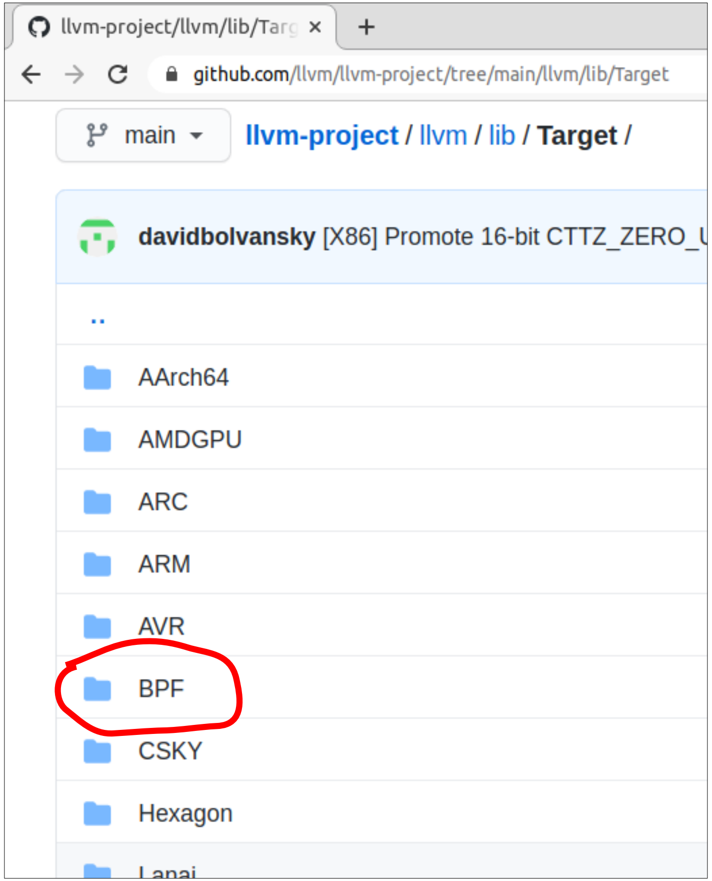
E.g., `call get_current_pid_tgid`

`(hex) 85 00 00 e0 00 00 00`

As per the BPF specification

(as defined in the Linux headers; now becoming the IETF standard)

LLVM/Clang has a BPF target



llvm-project/llvm/lib/Targ ×

github.com/llvm/llvm-project/tree/main/llvm/lib/Target

main

llvm-project / llvm / lib / **Target** /

davidbolvansky [X86] Promote 16-bit CTTZ_ZERO_U

..

AArch64

AMDGPU

ARC

ARM

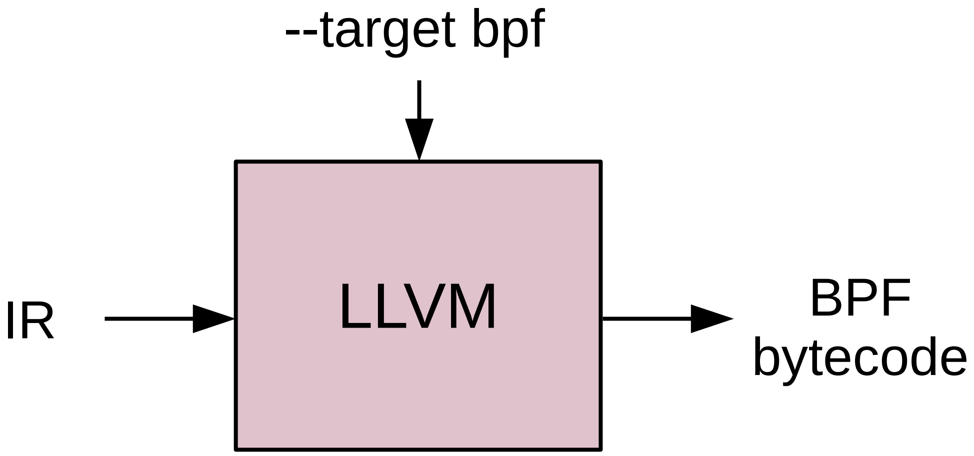
AVR

BPF

CSKY

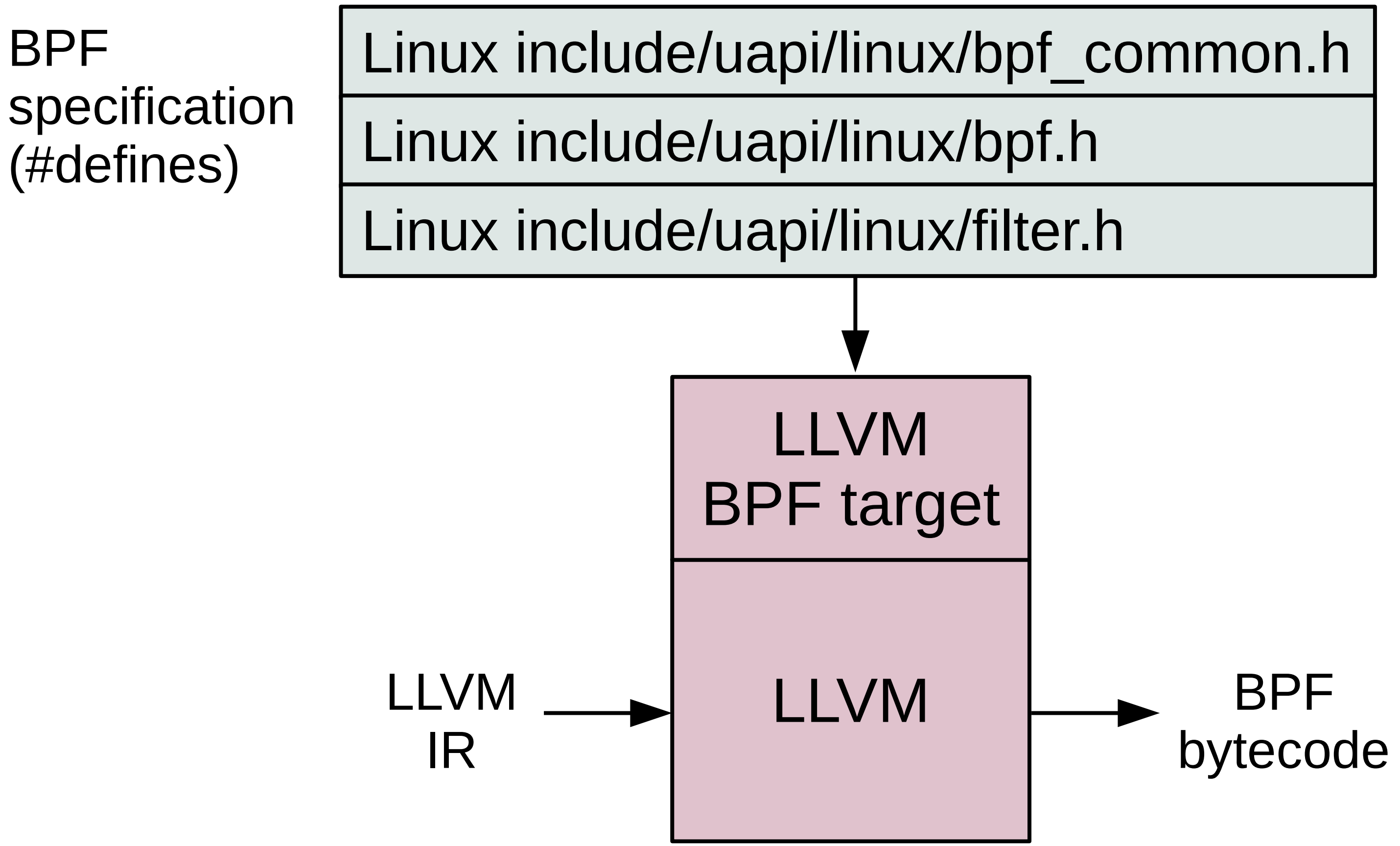
Hexagon

Lanai



Future: bpftrace may include its own lightweight bpftrace compiler (BC) as an *option*
(pros: no dependencies; cons: less optimal code)

LLVM/Clang has a BPF target (2)



BPF specification (#defines)

Linux include/uapi/linux/bpf_common.h

Linux include/uapi/linux/bpf.h

Linux include/uapi/linux/filter.h

LLVM BPF target

LLVM IR

LLVM

BPF bytecode

LLVM IR → BPF

E.g., tail call i64 inttoptr (i64 14 to i64 ()*)()

LLVM llvm/lib/Target/BPF/BPFInstrInfo.td

```
class CALL<string OpcodeStr>
    : TYPE_ALU_JMP<BPF_CALL.Value, BPF_K.Value,
                   (outs),
                   (ins calltarget:$BrDst),
                   !strconcat(OpcodeStr, " $BrDst"),
                   []> {
  bits<32> BrDst;

  let Inst{31-0} = BrDst; → 14
  let BPFClass = BPF_JMP;
}
```

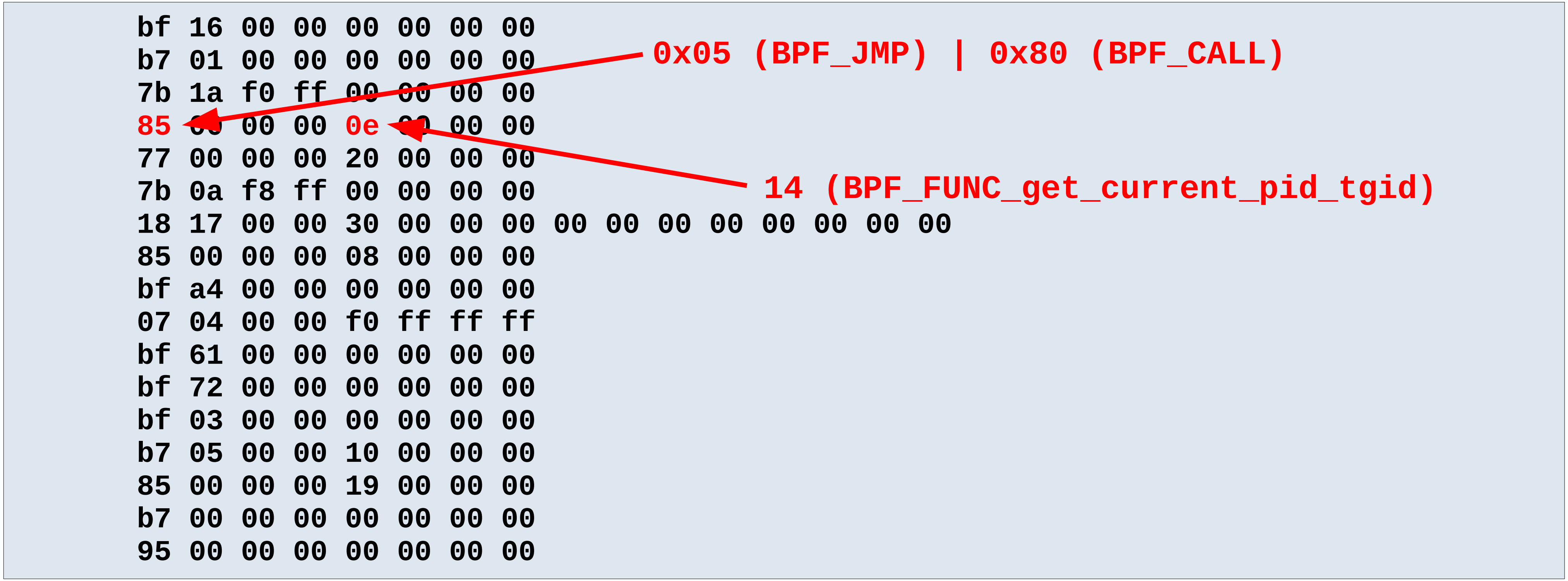
↓ Plus more llvm boilerplate & BPF headers shown earlier

85 00 00 e0 00 00 00

Now you have BPF bytecode!

```
        bf 16 00 00 00 00 00 00
        b7 01 00 00 00 00 00 00
        7b 1a f0 ff 00 00 00 00
        85 00 00 00 0e 00 00 00
        77 00 00 00 20 00 00 00
        7b 0a f8 ff 00 00 00 00
        18 17 00 00 30 00 00 00 00 00 00 00 00 00 00 00
        85 00 00 00 08 00 00 00
        bf a4 00 00 00 00 00 00
        07 04 00 00 f0 ff ff ff
        bf 61 00 00 00 00 00 00
        bf 72 00 00 00 00 00 00
        bf 03 00 00 00 00 00 00
        b7 05 00 00 10 00 00 00
        85 00 00 00 19 00 00 00
        b7 00 00 00 00 00 00 00
        95 00 00 00 00 00 00 00
```

Now you have BPF bytecode! (2)

