VIENNA 17.09.2022

MEASURING PERFORMANCE OVERHEAD **OF** *DTrace & eBPF*

MATEUSZ PIOTROWSKI

WHAT *OVERHEAD* DOES TRACING IMPOSE ON A SYSTEM?

Outline

About Me

Observability

Introduction to Observability

DTrace

eBPF

The Art of Benchmarking

Experiments

Experiment A

Experiment B

Analysis

Conclusions & Future Work

Who am I?

FreeBSD user since 2016

FreeBSD committer since 2018

FreeBSD core team member since 2022

Student @ Technische Universität Berlin

Working with folks @ Klara Inc

observability

Observability

We like to know what is going on in our systems.

Why do we need it?

- Unusually high memory consumption after an upgrade?
- Maybe the CPUs is busy doing things it does not need to be doing?
- Maybe you want see what kind of IO goes to and from the disks,
  why the performance is not as good as advertised?

instrumentation

Static vs Dynamic Instrumentation

Static instrumentation:

- Compiled-in
- Always present
- Potentially non-negligible overhead

Dynamic instrumentation:

- Activated when needed
- Low overhead
- More flexible

Tracing vs. Sampling

Tracing:

- Collects statistics on specific events
- Generally, does not miss events

Sampling:

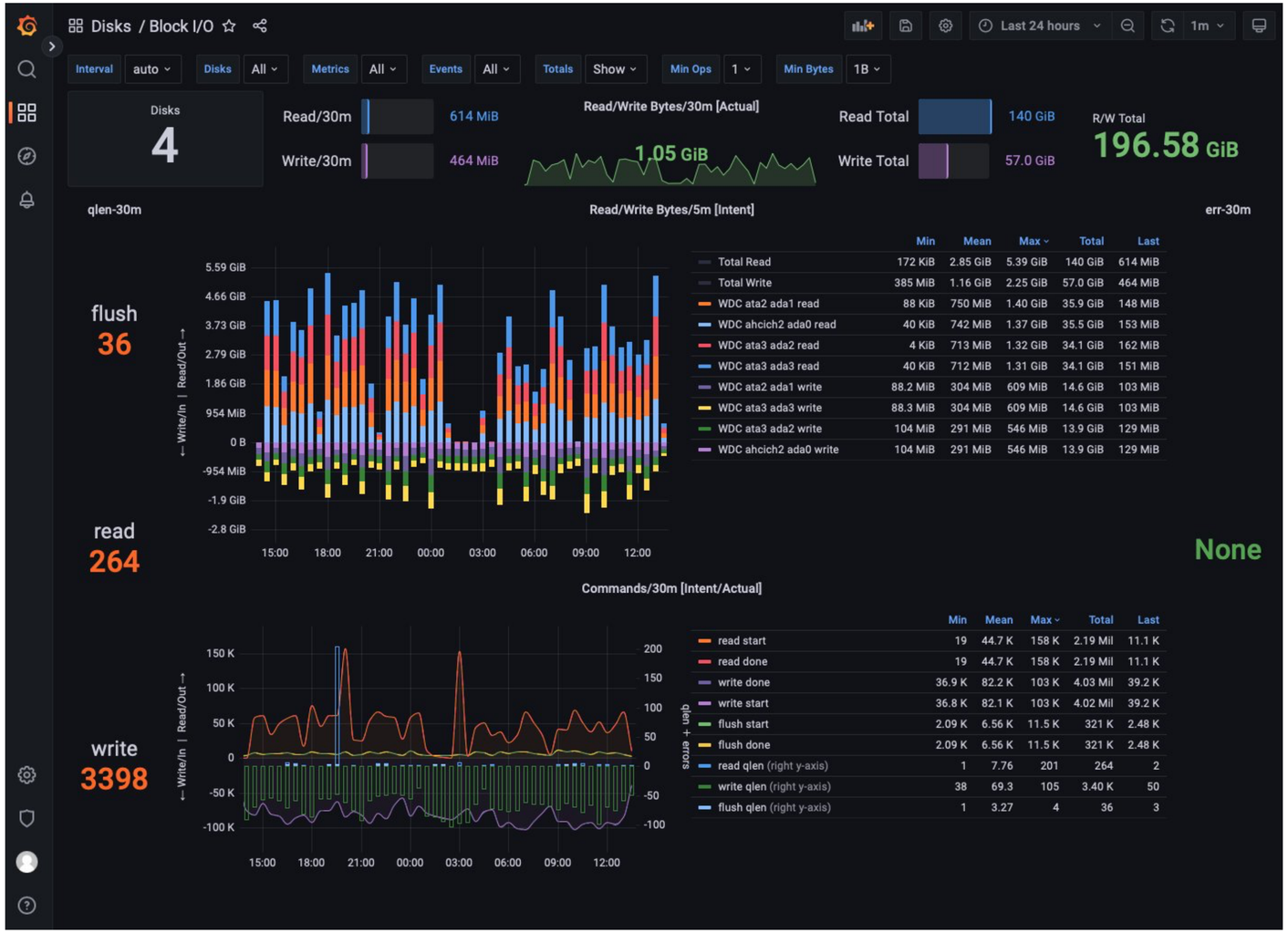
- Collects statistics periodically
- Suitable for profiling (and flame graphs)

examples?

Debugging

```
root@freebsd ~ # dwatch -X proc -k sleep

INFO Sourcing proc profile [found in /usr/libexec/dwatch]

INFO Watching 'proc:::create, proc:::exec, proc:::exec-failure, proc:::exec-success, proc:::exit,
proc:::signal-clear, proc:::signal-discard, proc:::signal-send' ...

INFO Setting execname: sleep

2022 Sep 16 00:23:35 1434078666.1434078666 sleep[16966]: INIT sleep 50

2022 Sep 16 00:23:36 1434078666.1434078666 sleep[16966]: EXIT child terminated abnormally

2022 Sep 16 00:23:36 1434078666.1434078666 sleep[16966]: SEND SIGCHLD[20] pid 16874 -- -bash
```

Monitoring



Source: https://twitter.com/freebsdfrau/status/1562905979489902592

isn't it slow?

Probe effect ...

... is unintended alteration in system behavior caused by measuring that system.

Source: https://en.wikipedia.org/wiki/Probe_effect

DTrace

DTrace: Introduction

A dynamic tracing framework:

- Userland tooling and libraries
- Scripting language
- Kernel module with core functionalities
- Deeply integrated with a kernel

Platforms: FreeBSD (7.1-RELEASE, 2009), illumos, Linux,
macOS (Mac OS X Leopard, 2007), NetBSD, Windows (2019)