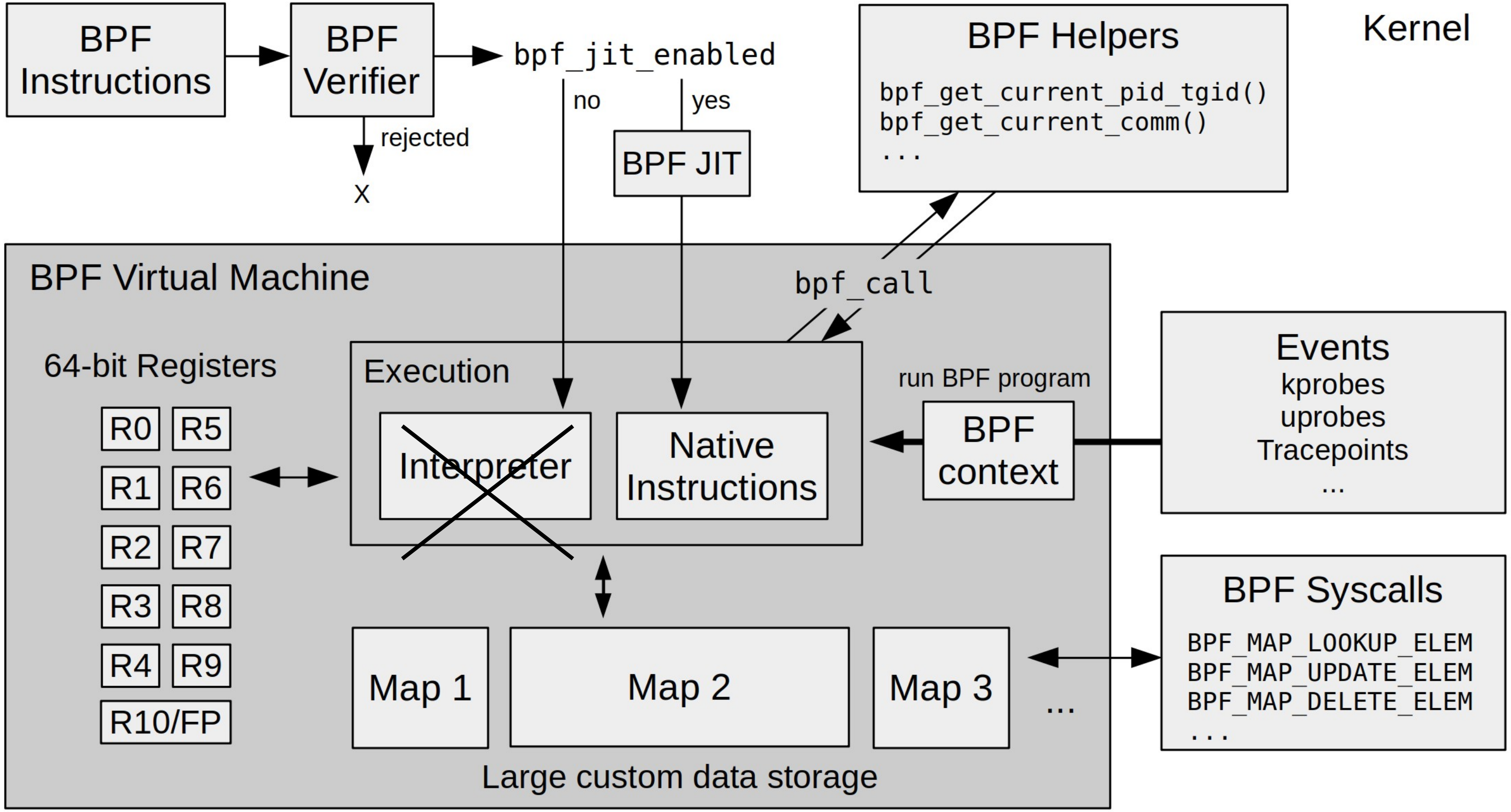


```
bf 16 00 00 00 00 00 00
b7 01 00 00 00 00 00 00
7b 1a f0 ff 00 00 00 00
85 00 00 00 0e 00 00 00
77 00 00 00 20 00 00 00
7b 0a f8 ff 00 00 00 00
18 17 00 00 30 00 00 00 00 00 00 00 00 00 00 00
85 00 00 00 08 00 00 00
bf a4 00 00 00 00 00 00
07 04 00 00 f0 ff ff ff
bf 61 00 00 00 00 00 00
bf 72 00 00 00 00 00 00
bf 03 00 00 00 00 00 00
b7 05 00 00 10 00 00 00
85 00 00 00 19 00 00 00
b7 00 00 00 00 00 00 00
95 00 00 00 00 00 00 00
```

0x05 (BPF_JMP) | 0x80 (BPF_CALL)

14 (BPF_FUNC_get_current_pid_tgid)

BPF mid-level internals



From: BPF Performance Tools, Figure 2-3

Verifying BPF instructions

```
85 00 00 00 12 34 56 78
```

Imagine we call a bogus function...

Linux kernel/bpf/verifier.c

```
static int do_check(struct bpf_verifier_env *env)
[...]
                } else if (class == BPF_JMP || class == BPF_JMP32) {
                        u8 opcode = BPF_OP(insn->code);
                        env->jmps_processed++;
                        if (opcode == BPF_CALL) {
[...]
                                        err = check_helper_call(env, insn->imm, env->insn_idx);
[...]
static int check_helper_call(struct bpf_verifier_env *env, int func_id, int insn_idx)
{
        const struct bpf_func_proto *fn = NULL;
        struct bpf_reg_state *regs;
        struct bpf_call_arg_meta meta;
        bool changes_data;
        int i, err;

        /* find function prototype */
        if (func_id < 0 || func_id >= __BPF_FUNC_MAX_ID) {
                verbose(env, "invalid func %s#%d\n", func_id_name(func_id),
                        func_id);
                return -EINVAL;
```

>20000 lines of code

BPF verifier

>20000 lines of code

>400 error returns

Checks every instruction

Checks every code path

Rewrites some bytecode

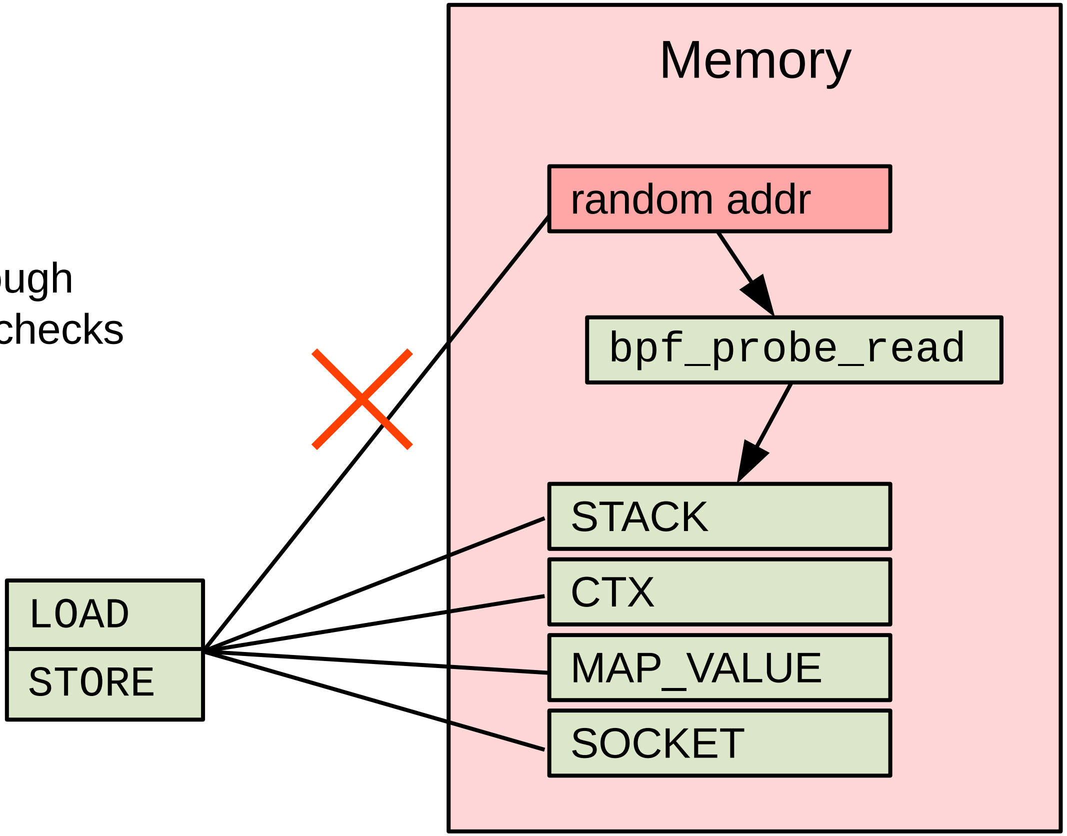
Verifier functions:

```
check_subprogs
check_reg_arg
check_stack_write
check_stack_read
check_stack_access
check_map_access_type
check_mem_region_access
check_map_access
check_packet_access
check_ctx_access
check_flow_keys_access
check_sock_access
check_pkt_ptr_alignment
check_generic_ptr_alignment
check_ptr_alignment
check_max_stack_depth
check_tp_buffer_access
check_ptr_to_btf_access
check_mem_access
check_xadd
check_stack_boundary
```

```
check_helper_mem_access
check_func_arg
check_map_func_compatibility
check_func_proto
check_func_call
check_reference_leak
check_helper_call
check_alu_op
check_cond_jmp_op
check_ld_imm
check_ld_abs
check_return_code
check_cfg
check_btf_func
check_btf_line
check_btf_info
check_map_prealloc
check_map_prog_compatibility
check_struct_ops_btf_id
check_attach_modify_return
check_attach_btf_id
```

Verifying Instructions

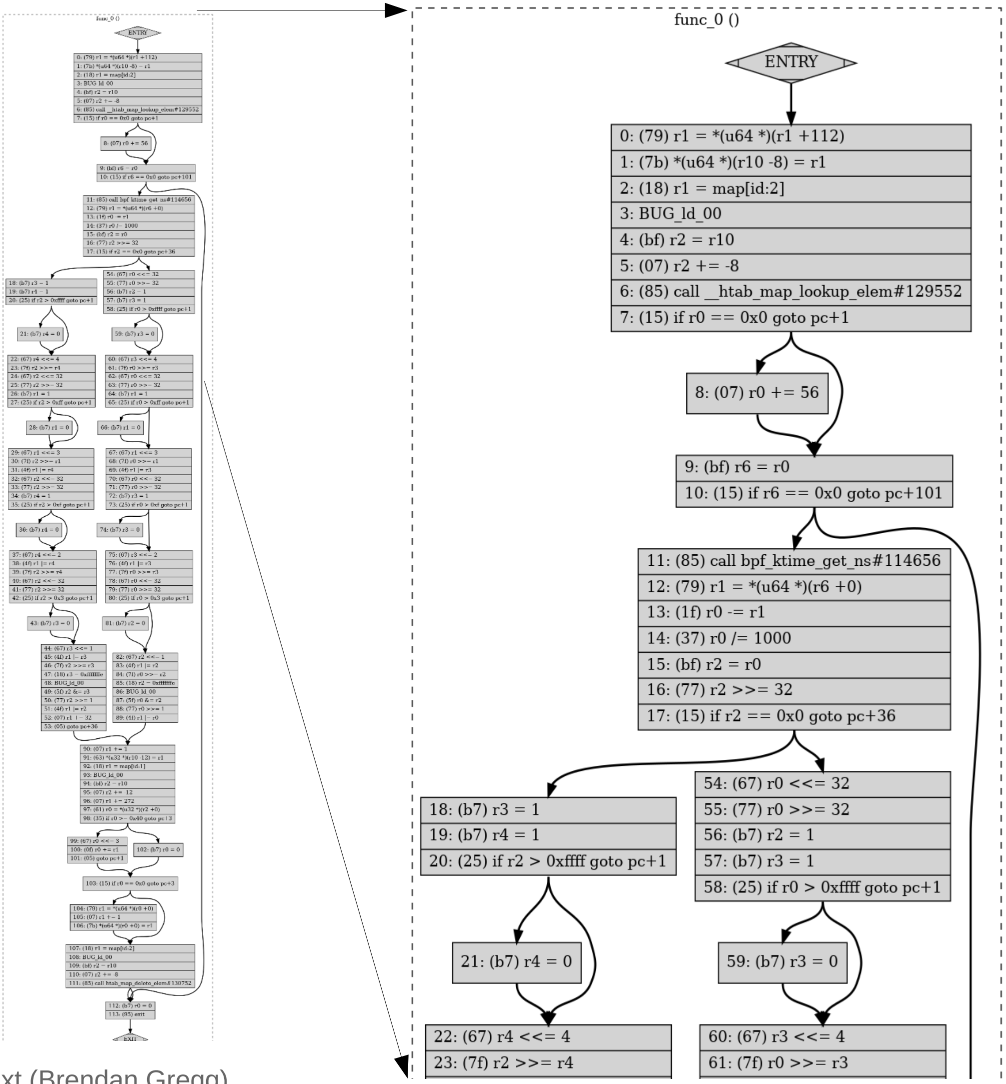
- Memory access
  - Direct access extremely restricted
  - Can only read initialized memory
  - Other kernel memory must pass through the bpf_probe_read() helper and its checks
- Arguments are the correct type
- Register usage allowed
  - E.g., no frame pointer writes
- No write overflows
- No addr leaks
- Etc.



Verifying Code Paths

- All instruction must lead to exit
- No unreachable instructions
- No backwards branches (loops) except BPF bounded loops

biolatency as GraphViz dot:



```
0: (79) r1 = *(u64 *)(r1 +112)
1: (7b) *(u64 *)(r10 -8) = r1
2: (18) r1 = map[id:2]
3: BUG_ld_00
4: (bf) r2 = r10
5: (07) r2 += -8
6: (85) call __htab_map_lookup_elem#129552
7: (15) if r0 == 0x0 goto pc+1
```

```
8: (07) r0 += 56
```

```
9: (bf) r6 = r0
10: (15) if r6 == 0x0 goto pc+101
```

```
11: (85) call bpf_ktime_get_ns#114656
12: (79) r1 = *(u64 *)(r6 +0)
13: (1f) r0 -= r1
14: (37) r0 /= 1000
15: (bf) r2 = r0
16: (77) r2 >>= 32
17: (15) if r2 == 0x0 goto pc+36
```

```
18: (b7) r3 = 1
19: (b7) r4 = 1
20: (25) if r2 > 0xffff goto pc+1
```

```
54: (67) r0 <<= 32
55: (77) r0 >>= 32
56: (b7) r2 = 1
57: (b7) r3 = 1
58: (25) if r0 > 0xffff goto pc+1
```

```
21: (b7) r4 = 0
```

```
59: (b7) r3 = 0
```

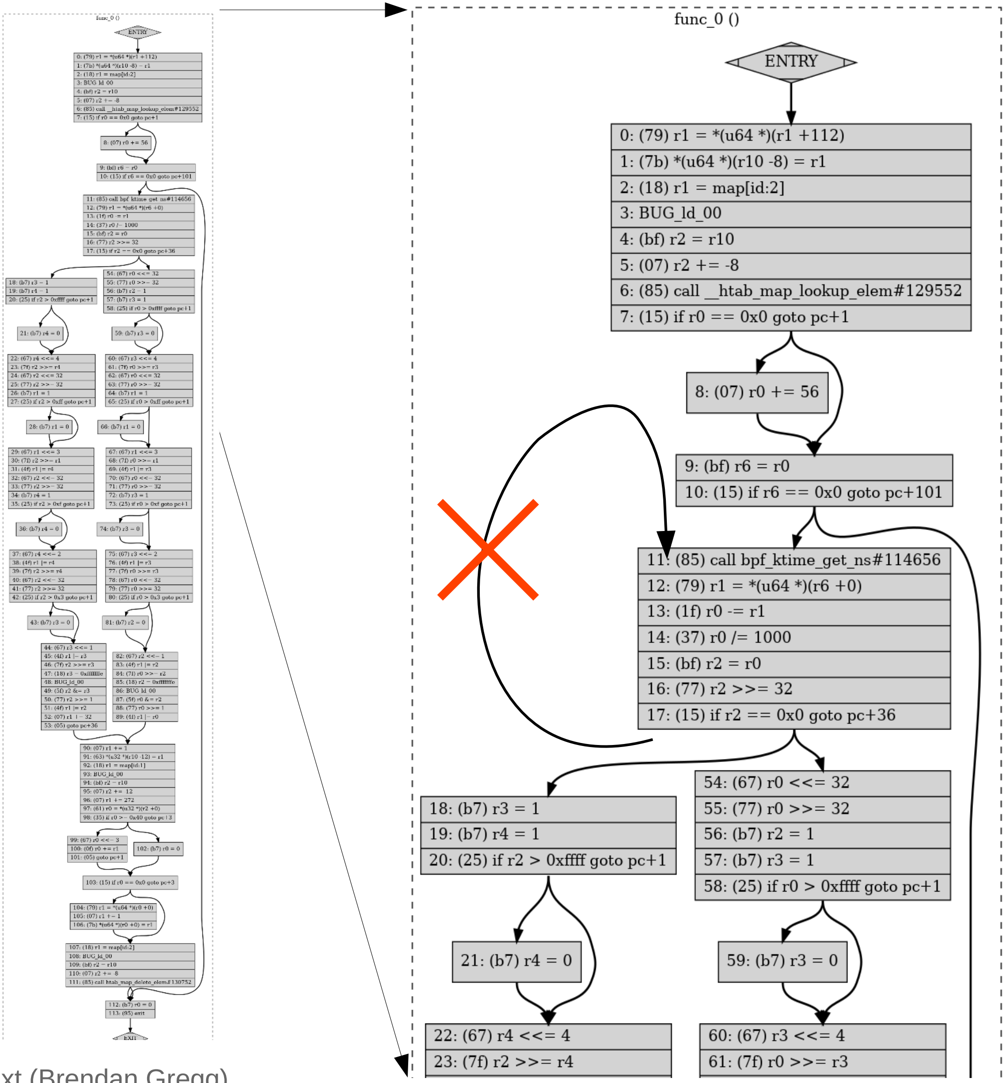
```
22: (67) r4 <<= 4
23: (7f) r2 >>= r4
```

```
60: (67) r3 <<= 4
61: (7f) r0 >>= r3
```