DTrace: Architecture Overview

OpenDTrace Architecture

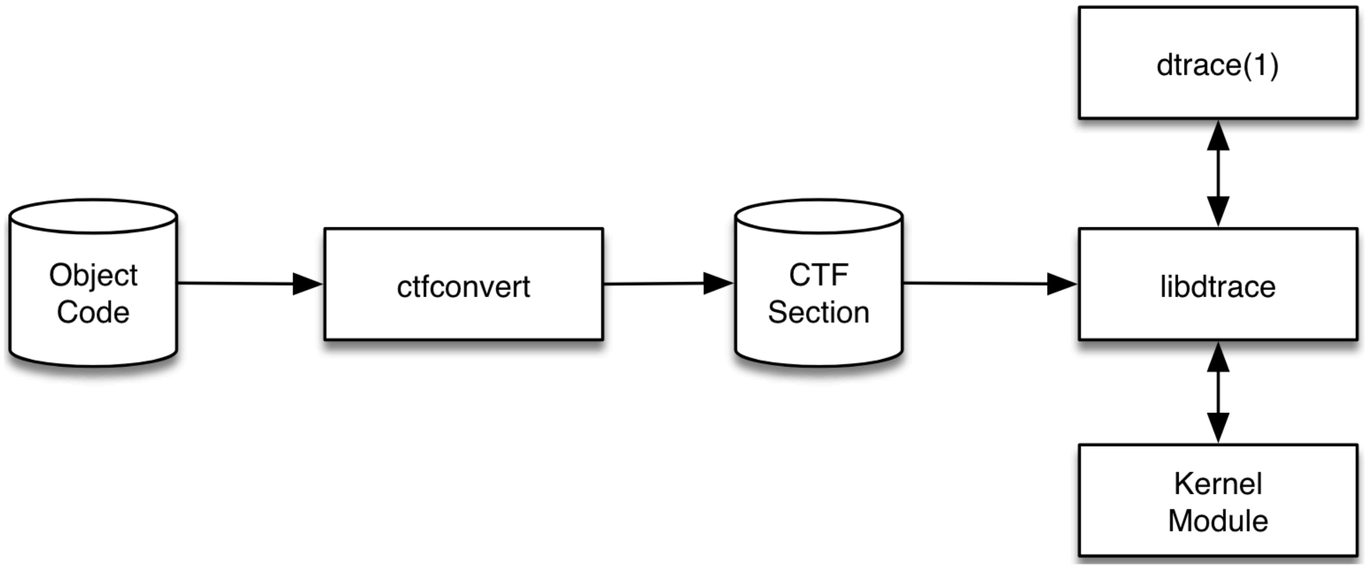


Figure 2.1: OpenDTrace Components

Source: OpenDTrace Specification version 1.0

DTrace: One-Liner Tutorial

1. Listing Probes

```
dtrace -l | grep 'syscall.*read'
```

2. Hello World

```
dtrace -n 'dtrace:::BEGIN { printf("Hello FreeBSD!\n"); }'
```

3. File Opens

```
dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
```

4. Syscall Counts By Process

```
dtrace -n 'syscall:::entry { @[execname, probefunc] = count(); }'
```

5. Distribution of read() Bytes

```
dtrace -n 'syscall::read:return /execname == "sshd"/ { @ = quantize(arg0); }'
```

Source: https://wiki.freebsd.org/DTrace/Tutorial

DTrace: One-Liner Tutorial

```
# dtrace -n '
    syscall::read:return        <-- Probe ([[[provider:] module:] function:] name)
    /execname == "sshd"/        <-- Predicate
    {                           <-- Action
        @ = quantize(arg0);     <-- Aggregation
    }
'
dtrace: description 'syscall::read:return ' matched 2 probes
^C
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@                     2
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@                     2
              64 |                                         0
```

eBPF

eBPF: Introduction

Based on BPF (Berkeley Packet Filter).

Very similar to DTrace, yet completely different.