Verifying Code Paths

- All instruction must lead to exit
- No unreachable instructions
- No backwards branches (loops) except BPF bounded loops

biolatency as GraphViz dot:



```
func_0 ()
ENTRY
0: (79) r1 = *(u64 *)(r1 +112)
1: (7b) *(u64 *)(r10 -8) = r1
2: (18) r1 = map[id:2]
3: BUG_ld_00
4: (bf) r2 = r10
5: (07) r2 += -8
6: (85) call __htab_map_lookup_elem#129552
7: (15) if r0 == 0x0 goto pc+1
8: (07) r0 += 56
9: (bf) r6 = r0
10: (15) if r6 == 0x0 goto pc+101
11: (85) call bpf_ktime_get_ns#114656
12: (79) r1 = *(u64 *)(r6 +0)
13: (1f) r0 -= r1
14: (37) r0 /= 1000
15: (bf) r2 = r0
16: (77) r2 >>= 32
17: (15) if r2 == 0x0 goto pc+36
18: (b7) r3 = 1
19: (b7) r4 = 1
20: (25) if r2 > 0xffff goto pc+1
21: (b7) r4 = 0
22: (67) r4 <<= 4
23: (7f) r2 >>= r4
24: (67) r2 <<= 32
25: (77) r2 >>= 32
26: (b7) r1 = 1
27: (25) if r2 > 0xff goto pc+1
28: (b7) r1 = 0
29: (67) r1 <<= 3
30: (7f) r2 >>= r1
31: (4f) r1 |= r4
32: (67) r2 <<= 32
33: (77) r2 >>= 32
34: (b7) r4 = 1
35: (25) if r2 > 0xf goto pc+1
36: (b7) r4 = 0
37: (67) r4 <<= 2
38: (4f) r1 |= r4
39: (7f) r2 >>= r4
40: (67) r2 <<= 32
41: (77) r2 >>= 32
42: (25) if r2 > 0x3 goto pc+1
43: (b7) r3 = 0
44: (67) r3 <<= 1
45: (4f) r1 |= r3
46: (7f) r2 >>= r3
47: (18) r3 = 0xfffffffe
48: BUG_ld_00
49: (5f) r2 &= r3
50: (77) r2 >>= 1
51: (4f) r1 |= r2
52: (07) r1 += 32
53: (05) goto pc+36
54: (67) r0 <<= 32
55: (77) r0 >>= 32
56: (b7) r2 = 1
57: (b7) r3 = 1
58: (25) if r0 > 0xffff goto pc+1
59: (b7) r3 = 0
60: (67) r3 <<= 4
61: (7f) r0 >>= r3
62: (67) r0 <<= 32
63: (77) r0 >>= 32
64: (b7) r1 = 1
65: (25) if r0 > 0xff goto pc+1
66: (b7) r1 = 0
67: (67) r1 <<= 3
68: (7f) r0 >>= r1
69: (4f) r1 |= r3
70: (67) r0 <<= 32
71: (77) r0 >>= 32
72: (b7) r3 = 1
73: (25) if r0 > 0xf goto pc+1
74: (b7) r3 = 0
75: (67) r3 <<= 2
76: (4f) r1 |= r3
77: (7f) r0 >>= r3
78: (67) r0 <<= 32
79: (77) r0 >>= 32
80: (25) if r0 > 0x3 goto pc+1
81: (b7) r2 = 0
82: (67) r2 <<= 1
83: (4f) r1 |= r2
84: (7f) r0 >>= r2
85: (18) r2 = 0xfffffffe
86: BUG_ld_00
87: (5f) r0 &= r2
88: (77) r0 >>= 1
89: (4f) r1 |= r0
90: (07) r1 += 1
91: (63) *(u32 *)(r10 -12) = r1
92: (18) r1 = map[id:1]
93: BUG_ld_00
94: (bf) r2 = r10
95: (07) r2 += -12
96: (07) r1 += 272
97: (61) r0 = *(u32 *)(r2 +0)
98: (35) if r0 >= 0x40 goto pc+3
99: (67) r0 <<= 3
100: (0f) r0 += r1
101: (05) goto pc+1
102: (b7) r0 = 0
103: (15) if r0 == 0x0 goto pc+3
104: (79) r1 = *(u64 *)(r0 +0)
105: (07) r1 += 1
106: (7b) *(u64 *)(r0 +0) = r1
107: (18) r1 = map[id:2]
108: BUG_ld_00
109: (bf) r2 = r10
110: (07) r2 += -8
111: (85) call htab_map_delete_elem#130752
112: (b7) r0 = 0
113: (95) exit
EXIT
```

Pre-verifier BPF bytecode

```
     bf 16 00 00 00 00 00 00
     b7 01 00 00 00 00 00 00
     7b 1a f0 ff 00 00 00 00
     85 00 00 00 0e 00 00 00
     77 00 00 00 20 00 00 00
     7b 0a f8 ff 00 00 00 00
     18 17 00 00 30 00 00 00 00 00 00 00 00 00 00 00
     85 00 00 00 08 00 00 00
     bf a4 00 00 00 00 00 00
     07 04 00 00 f0 ff ff ff
     bf 61 00 00 00 00 00 00
     bf 72 00 00 00 00 00 00
     bf 03 00 00 00 00 00 00
     b7 05 00 00 10 00 00 00
     85 00 00 00 19 00 00 00
     b7 00 00 00 00 00 00 00
     95 00 00 00 00 00 00 00
```

Post-verifier BPF bytecode

```
      bf 16 00 00 00 00 00 00
      b7 01 00 00 00 00 00 00
      7b 1a f0 ff 00 00 00 00
      85 00 00 00 d0 81 01 00
      77 00 00 00 20 00 00 00
      7b 0a f8 ff 00 00 00 00
      18 17 00 00 18 00 00 00 00 00 00 00 00 00 00 00
      85 00 00 00 f0 80 01 00
      bf a4 00 00 00 00 00 00
      07 04 00 00 f0 ff ff ff
      bf 61 00 00 00 00 00 00
      bf 72 00 00 00 00 00 00
      bf 03 00 00 00 00 00 00
      b7 05 00 00 10 00 00 00
      85 00 00 00 30 2c ff ff
      b7 00 00 00 00 00 00 00
      95 00 00 00 00 00 00 00
```

Post-verifier BPF bytecode (2)

```
bf 16 00 00 00 00 00 00
b7 01 00 00 00 00 00 00
7b 1a f0 ff 00 00 00 00
85 00 00 00 d0 81 01 00
77 00 00 00 20 00 00 00
7b 0a f8 ff 00 00 00 00
18 17 00 00 18 00 00 00 00 00 00 00 00 00 00 00
85 00 00 00 f0 80 01 00
bf a4 00 00 00 00 00 00
07 04 00 00 f0 ff ff ff
bf 61 00 00 00 00 00 00
bf 72 00 00 00 00 00 00
bf 03 00 00 00 00 00 00
b7 05 00 00 10 00 00 00
85 00 00 00 30 2c ff ff
b7 00 00 00 00 00 00 00
95 00 00 00 00 00 00 00
```

E.g., call get_current_pid_tgid

helper index value has become an instruction offset addresses from __bpf_call_base

BPF Internals: More Information

- https://www.brendangregg.com/blog/2021-06-15/bpf-internals.html
- Linux include/uapi/linux/bpf_common.h
- Linux include/uapi/linux/bpf.h
- Linux include/uapi/linux/filter.h
- https://docs.cilium.io/en/v1.15/bpf/
- https://ebpf.io/what-is-ebpf
- https://lwn.net/Kernel/Index/#BPF
- https://events.static.linuxfound.org/sites/events/files/slides/bpf_collabsummit_2015feb20.pdf
- https://kernel-recipes.org/en/2022/talks/the-untold-story-of-bpf/
- BPF Performance Tools, Addison-Wesley 2020, chapter 2

BPF Internals

(eBPF)

Tracing Examples

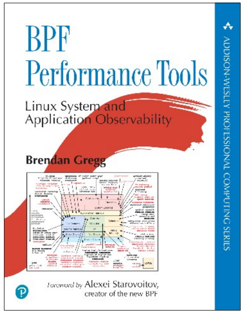
Brendan Gregg

USENIX

LISA21

Jun, 2021

NETFLIX



BPF Performance Tools

Linux System and Application Observability

Brendan Gregg

Foreword by Alexei Starovoitov, creator of the new BPF

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

BPF Recent Additions (2019-2024)

...absent from docs/programs/talks that were written prior to their existence

- BTF: BPF Type Format
- CO-RE: Compile Once Run Everywhere
- Bounded loops
- Multi-event-attach (faster init)
- Somewhat faster uprobes (but not fast yet)
- kfunc: lighter-weight kprobes
- sched_ext: Kernel CPU scheduler hooks
- Work on signed BPF

BTF & CO-RE

Allowing small stand-alone ELF/BPF binaries

BTF: BPF Type Format

CO-RE: Compile Once Run Everywhere

Agenda

1) What: A type of software
2) Why: Case Study
3) **How:** History, Internals, **Usage**, Recommendations
4) What's Next: Challenges, Future
5) Discussion & Q&A

Major BPF Performance Tools

bcc

https://github.com/iovisor/bcc/

BPF Compiler Collection
100 performance tools
C programming

bpftrace

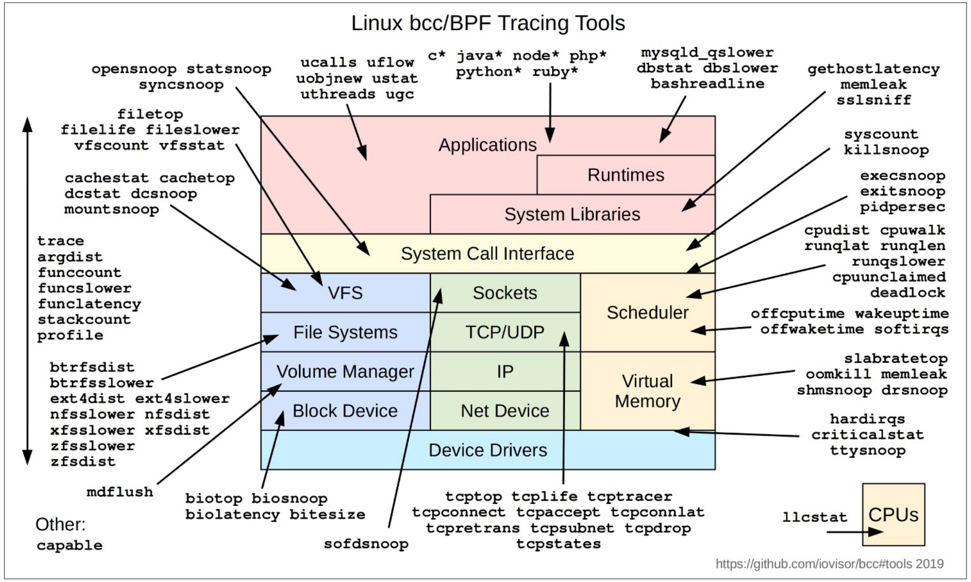
https://github.com/bpftrace/bpftrace

BPF Tracer
~40 basic tools
Scripting

Note: complies to the same bytecode as bcc,
so is just as fast. Easier to write.

Ubuntu Install (2024)

Ubuntu 24.04 Includes the bcc and bpftrace tools by default



https://github.com/iovisor/bcc#tools 2019

For **older** Ubuntu:

BCC (BPF Compiler Collection): complex tools

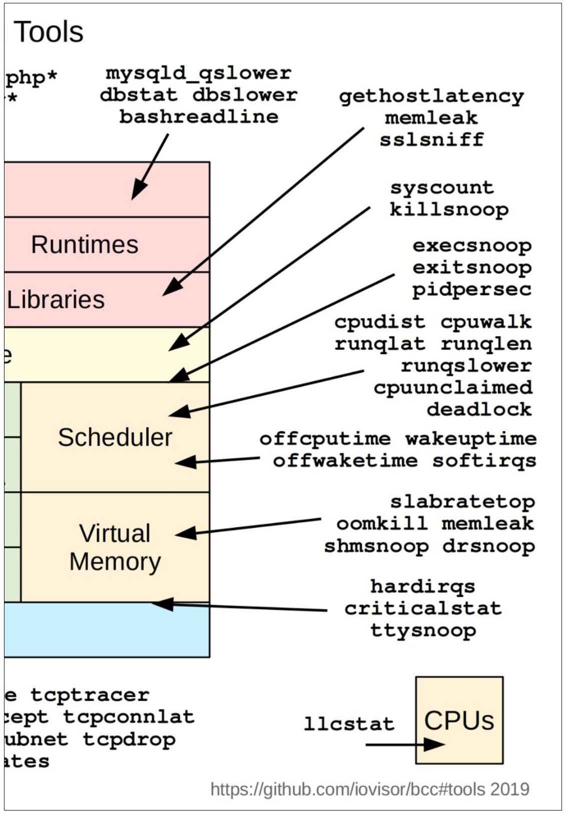
```
# apt install bpfcc-tools
```

bpftrace: custom tools (Ubuntu 19.04+)

```
# apt install bpftrace
```

These are default installs at Netflix, Meta, etc.

Some Tool Examples



https://github.com/iovisor/bcc#tools 2019

`*.bt`: bpftrace, incl. some book tools[0]

`*.py`: BCC Python, obsolete interface

`*`[no ext]: BCC C libbpf-tools

(some tools exist for all three)

[0] https://github.com/brendangregg/bpf-perf-tools-book

CPUs: execsnoop

New process trace

```
# execsnoop -T
TIME(s) PCOMM                PID    PPID    RET ARGS
0.506   run                  8745   1828      0 ./run
0.507   bash                 8745   1828      0 /bin/bash
0.511   svstat               8747   8746      0 /command/svstat /service/nflx-httpd
0.511   perl                 8748   8746      0 /usr/bin/perl -e $l=<>;$l=~/(\d+) sec/;pr...
0.514   ps                   8750   8749      0 /bin/ps --ppid 1 -o pid,cmd,args
0.514   grep                 8751   8749      0 /bin/grep org.apache.catalina
0.514   sed                  8752   8749      0 /bin/sed s/^ *//;
0.515   xargs                8754   8749      0 /usr/bin/xargs
0.515   cut                  8753   8749      0 /usr/bin/cut -d  -f 1
0.523   echo                 8755   8754      0 /bin/echo
0.524   mkdir                8756   8745      0 /bin/mkdir -v -p /data/tomcat
[...]
1.528   run                  8785   1828      0 ./run
1.529   bash                 8785   1828      0 /bin/bash
1.533   svstat               8787   8786      0 /command/svstat /service/nflx-httpd
1.533   perl                 8788   8786      0 /usr/bin/perl -e $l=<>;$l=~/(\d+) sec/;pr...
[...]
```

CPUs: runqlat

Scheduler latency (run queue latency)

```
# runqlat 10 1
Tracing run queue latency... Hit Ctrl-C to end.

     usecs               : count     distribution
         0 -> 1          : 1906     |***                                     |
         2 -> 3          : 22087    |****************************************|
         4 -> 7          : 21245    |**************************************  |
         8 -> 15         : 7333     |*************                           |
        16 -> 31         : 4902     |********                                |
        32 -> 63         : 6002     |**********                              |
        64 -> 127        : 7370     |*************                           |
       128 -> 255        : 13001    |***********************                 |
       256 -> 511        : 4823     |********                                |
       512 -> 1023       : 1519     |**                                      |
      1024 -> 2047       : 3682     |******                                  |
      2048 -> 4095       : 3170     |*****                                   |
      4096 -> 8191       : 5759     |**********                              |
      8192 -> 16383      : 14549    |**************************              |
     16384 -> 32767      : 5589     |**********                              |
```

Disks: biolatency

Disk I/O latency histograms, per second

```
# biolatency -mT 1 5
Tracing block device I/O... Hit Ctrl-C to end.


06:20:16
     msecs               : count     distribution
       0 -> 1            : 36       |**************************************|
       2 -> 3            : 1        |*                                     |
       4 -> 7            : 3        |***                                   |
       8 -> 15           : 17       |*****************                     |
      16 -> 31           : 33       |**********************************    |
      32 -> 63           : 7        |*******                               |
      64 -> 127          : 6        |******                                |


06:20:17
     msecs               : count     distribution
       0 -> 1            : 96       |************************************  |
       2 -> 3            : 25       |*********                             |
[...]
```

File Systems: xfsslower

XFS I/O slower than a threshold (variants for ext4, btrfs, zfs)

```
# xfsslower.py 50
Tracing XFS operations slower than 50 ms
TIME     COMM           PID    T BYTES   OFF_KB   LAT(ms) FILENAME
21:20:46 java           112789 R 8012    13925      60.16 file.out
21:20:47 java           112789 R 3571    4268      136.60 file.out
21:20:49 java           112789 R 5152    1780       63.88 file.out
21:20:52 java           112789 R 5214    12434     108.47 file.out
21:20:52 java           112789 R 7465    19379      58.09 file.out
21:20:54 java           112789 R 5326    12311      89.14 file.out
21:20:55 java           112789 R 4336    3051       67.89 file.out
[...]
22:02:39 java           112789 R 65536   1486748   182.10 shuffle_6_646_0.data
22:02:39 java           112789 R 65536   872492     30.10 shuffle_6_646_0.data
22:02:39 java           112789 R 65536   1113896   309.52 shuffle_6_646_0.data
22:02:39 java           112789 R 65536   1481020   400.31 shuffle_6_646_0.data
22:02:39 java           112789 R 65536   1415232   324.92 shuffle_6_646_0.data
22:02:39 java           112789 R 65536   1147912   119.37 shuffle_6_646_0.data
[...]
```

Networking: tcplife

TCP session lifespans with connection details

```
# tcplife
PID   COMM       LADDR           LPORT RADDR           RPORT TX_KB RX_KB MS
22597 recordProg 127.0.0.1       46644 127.0.0.1       28527     0     0 0.23
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46644     0     0 0.28
22598 curl       100.66.3.172    61620 52.205.89.26    80        0     1 91.79
22604 curl       100.66.3.172    44400 52.204.43.121   80        0     1 121.38
22624 recordProg 127.0.0.1       46648 127.0.0.1       28527     0     0 0.22
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46648     0     0 0.27
22647 recordProg 127.0.0.1       46650 127.0.0.1       28527     0     0 0.21
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46650     0     0 0.26
[...]
```

Networking: tcpsynbl (book)

TCP SYN backlogs as histograms

```
# tcpsynbl.bt
Attaching 4 probes...
Tracing SYN backlog size. Ctrl-C to end.
^C
@backlog[backlog limit]: histogram of backlog size

@backlog[128]:
[0]                    2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|

@backlog[500]:
[0]                 2783 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1]                    9 |                                                    |
[2, 4)                 4 |                                                    |
[4, 8)                 1 |                                                    |
```

Applications: mysqld_qslower

MySQL queries slower than a threshold

```
# mysqld_qslower.py $(pgrep mysqld)
Tracing MySQL server queries for PID 9908 slower than 1 ms...
TIME(s)    PID         MS QUERY
0.000000   9962  169.032 SELECT * FROM words WHERE word REGEXP '^bre.*n$'
1.962227   9962  205.787 SELECT * FROM words WHERE word REGEXP '^bpf.tools$'
9.043242   9962   95.276 SELECT COUNT(*) FROM words
23.723025  9962  186.680 SELECT count(*) AS count FROM words WHERE word REGEXP
'^bre.*n$'
30.343233  9962  181.494 SELECT * FROM words WHERE word REGEXP '^bre.*n$' ORDER
BY word
[...]
```

Kernel: workq (book)

Work queue function execution times

```
# workq.bt
Attaching 4 probes...
Tracing workqueue request latencies. Ctrl-C to end.
^C
@us[blk_mq_timeout_work]:
[1]                    1 |@@                                                  |
[2, 4)                11 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@                     |
[4, 8)                18 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|


@us[xfs_end_io]:
[1]                    2 |@@@@@@@@                                            |
[2, 4)                 6 |@@@@@@@@@@@@@@@@@@@@@@@@@@                          |
[4, 8)                 6 |@@@@@@@@@@@@@@@@@@@@@@@@@@                          |
[8, 16)               12 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[16, 32)              12 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[32, 64)               3 |@@@@@@@@@@@@@                                       |


[...]
```

Containers: blkthrot (book)

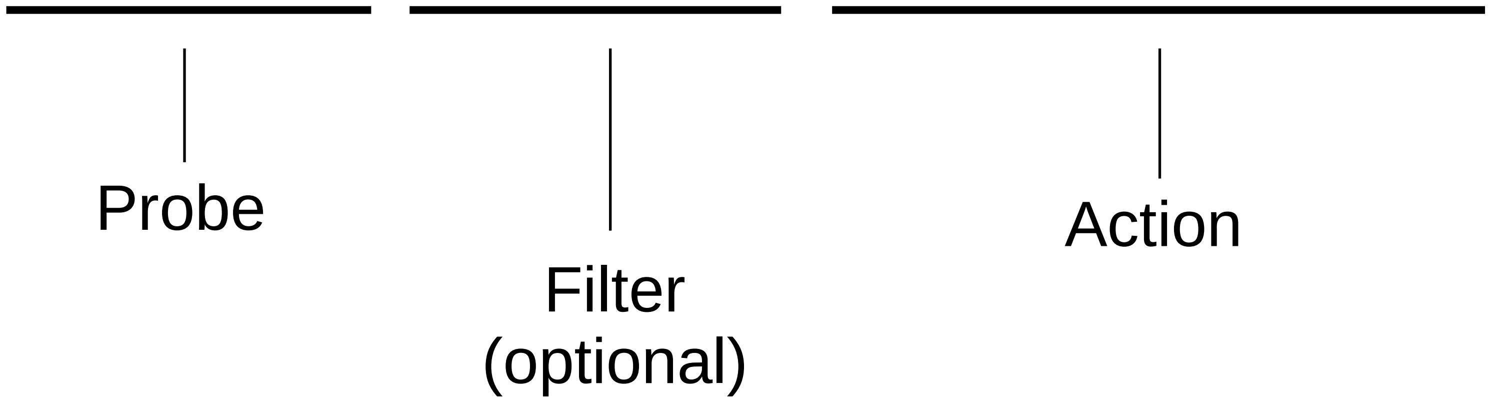
Count block I/O throttles by blk cgroup

```
# blkthrot.bt
Attaching 3 probes...
Tracing block I/O throttles by cgroup. Ctrl-C to end
^C

@notthrottled[1]: 506

@throttled[1]: 31
```

bpftrace

Ad-hoc tools and one-liners:

```
bpftrace -e 'kr:vfs_read /retval > 0/ { @ = hist(retval); }'
```



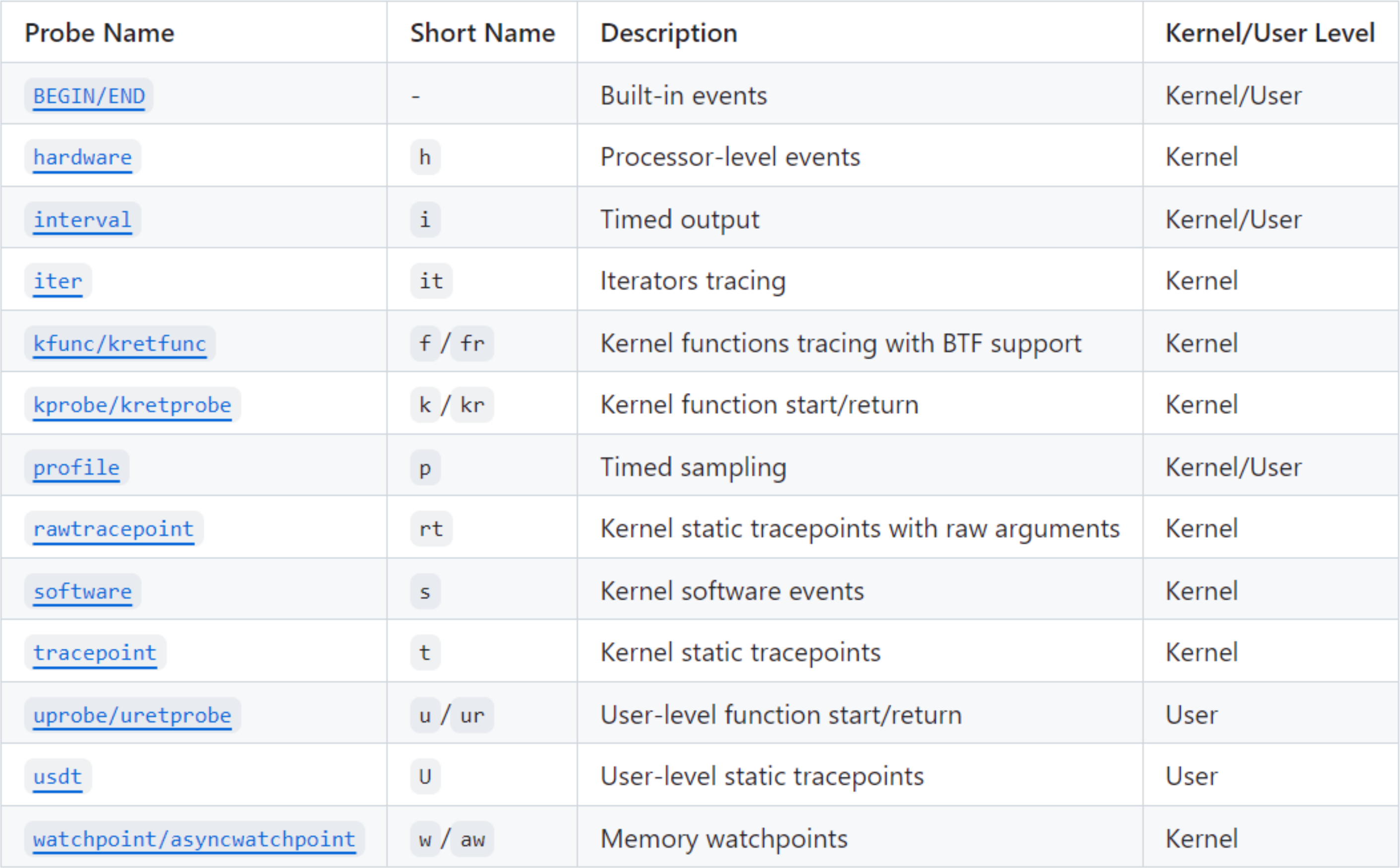
Probe

Filter (optional)

Action

https://github.com/bpftrace/bpftrace

bpftrace: Probe Type Shortcuts



Probe Name	Short Name	Description	Kernel/User Level
BEGIN/END	-	Built-in events	Kernel/User
hardware	h	Processor-level events	Kernel
interval	i	Timed output	Kernel/User
iter	it	Iterators tracing	Kernel
kfunc/kretfunc	f / fr	Kernel functions tracing with BTF support	Kernel
kprobe/kretprobe	k / kr	Kernel function start/return	Kernel
profile	p	Timed sampling	Kernel/User
rawtracepoint	rt	Kernel static tracepoints with raw arguments	Kernel
software	s	Kernel software events	Kernel
tracepoint	t	Kernel static tracepoints	Kernel
uprobe/uretprobe	u / ur	User-level function start/return	User
usdt	U	User-level static tracepoints	User
watchpoint/asyncwatchpoint	w / aw	Memory watchpoints	Kernel

https://github.com/bpftrace/bpftrace/blob/master/man/adoc/bpftrace.adoc#probes

bpftrace: 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
`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

bpftrace: Variable Types

Basic Variables

- **`@global`**
- **`@thread_local[tid]`**
- **`$scratch`**

Associative Arrays

- **`@array[key] = value`**

Buitins

- Integers: **`pid, tid, uid, cgroup, cpu, nsecs, arg0..N, retval, ...`**
- Strings: **`comm, func, probe`**
- Stacks: **`kstack, ustack`**
- Structs: **`args, curtask`**

bpftrace: Handy 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)) }’
...
```

Linux 4.9+, https://github.com/bpftrace/bpftrace

bpftrace tool: File system readahead