Platforms: Linux, Windows

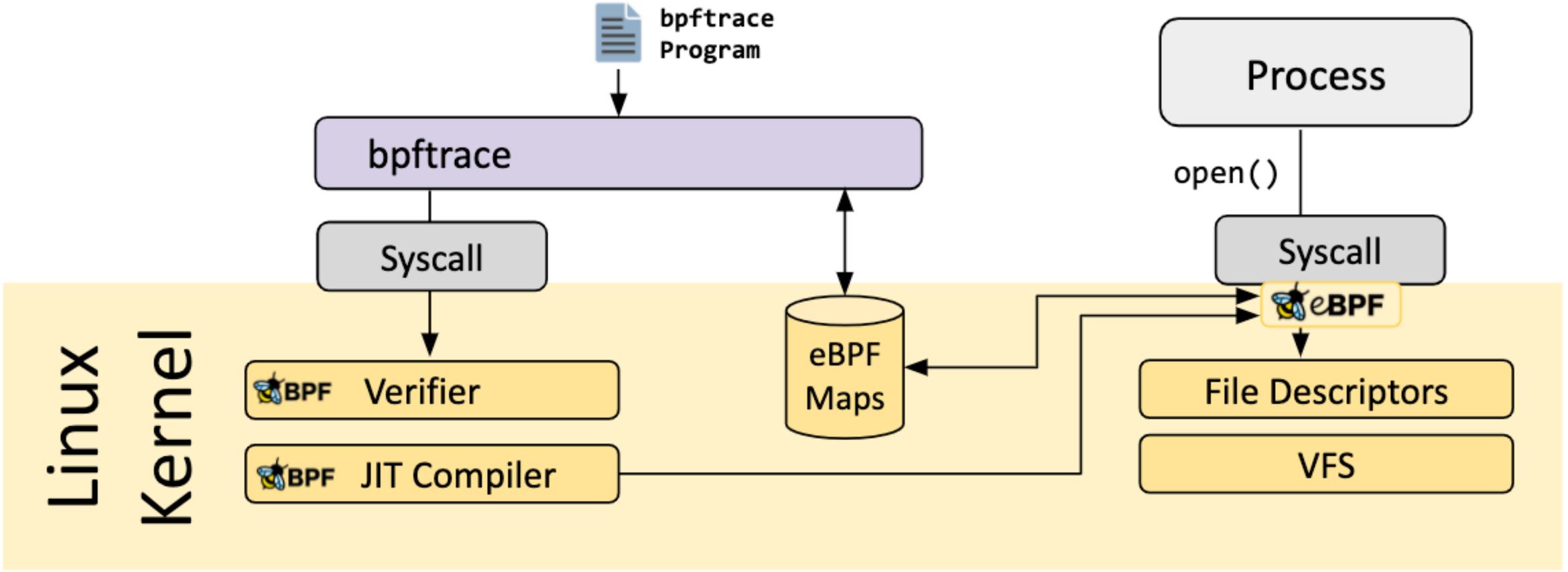
eBPF: Frontends

Writing eBPF byte code by hand is not easy.

There are many frontends to eBPF:

- BCC (BPF Compiler Collection)
  - C with some Python/Lua glue
- bpftrace
  - DTrace/AWK-inspired scripting

eBPF: Architecture Overview



Source: https://ebpf.io/what-is-ebpf/

bpftrace: One-Liner Tutorial

1. Listing Probes

```
bpftrace -l 'tracepoint:syscalls:sys_enter_*'
```

2. Hello World

```
bpftrace -e 'BEGIN { printf("hello world\n"); }'
```

3. File Opens

```
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args->filename)); }'
```

4. Syscall Counts By Process

```
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
```

5. Distribution of read() Bytes

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

Source: https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners.md

The Art of Benchmarking

Benchmarking: Computer Resources

- CPU
- Memory
- Disk
- Network

Benchmarking: Workload Generators

- dd
- Bonnie
- fio
- TPC-C

Benchmarking: Checklist

- Why not double?
- Was it tuned?
- Did it break limits?
- Did it error?
- Does it reproduce?
- Does it matter?
- Did it even happen?

Source: https://www.brendangregg.com/blog/2018-06-30/benchmarking-checklist.html

Benchmarking: Performance Measurement Tools

Examples from Understanding Software Dynamics by Richard Sites:

- blktrace can have 5% CPU overhead
- mtrace can slow down a program by 1200%
- Richard Sites' tracing tool, called KUtrace, aims for less than 1%

Experiments

Experiments

- Experiment 1
  - dd(1)
  - small Azure VM
- Experiment 2
  - dd(1)
  - large bare metal server

1

Experiment 1: Setup

Hardware:

- VMs (Standard_DS1_v2, 1 vCPU, 3.5 GB RAM)
- FreeBSD 13.0, Ubuntu 20.04

Workload:

- 10 million 1 byte blocks transferred with dd(1) from /dev/zero to /dev/null

Tracing:

- Beginning of the read syscall

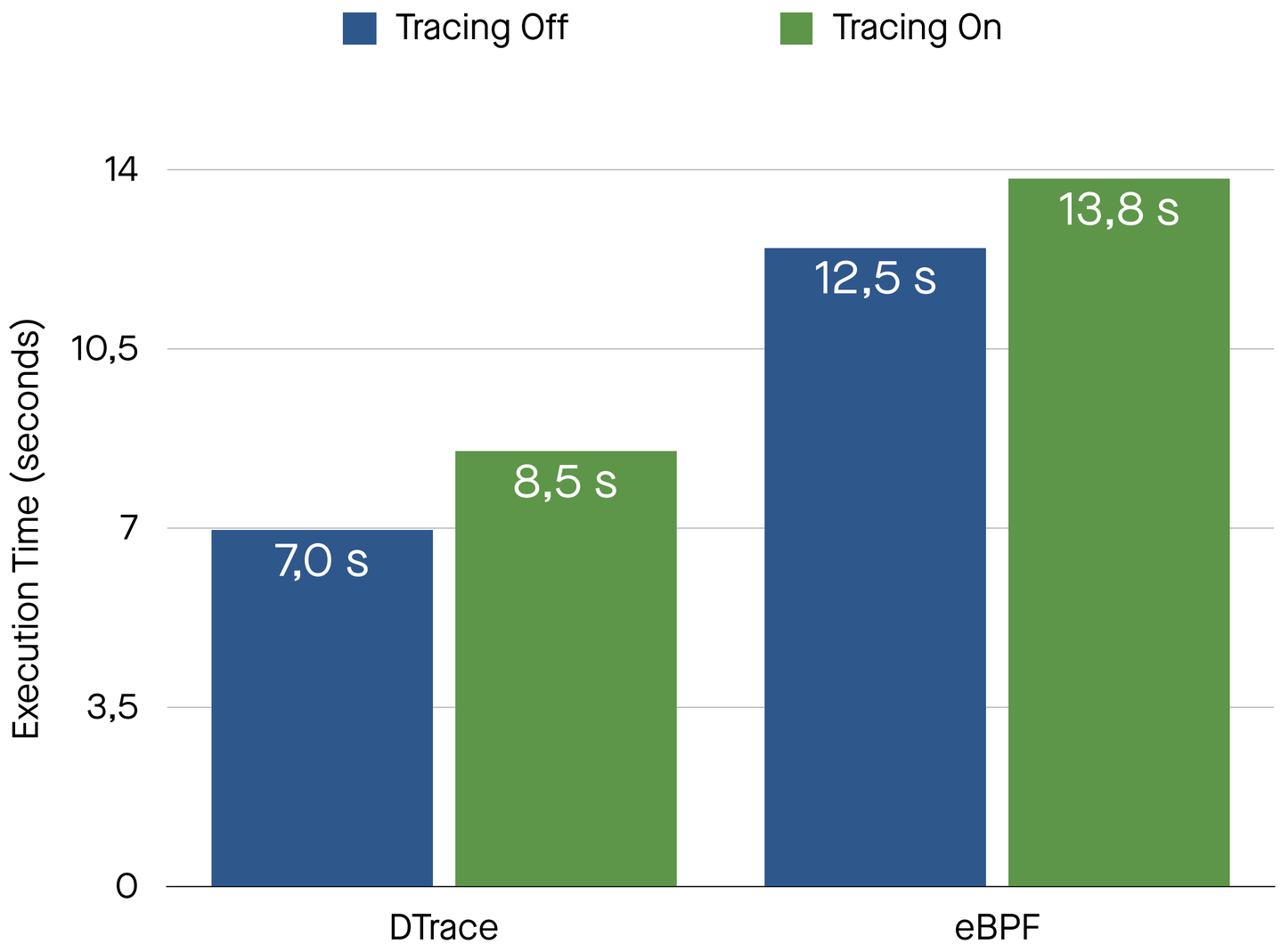
Experiment 1: Workload & Tracing Scripts

```
dd if=/dev/zero of=/dev/null bs=1 count=$((10 * 1000 * 1000))
```

```
dtrace -n 'syscall:freebsd:read:entry { }'
```

```
bpftrace -e 'tracepoint:syscalls:sys_enter_read { }'
```

Experiment 1: Results



2

Experiment 2: Setup

Hardware:

- VMs (32 CPUs, 400 GB RAM)
- FreeBSD 13.1, Ubuntu 18.04

Workload:

- 2.3 million 4k blocks transferred with dd(1) from /dev/urandom to an SSD

Tracing:

- Histogram of the return value of read() syscalls

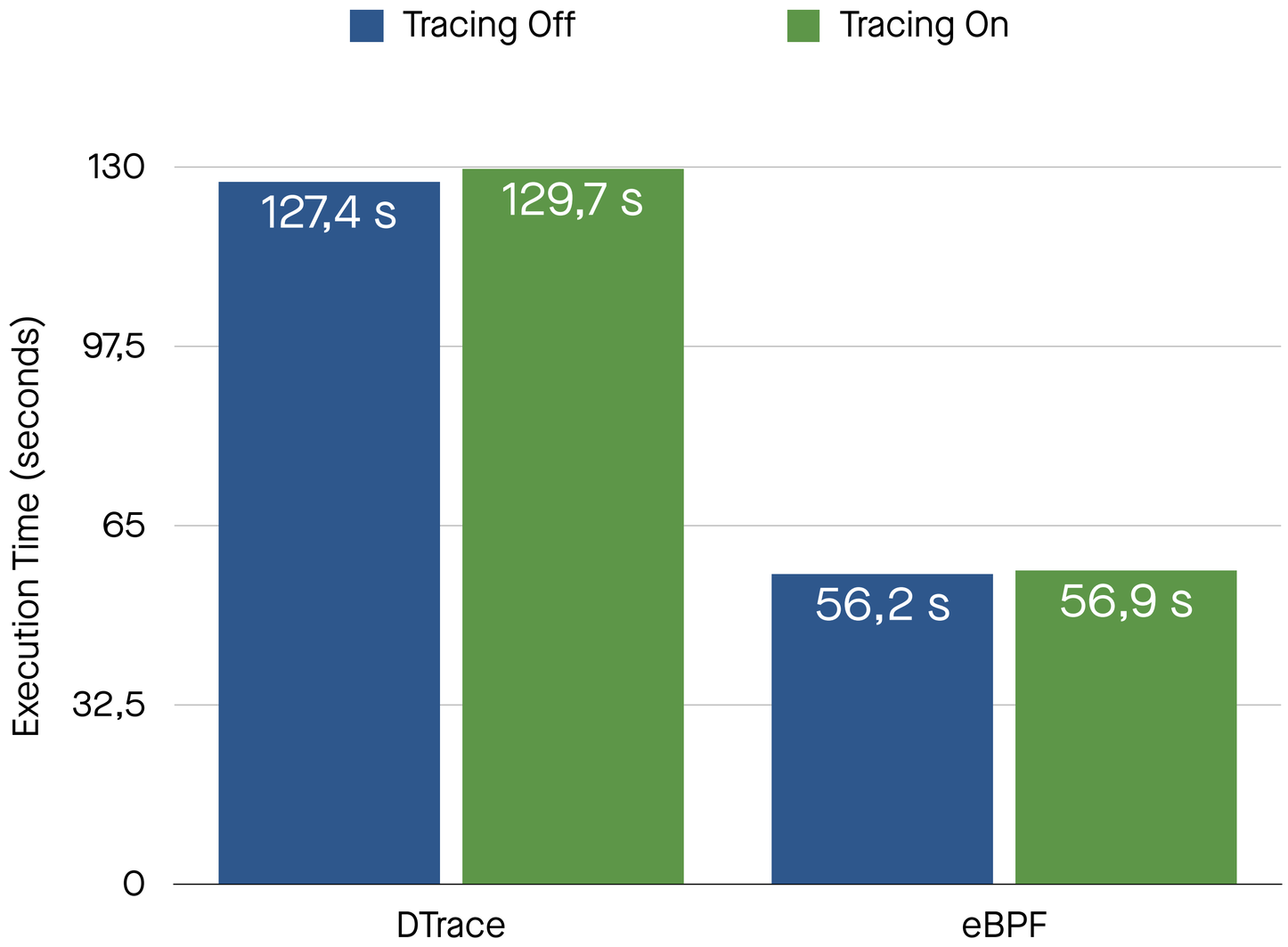
Experiment 2: Workload & Tracing Scripts

```
dd if=/dev/urandom of=... bs=4k count=2300000 conv=fsync
```

```
dtrace -n 'syscall::read:return /pid == $target/ { @ = quantize(arg0); }' \
        -c "$(command -v dd) if=/dev/urandom of=/dev/ada1p2 bs=4k count=2300000 conv=fsync"
```

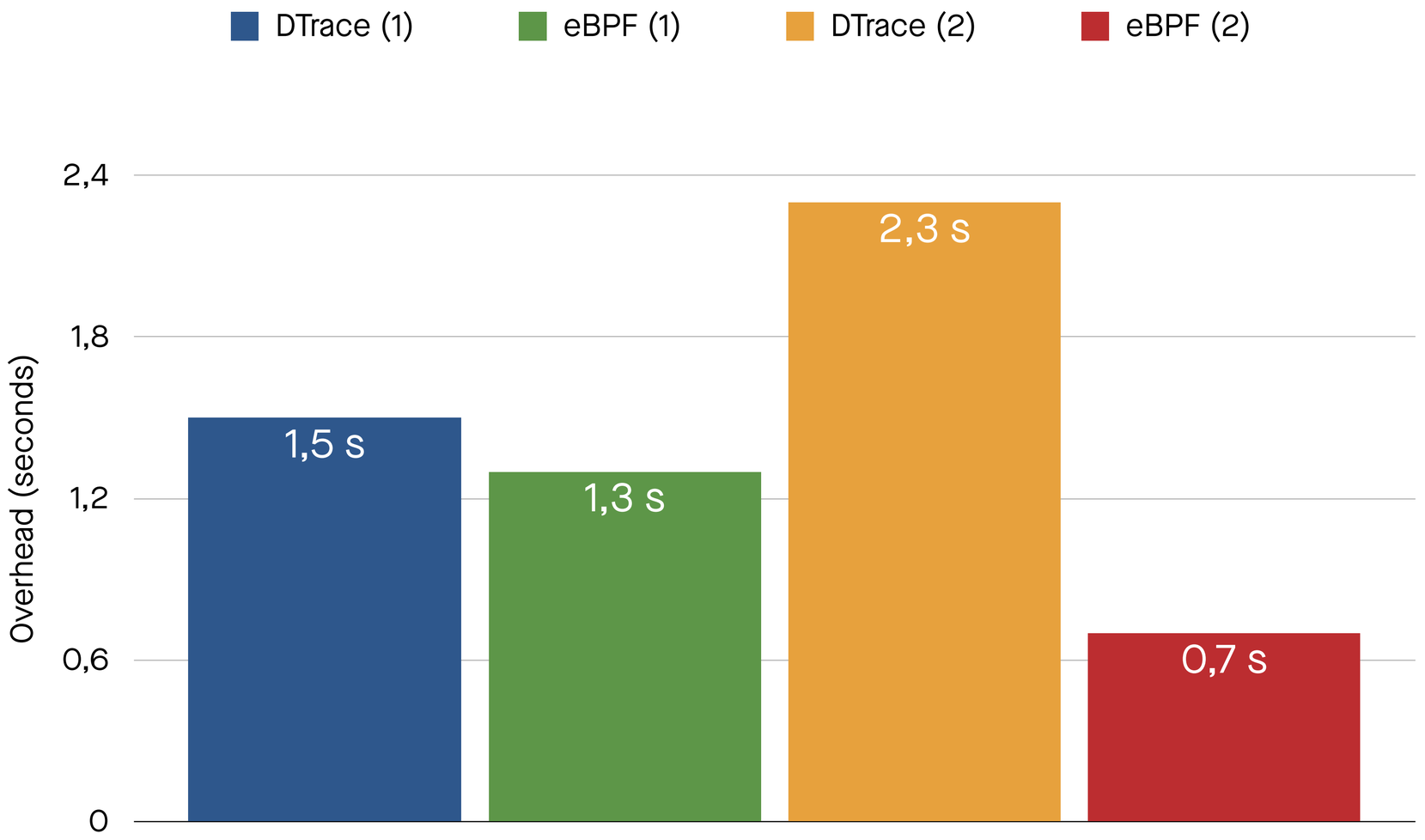
```
bpftrace -e 'tracepoint:syscalls:sys_exit_read /pid == cpid/ { @bytes = hist(args→ret); }' \
        -c "$(command -v dd) if=/dev/urandom of=/dev/sdb2 bs=4k count=2300000 conv=fsync"
```