```
# readahead.bt
Attaching 5 probes...
^C
Readahead unused pages: 128
Readahead used page age (ms):
@age_ms:
[1]                 2455 |@@@@@@@@@@@@@@@                                     |
[2, 4)              8424 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4, 8)              4417 |@@@@@@@@@@@@@@@@@@@@@@@@@@@                         |
[8, 16)             7680 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     |
[16, 32)            4352 |@@@@@@@@@@@@@@@@@@@@@@@@@@                          |
[32, 64)               0 |                                                    |
[64, 128)              0 |                                                    |
[128, 256)           384 |@@                                                  |
#!/usr/local/bin/bpftrace

#include <linux/mm_types.h>

kprobe:__do_page_cache_readahead    { @in_readahead[tid] = 1; }
kretprobe:__do_page_cache_readahead { @in_readahead[tid] = 0; }

kretprobe:__page_cache_alloc
/@in_readahead[tid]/
{
        @birth[retval] = nsecs;
        @rapages++;
}

kprobe:mark_page_accessed
/@birth[arg0]/
{
        @age_ms = hist((nsecs - @birth[arg0]) / 1000000);
        delete(@birth[arg0]);
        @rapages--;
}

END
{
        printf("\nReadahead unused pages: %d\n", @rapages);
        printf("\nReadahead used page age (ms):\n");
        print(@age_ms); clear(@age_ms);
        clear(@birth); clear(@in_readahead); clear(@rapages);
}
```

Source:
https://www.brendangregg.com/Slides/LSFMM2019_BPF_Observability.pdf

Flame Graphs

Visualizes a collection of stack traces

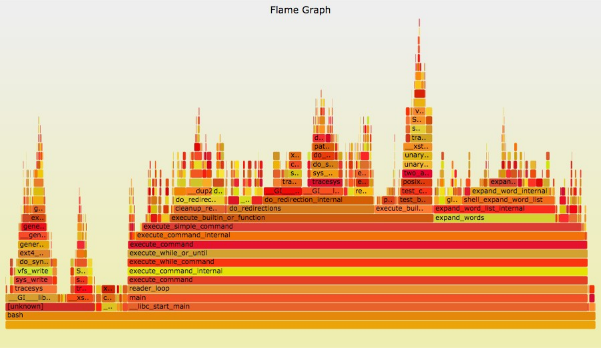
- **x-axis**: population: e.g., alphabetical sort to maximize merging
- **y-axis**: stack depth
- **color**: random (default) or a dimension

Over 80+ Implementations

- **https://github.com/brendangregg/FlameGraph:** Original, uses Perl + SVG + JavaScript
- https://github.com/spiermar/d3-flame-graph: By my Netflix colleague Martin Spier
- Linux perf now includes them: **`perf script flamegraph`**

References:

- http://www.brendangregg.com/flamegraphs.html
- http://queue.acm.org/detail.cfm?id=2927301
- "The Flame Graph" CACM, June 2016
- https://www.brendangregg.com/Slides/YOW2022_flame_graphs/



Flame Graph Instructions

Original (2011):

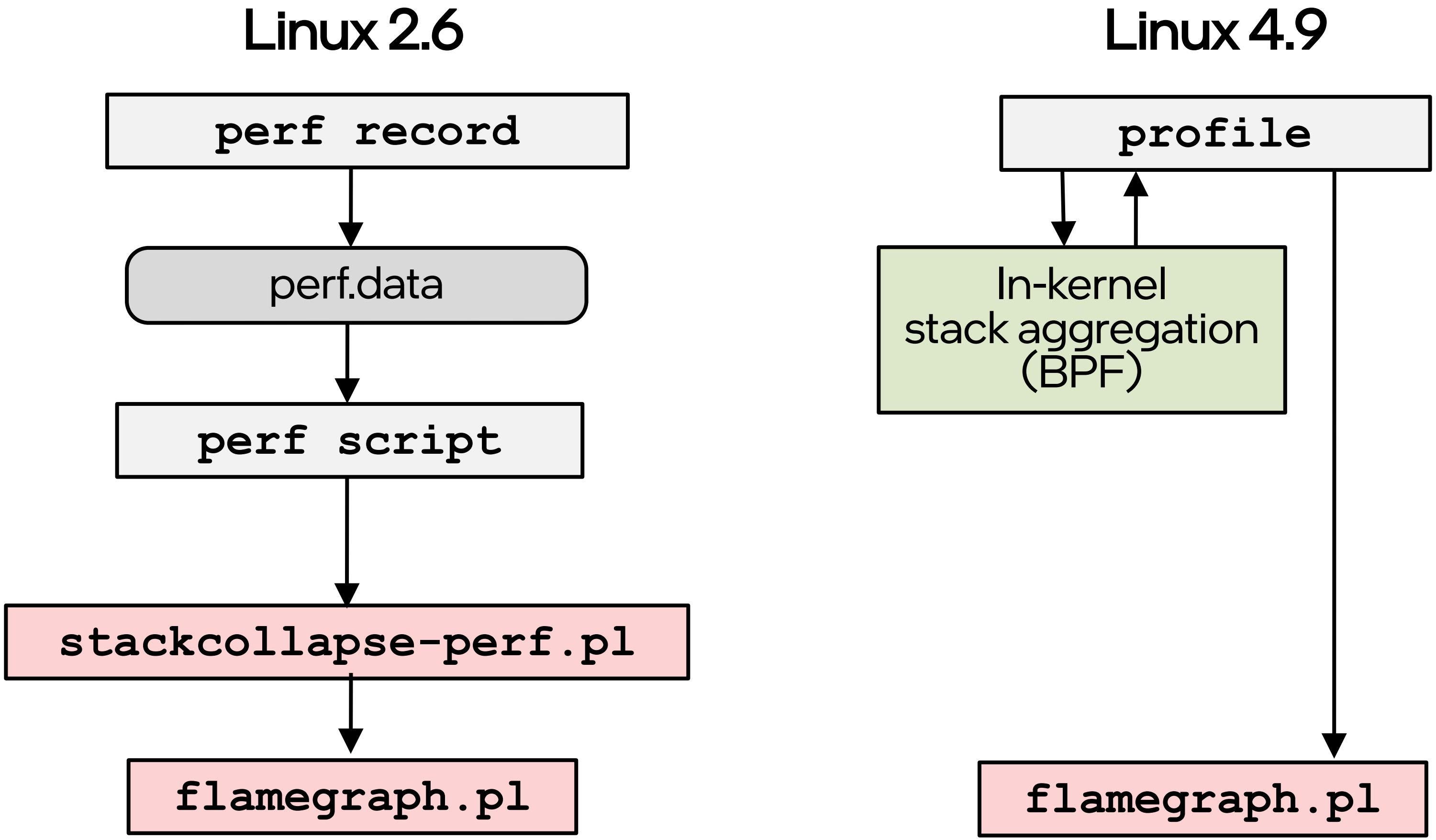
```
# git clone https://github.com/brendangregg/FlameGraph; cd FlameGraph
# perf record -F 49 -ag -- sleep 30
# perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > out.svg
```

eBPF:

```
# profile -af -F 49 30 | ./flamegraph.pl > out.svg
```

Some runtimes (e.g., JVM) require extra steps for stacks & symbols
see https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html

BPF-based CPU Flame Graphs (2017)



Agenda

1) What: A type of software
2) Why: Case Study
3) **How:** History, Internals, Usage, **Recommendations**
4) What's Next: Challenges, Future
5) Discussion & Q&A

Recommended tracing front-ends

I want to run some tools

- bcc, bpftrace

I want to hack up some *new* tools

- bpftrace

I want to spend weeks developing a BPF product

- bcc libbpf C, ~~bcc Python~~ (avoid), gobpf, libbpf-rs

New, lightweight,
CO-RE & BTF based

Requires LLVM;
now obsolete / special-use only

Recommended tracing front-ends

I want to run some tools

- bcc, bpftrace

I want to hack up some *new* tools

- bpftrace

I want to spend weeks developing a BPF product

- bcc libbpf C, ~~bcc Python~~ (avoid), gobpf, libbpf-rs

New, lightweight,
CO-RE & BTF based

Requires LLVM;
now obsolete / special-use only

Unix analogies

/usr/bin/*

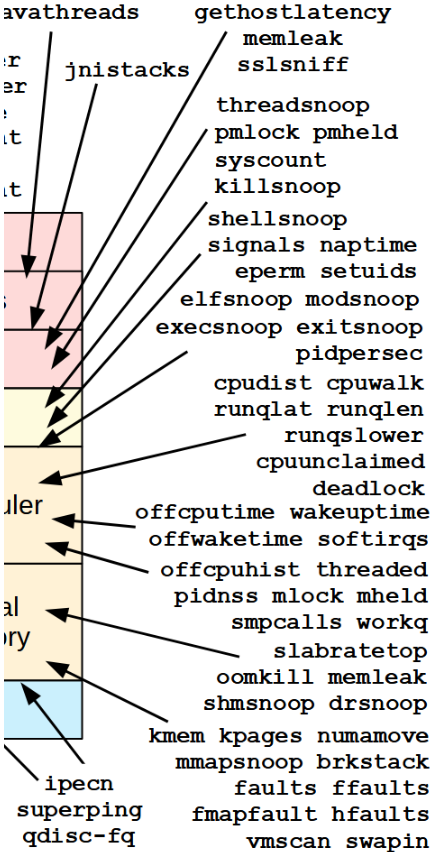
bash, awk

C, C++, Rust

Developing a new performance tool

1) Research the topic landscape
2) Create a known workload
3) Do one thing and do it well
4) Preference: tracepoints → kfunc → kprobe
5) Crosscheck measured numbers
6) Measure tool overhead
7) <80 chars wide by default
8) Add CLI options: follow other tool style
9) Concise, intuitive, self-explanatory output

From: https://github.com/iovisor/bcc/blob/master/CONTRIBUTING-SCRIPTS.md



Publishing a new performance tool

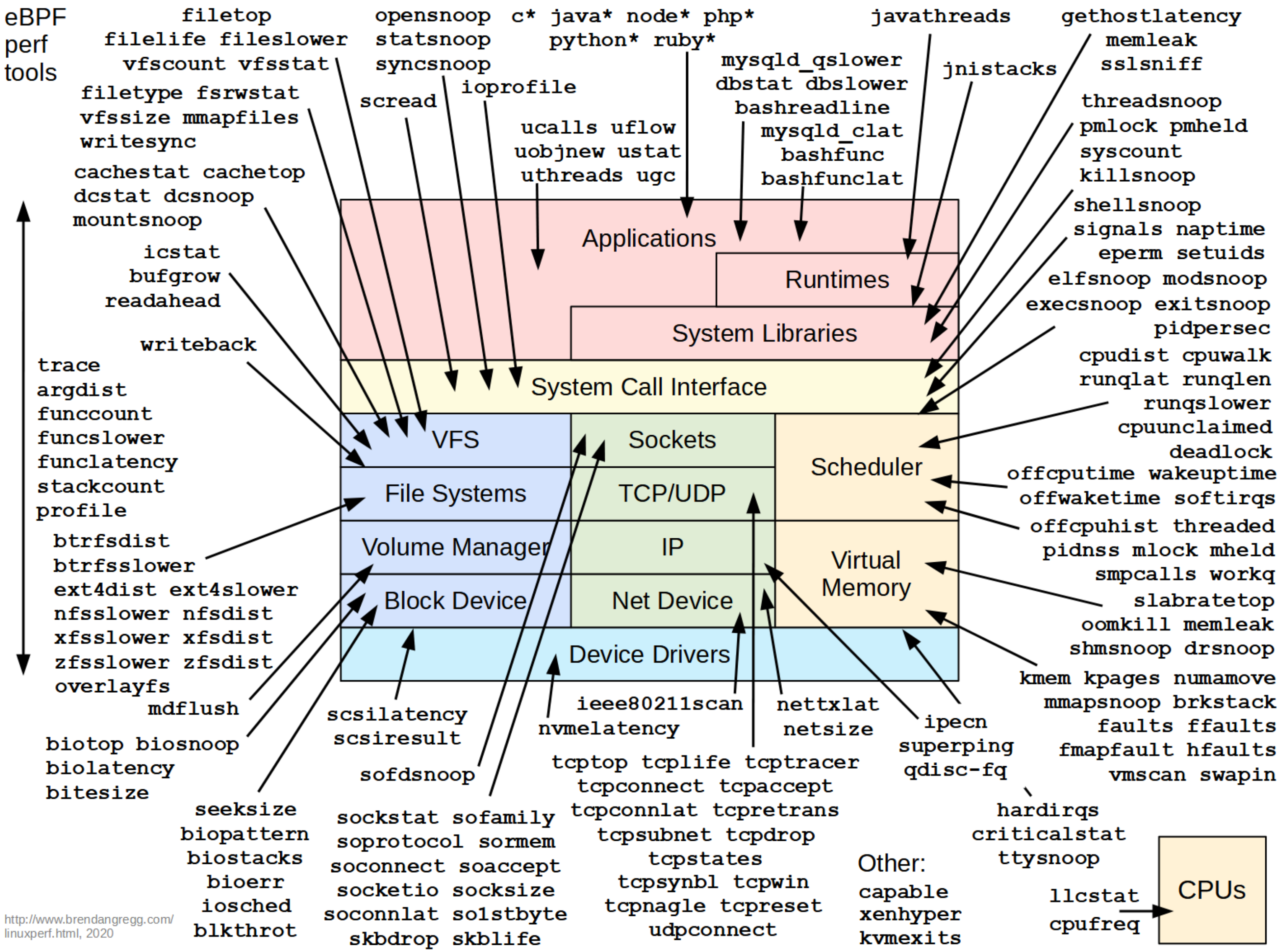
1) Publish to your own open source repo!
2) Publish to the community repo:
   - bpftrace: https://github.com/bpftrace/user-tools
   - bcc: <we haven't done it yet>
3) Promote in a blog post / conference
4) Gather feedback, iterate
5) Gather/share case studies
6) At this point the bcc/bpftrace maintainers may consider it for inclusion

We're trying to avoid having *too many* tools

Agenda

1) What: A type of software
2) Why: Case Study
3) How: History, Internals, Usage, Recommendations
4) **What's Next: Challenges**, Future
5) Discussion & Q&A

Too many tools



eBPF perf tools

filetop
filelife fileslower
vfscount vfsstat
filetype fsrwstat
vfssize mmapfiles
writesync
cachestat cachetop
dcstat dcsnoop
mountsnoop
icstat
bufgrow
readahead
writeback

opensnoop
statsnoop
syncsnoop
scread
ioprofile

c* java* node* php*
python* ruby*
ucalls uflow
uobjnew ustat
uthreads ugc
mysqld_qslower
dbstat dbslower
bashreadline
mysqld_clat
bashfunc
bashfunclat

javathreads
jnistacks

gethostlatency
memleak
sslsniff
threadsnoop
pmlock pmheld
syscount
killsnoop
shellsnoop
signals naptime
eperm setuids
elfsnoop modsnoop
execsnoop exitsnoop
pidpersec
cpudist cpuwalk
runqlat runqlen
runqslower
cpuunclaimed
deadlock
offcputime wakeuptime
offwaketime softirqs
offcpuhist threaded
pidnss mlock mheld
smpcalls workq
slabratetop
oomkill memleak
shmsnoop drsnoop
kmem kpages numamove
mmapsnoop brkstack
faults ffaults
fmapfault hfaults
vmscan swapin

Applications
Runtimes
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device
Net Device
Device Drivers

trace
argdist
funccount
funcslower
funclatency
stackcount
profile

btrfsdist
btrfsslower
ext4dist ext4slower
nfsslower nfsdist
xfsslower xfsdist
zfsslower zfsdist
overlayfs
mdflush

biotop biosnoop
biolatency
bitesize

seeksize
biopattern
biostacks
bioerr
iosched
blkthrot

scsilatency
scsiresult
sofdsnoop

sockstat sofamily
soprotocol sormem
soconnect soaccept
socketio socksize
soconnlat so1stbyte
skbdrop skblife

ieee80211scan
nvmelatency
nettxlat
netsize

tcptop tcplife tcptracer
tcpconnect tcpaccept
tcpconnlat tcpretrans
tcpsubnet tcpdrop
tcpstates
tcpsynbl tcpwin
tcpnagle tcpreset
udpconnect

ipecn
superping
qdisc-fq

hardirqs
criticalstat
ttysnoop

Other:
capable
xenhyper
kvmexits

llcstat
cpufreq
CPUs

http://www.brendangregg.com/ linuxperf.html, 2020

Challenges

Too many perf tools

BPF is hard

bcc tools are old

- They were designed in 2015 prior to tracepoint support and other features

uprobes are slow

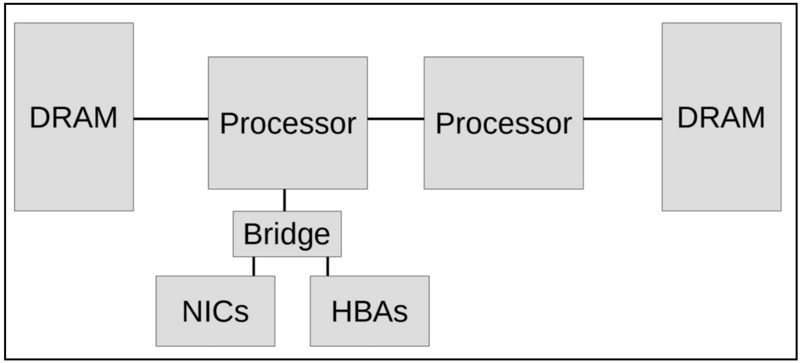
Symbols and stacks

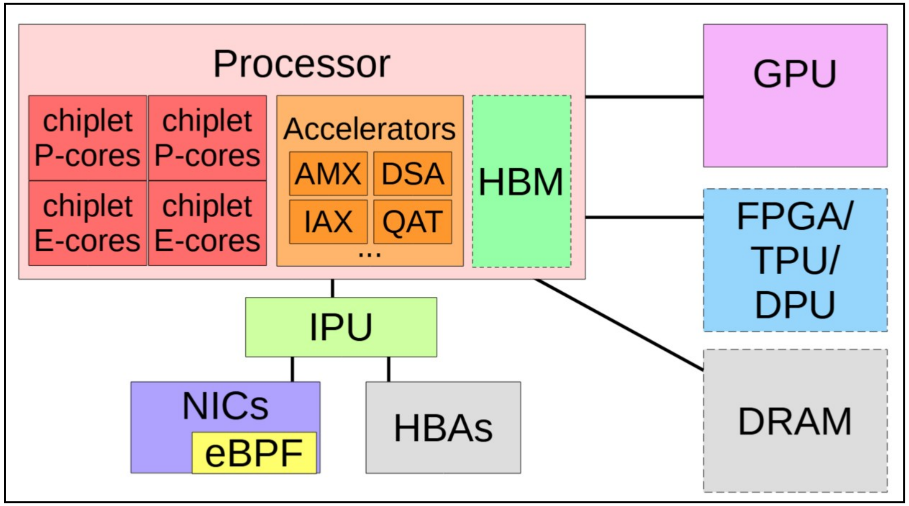
Signed or trusted BPF

Workloads are moving to GPU/AI accelerators

Performance is a moving target

Computers are getting increasingly complex





Just one example (computer hardware) of increasing complexity.
Software is worse!

Performance issues can now go **unsolved for weeks, months, years**

Product decisions **miss improvements** as analysis and tuning takes too long

Agenda

1) What: A type of software
2) Why: Case Study
3) How: History, Internals, Usage, Recommendations
4) **What's Next:** Challenges, **Future**
5) Discussion & Q&A

A vision:

"Fast by Friday":

Any computer performance issue

reported on Monday

should be solved by Friday

(or sooner)

Future

Fast by Friday

Off-CPU Flame Graphs (adoption, given frame pointers now in distros)

Zero-Instrumentation APM

Fast uprobes

Custom kernel algorithms (scheduling, networking, memory)

eBPF accelerators (including HW offload)

"Fast by Friday": Proposed Agenda

Prior weeks:	**Preparation**
Monday:	**Quantify, static tuning, load**
Tuesday:	**Checklists, elimination**
Wednesday:	**Profiling**
Thursday:	**Latency, logs, critical path**
Friday:	**Efficiency, algorithms**
Post weeks:	**Case study, retrospective**

"Fast by Friday": Proposed Agenda

Prior weeks:	**Preparation**
Monday:	**Quantify, static tuning, load**