Experiment 2: Results

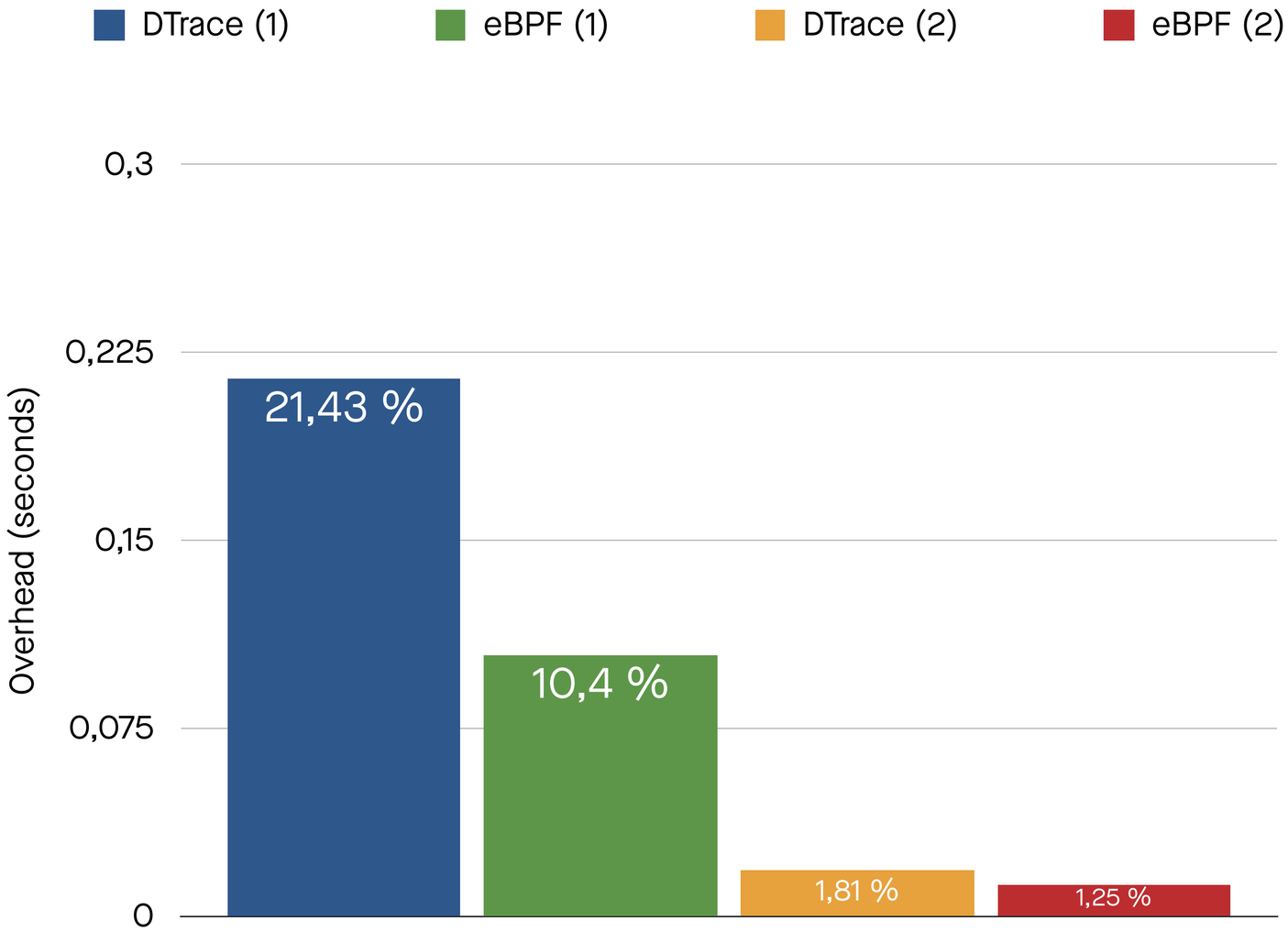


Analysis

Analysis: Overhead in Seconds



Analysis: Overhead in Percents



How does it compare to the results of others?

Results of benchmarking the eBPF overhead when tracing getpid():

Case	ns/op	overhead ns/op	ops/s	overhead percent
no probe	316	0	3,164,556	0 %
simple	424	108	2,358,490	34 %
complex	647	331	1,545,595	105 %

Source: https://github.com/cloudflare/ebpf_exporter/tree/master/benchmark

Conclusions & Future Work

Conclusions & Future Work

- The overhead exists!
- DTrace on FreeBSD and eBPF on Linux are difficult to compare
- Taming your benchmarking environment is hard
- Passive benchmarking is surely not enough
  → Need to explore other methodologies (e.g., active benchmarking)

VIENNA 17.09.2022

thank you

MATEUSZ PIOTROWSKI · 0MP@FREEBSD.ORG · TWITTER @0MPTS