Tuesday:	**Checklists, elimination**
Wednesday:	**Profiling**
Thursday:	**Latency, logs, critical path**
Friday:	**Efficiency, algorithms**
Post weeks:	**Case study, retrospective**

eBPF:

- → **Exoneration tools**
- → **CPU & off-CPU profiling**
- → **Latency drill downs**

The following are the eBPF excerpts from a full talk:
https://www.brendangregg.com/Slides/KernelRecipes2023_FastByFriday/

Prior weeks: **Preparation**

Everything must work on Monday!

- Critical analysis tools ("crisis tools") must be preinstalled; E.g., Linux: `procps, sysstat, linux-tools-common, bcc-tools, bpftrace, …`
- **Stack tracing and symbols** should work for the kernel, libraries, and applications
- Tracing (host & distributed) must work
- The performance engineers must already have host **SSH root access**
- A functional diagram of the system must be known
- Source code should be available

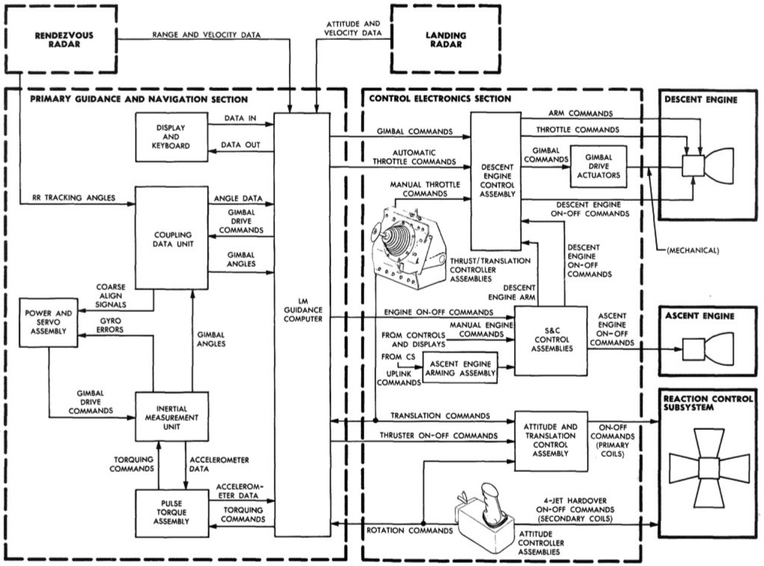


Figure 3-2.4. Primary Guidance Path - Simplified Block Diagram

Example functional diagram
Source: Lunar Module - LM10 Through LM14 Familiarization Manual" (1969):

Current industry status: 1 out of 5

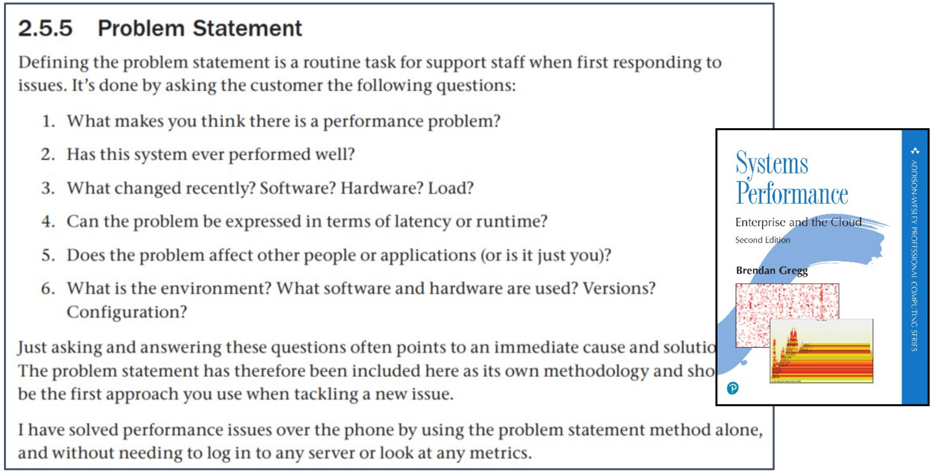
Monday: **Quantify, static tuning, load**

1) Quantify the problem
- Problem statement method

2) Static performance tuning
- The system without load
- Check all hardware, software
- Versions, past errors, config
- Covered in sysperf

3) Load vs implementation
- Just a problem of load?
- Usually solved via basic monitoring and line charts

Current industry status: 4 out of 5



2.5.5 Problem Statement

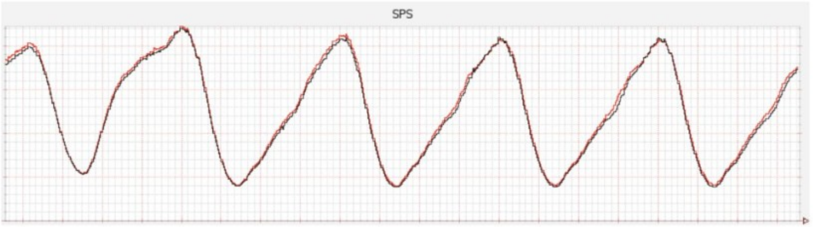
Defining the problem statement is a routine task for support staff when first responding to issues. It's done by asking the customer the following questions:

1. What makes you think there is a performance problem?
2. Has this system ever performed well?
3. What changed recently? Software? Hardware? Load?
4. Can the problem be expressed in terms of latency or runtime?
5. Does the problem affect other people or applications (or is it just you)?
6. What is the environment? What software and hardware are used? Versions? Configuration?

Just asking and answering these questions often points to an immediate cause and solutio The problem statement has therefore been included here as its own methodology and sho be the first approach you use when tackling a new issue.

I have solved performance issues over the phone by using the problem statement method alone, and without needing to log in to any server or look at any metrics.

Problem Statement method
Source: Systems Performance 2nd edition, page 44



A familiar pattern of load
Source: https://www.brendangregg.com/Slides/SREcon_2016_perf_checklists

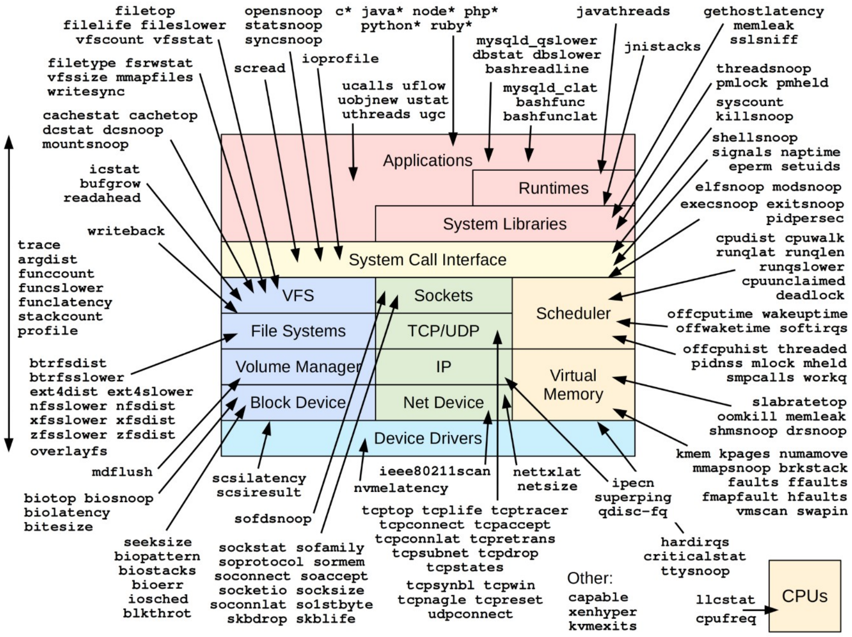
Tuesday: Checklists, Elimination

Current eBPF tools

- ***snoop, *top, *stat, *count, *slower, *dist**
- Supports later methodologies
  - Workload characterization, latency analysis, off-CPU analysis, USE method, etc.

Future elimination tools

- ***health, *diagnosis**
- Supports "fast by friday"
- Analyzes existing dynamic workload
- Open source & in the target code repo (same as tests)



Current eBPF performance tools

Source: BPF Performance Tools, cover art [Gregg 2019]

Wednesday: **Profiling**

1) CPU Flame Graphs

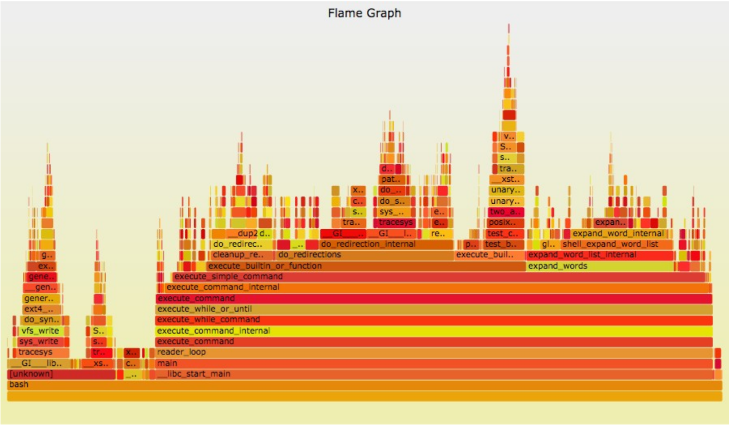
- More efficient with eBPF
- eBPF runtime stack walkers

2) Off-CPU Flame Graphs

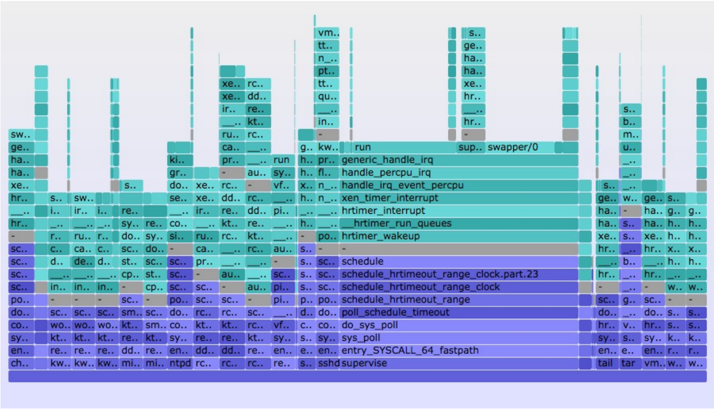
- Impractical without eBPF

Solves most performance issues

Needs prep! (stack walking and symbols)



CPU flame graph



Off-CPU/waker time flame graph

Thursday: **Latency, logs, critical path**

1) Latency drilldowns

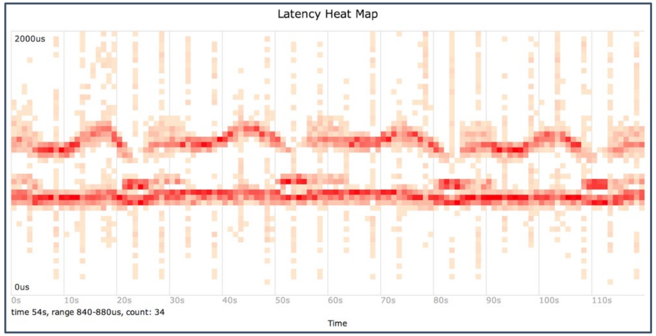
- Latency histograms
- Latency heat maps
- Latency outliers
- Drill down to origin of latency

2) Logs, event tracing

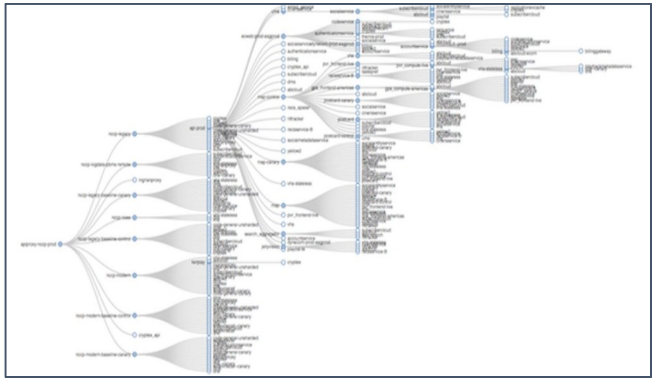
- Custom event logs

3) Critical path analysis

- Multi-threaded tracing
- Distributed tracing across a distributed environment



Latency heat maps
Source: https://www.brendangregg.com/HeatMaps/latency.html



Distributed tracing
Source:
https://www.brendangregg.com/Slides/Monitorama2015_NetflixInstanceAnalysis

Thursday: **Latency, logs, critical path**

eBPF Tools

1) Latency drilldowns

- Latency histograms
- Latency heat maps
- Latency outliers
- Drill down to origin of latency

***dist**

***slower**

2) Logs, event tracing

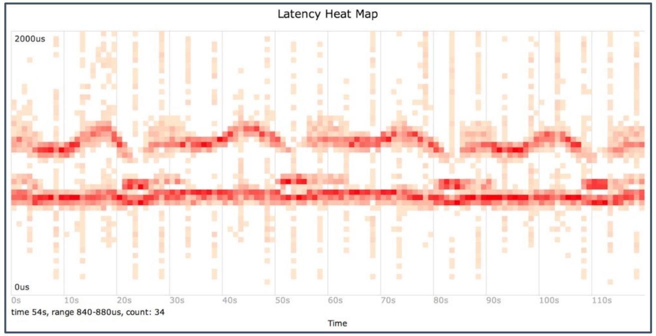
- Custom event logs

***snoop, bpftrace**

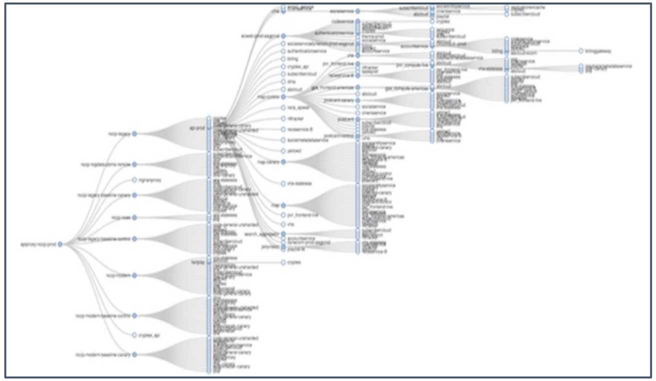
3) Critical path analysis

- Multi-threaded tracing
- Distributed tracing across a distributed environment

"Zero instrumentation"
(when faster uprobes is done)



Latency heat maps
Source: https://www.brendangregg.com/HeatMaps/latency.html



Distributed tracing
Source:
https://www.brendangregg.com/Slides/Monitorama2015_NetflixInstanceAnalysis

Friday: Efficiency, algorithms

1) Is the target *efficient*?

- A largely unsolved problem
- Cycles/carbon per request
- Compare with similar products
- **New efficiency tools (eBPF?)**
- System efficiency equals the least efficient component
- Modeling, theory

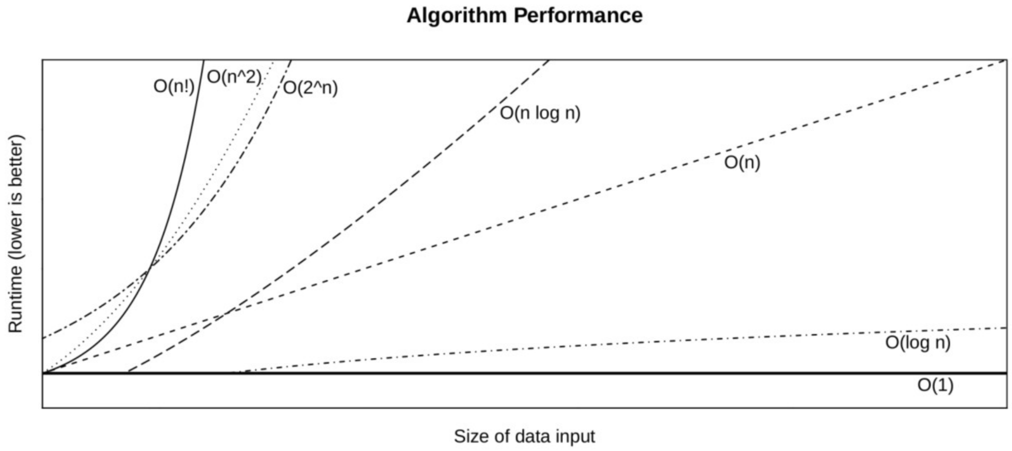
1) Use faster algorithms?

- Big O Notation

Current industry status: 1 out of 5

Protocol	CIFS	iSCSI	FTP	NFSv3	NFSv4
Cycles(k) per 1k read	2241	1843	970	395	485

Example efficiency comparisons (made up)



Source: Systems Performance 2nd Edition, page 175

"Fast by Friday": Summary

Any computer performance issue reported on Monday should be solved by Friday (or sooner)

Prior weeks:	**Preparation**
Monday:	**Quantify, static tuning, load**
Tuesday:	**Checklists, elimination**
Wednesday:	**Profiling**
Thursday:	**Latency, logs, critical path**
Friday:	**Efficiency, algorithms**
Post weeks:	**Case study, retrospective**

eBPF:

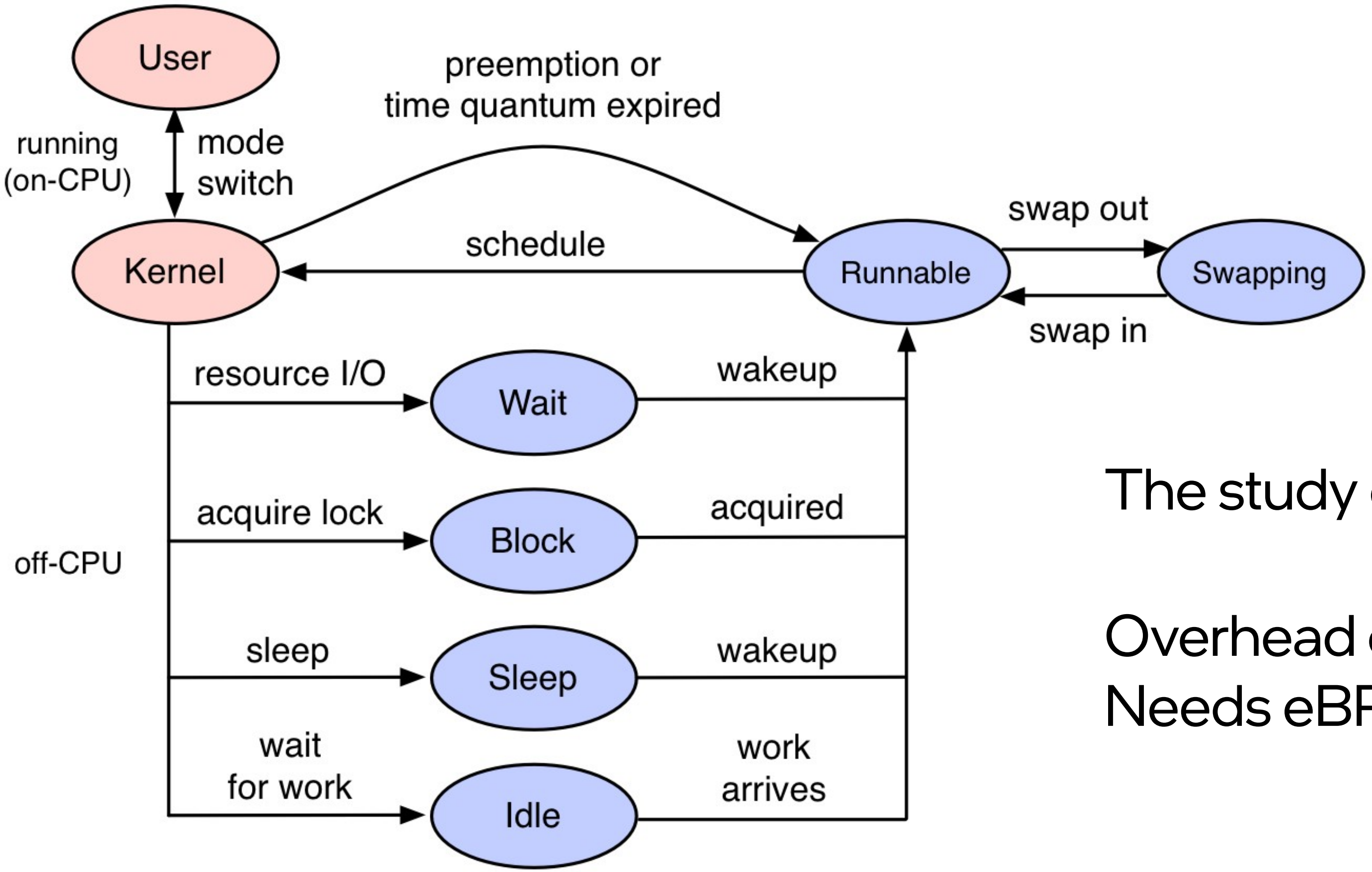
→ **Exoneration tools**

→ **CPU & off-CPU profiling**

→ **Latency drill downs**

More details: https://www.brendangregg.com/Slides/KernelRecipes2023_FastByFriday/

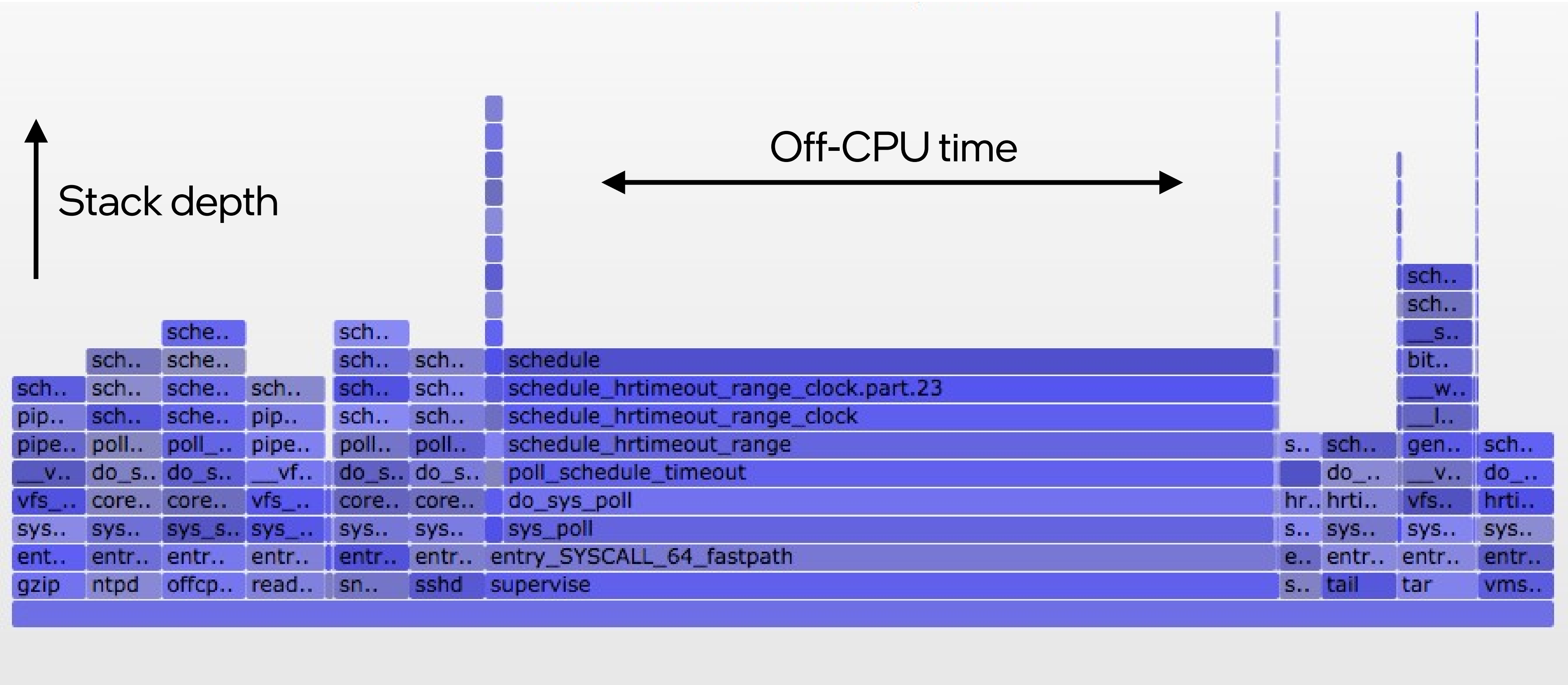
Off-CPU Analysis



The study of blocking states

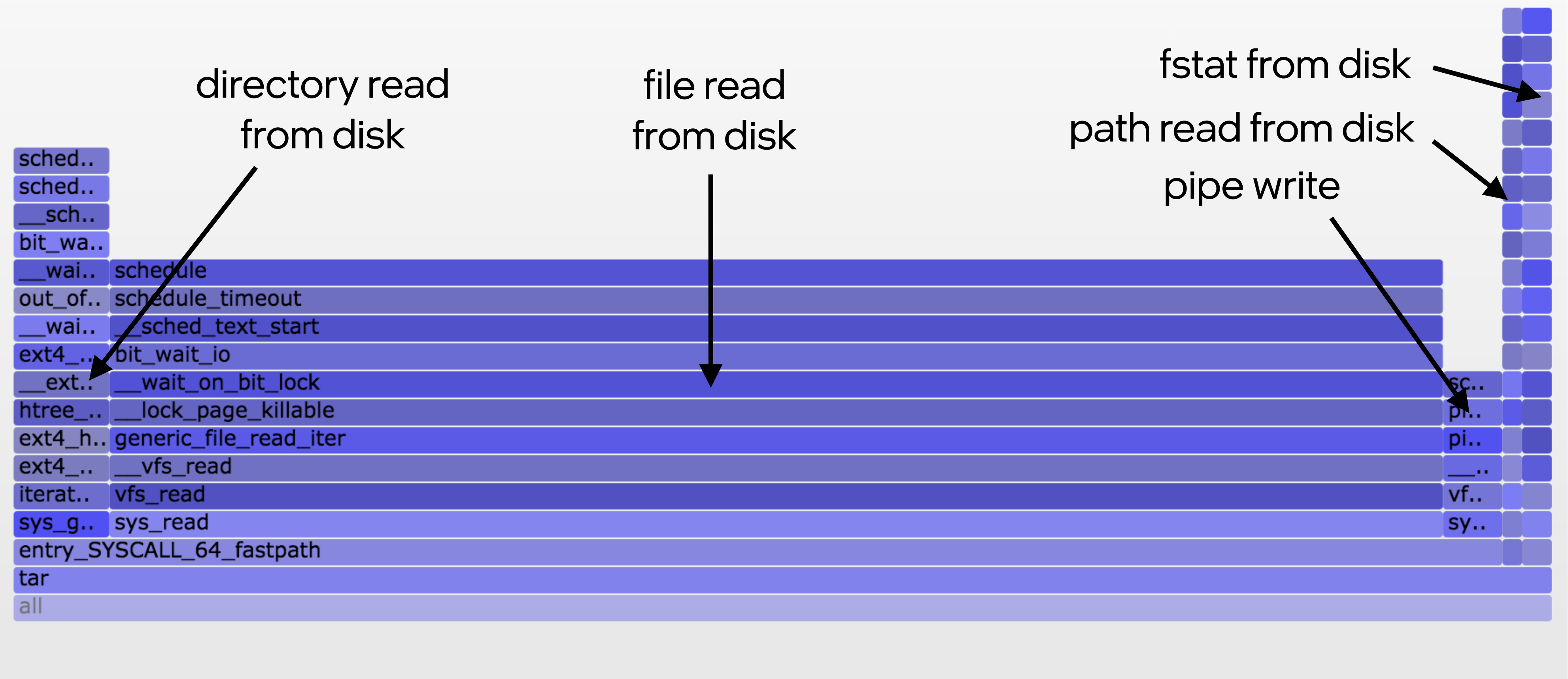
Overhead often prohibitive
Needs eBPF

Off-CPU Time Flame Graph



http://www.brendangregg.com/blog/2016-02-01/linux-wakeup-offwake-profiling.html

Off-CPU Time (zoomed): tar(1)



directory read from disk

file read from disk

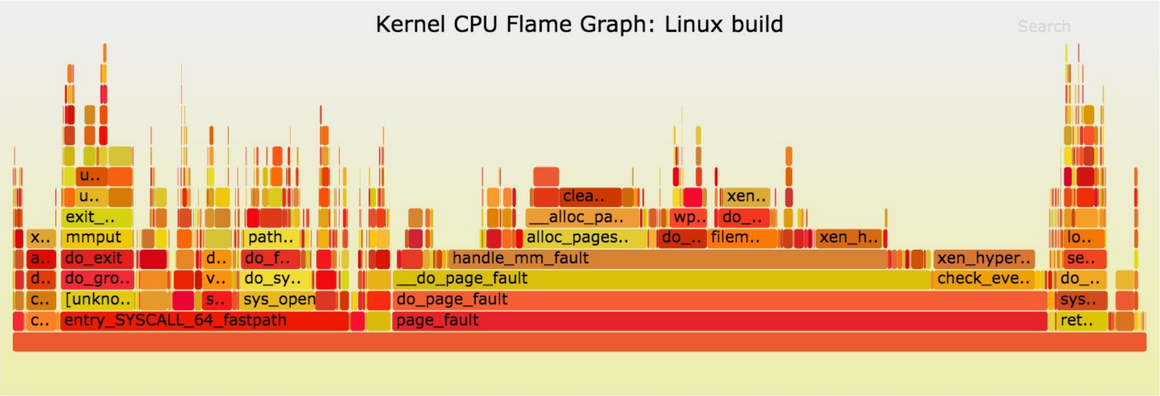
fstat from disk

path read from disk

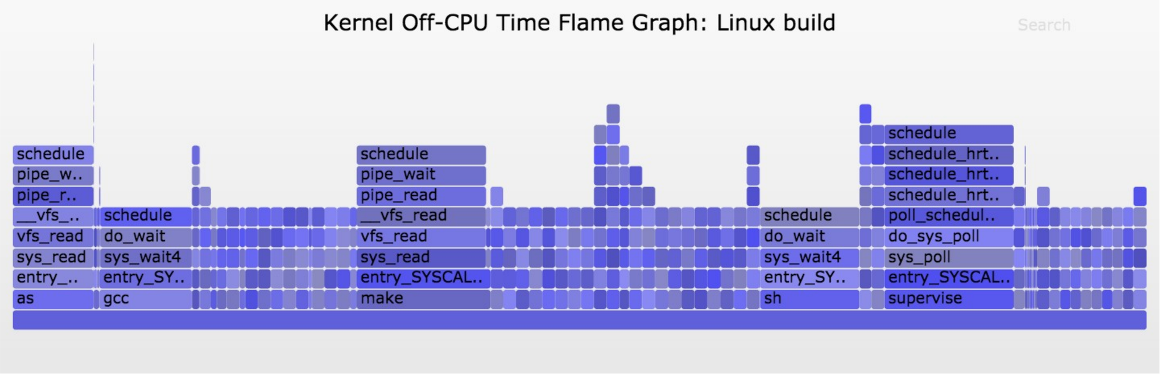
pipe write

Only showing kernel stacks in this example

CPU + Off-CPU Flame Graphs: See Everything



CPU



Off-CPU

eBPF Flame Graph Futures

Practical off-CPU flame graphs

- Much easier now that frame pointers are default in Ubuntu, Fedora, etc. (2024)

Other types: disk, network, malloc, etc.

Custom stack walking

- Frame pointers not needed: SFrames, shadow stacks
- Include other app context

Zero-instrumentation APM

(Application Performance Monitoring)

Installation:

1) Install the agent
2) Done! (no code changes required)

Uses uprobes to instrument HTTP/SSL calls

Multiple startups will be selling this

Possible headline: "OpenTelemetry more stable *and faster*"

- This gives uprobes/eBPF a bad name, unfairly, as none of us in uprobe/eBPF land recommend this use case until the speed/stability issues are fixed

Fast uprobes available in Linux in ~~2024~~ 2025?

Custom Kernel Algorithms (for performance)

~~TCP congestion controls~~ → already done via STRUCT_OPS, also see "TCP's Third Eye" paper (https://schmiste.github.io/ebpf23.pdf, SIGCOMM 2023)

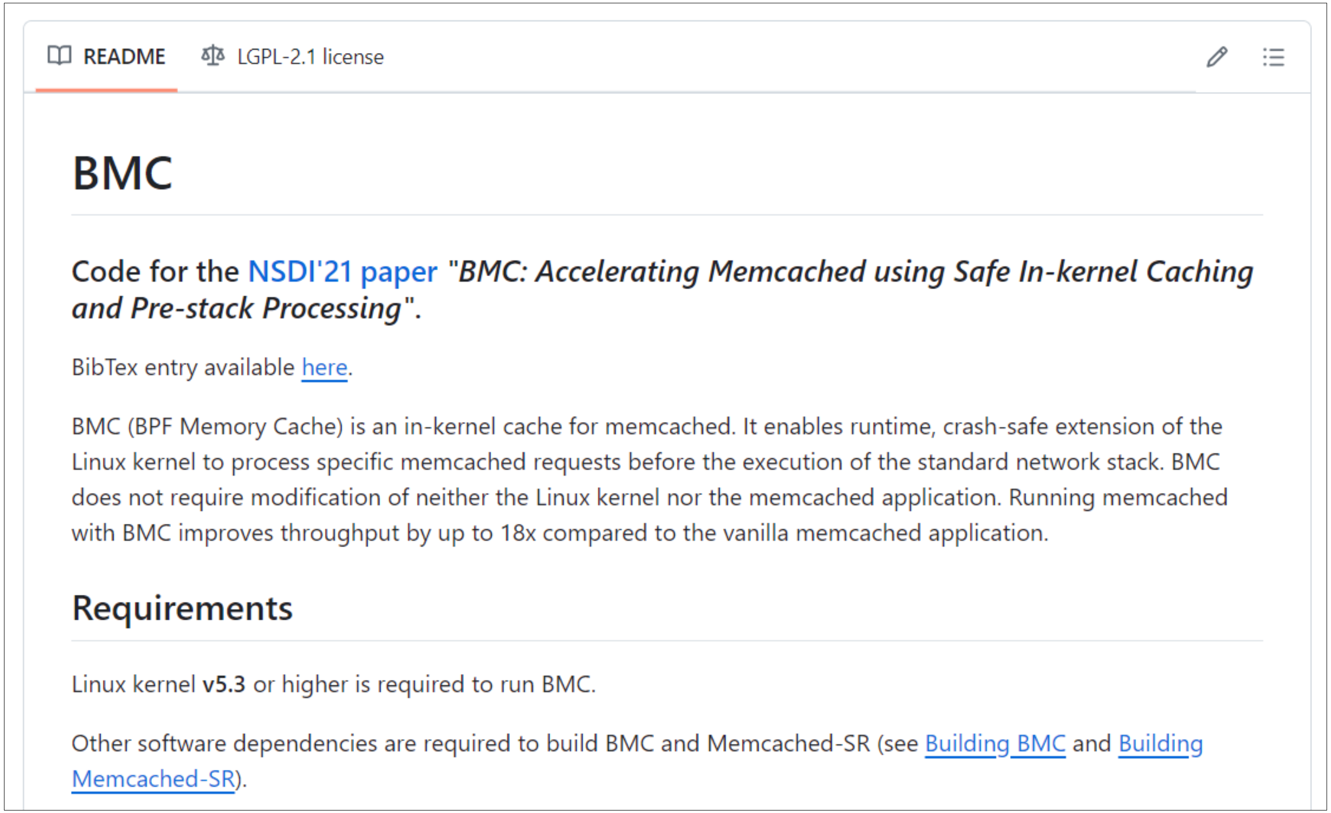
~~CPU & container schedulers~~ → already done: sched_ext

- On most generic systems I don't forsee huge utilization wins; we will see tail-latency wins, and some wins for complex scheduling needs (Beowulf clusters; P/E-core?; Contaniers/cgroups).

FS readahead policies

eBPF Accelerators

First proof of concept:



BMC

Code for the NSDI'21 paper "*BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing*".

BibTex entry available here.

BMC (BPF Memory Cache) is an in-kernel cache for memcached. It enables runtime, crash-safe extension of the Linux kernel to process specific memcached requests before the execution of the standard network stack. BMC does not require modification of neither the Linux kernel nor the memcached application. Running memcached with BMC improves throughput by up to 18x compared to the vanilla memcached application.

Requirements

Linux kernel **v5.3** or higher is required to run BMC.

Other software dependencies are required to build BMC and Memcached-SR (see Building BMC and Building Memcached-SR).

https://github.com/Orange-OpenSource/bmc-cache

Other Future Predictions

More device drivers, incl. USB on BPF (ghk)

Performance monitoring agents

~~Intrusion detection systems~~ → already seeing adoption

Runtimes come with eBPF accelerators

- `java -XX:+eBPF`

New Windows eBPF things we haven't thought of yet

What would you like to imagine?

There's a good chance it can be built using eBPF and used in production today.

Measure first (observability/tracing) to prove and quantify a problem, then build/offer the solution armed with expected speedups based on your measurements.

Agenda

1) What: A type of software
2) Why: Case Study
3) How: History, Internals, Usage, Recommendations
4) What's Next: Challenges, Future
5) **Discussion & Q&A**

Discussion and Q&A

eBPF References/URLs

- https://ebpf.io
- https://github.com/iovisor/bcc
- https://github.com/bpftrace/bpftrace
- https://www.brendangregg.com/ebpf.html
- https://lwn.net/Kernel/Index/#BPF
- https://docs.cilium.io/en/v1.15/bpf/
- https://ebpf.io/what-is-ebpf
- Documentary: https://www.youtube.com/watch?v=Wb_vD3XZYOA
- Intel iwl tracing demo: https://www.youtube.com/watch?v=16slh29iN1g
- http://www.brendangregg.com/flamegraphs.html
- https://www.brendangregg.com/Slides/KernelRecipes2023_FastByFriday/

Thanks

BPF: Alexei Starovoitov (Meta), Daniel Borkmann (Isovalent/Cisco), David S. Miller (Red Hat), Jakub Kicinski (Meta), Yonghong Song (Meta), Martin KaFai Lau (Meta), John Fastabend (Isovalent), Quentin Monnet (Isovalent), Jesper Dangaard Brouer (Isovalent), Andrey Ignatov (Meta), Stanislav Fomichev (Google), Linus Torvalds, and many more in the BPF community

BCC: Brenden Blanco (VMware), Yonghong Song, Sasha Goldsthein (Google), Teng Qin (Meta), Paul Chaignon (Isovalent), Vicent Martí (PlanetScale), Dave Marchevsky (Meta), Hengqi Chen (Tencent), and many more in the BCC community

bpftrace: Alastair Robertson (Meta), Dan Xu (Meta), Bas Smit, Mary Marchini (Netflix), Masanori Misono, Jiri Olsa, Viktor Malík, Dale Hamel, Willian Gaspar, Augusto Mecking Caringi, and many more in the bpftrace community

Canonical Ubuntu: BPF support, frame pointers by default, bcc and bpftrace by default

brendan@intel.com

All photos my own; except slide 32 (DockerCon), 35 (KernelRecipes) and 36 (UbuntuMasters)