OTTAWA 19.05.2023

PERFORMANCE ANALYSIS
of DTrace on FreeBSD & eBPF on Linux

MATEUSZ PIOTROWSKI

What overhead does tracing impose on a system?

OUTLINE

A FEW WORDS ABOUT ME

- FreeBSD user since 2016
- FreeBSD committer since 2018
- FreeBSD core team member since 2022
- Working with folks @ Klara Inc.

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?

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?

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

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

tracers 🦄

DTRACE CRASH COURSE

```
# dtrace -n '
    syscall::read:return                                     <-- Probe
    /execname == "sshd"/                                     <-- Predicate
    {                                                        <-- Action body (clause)
        @ = quantize(arg0);                                  <-- Aggregation (action)
    }
'
dtrace: description 'syscall::read:return ' matched 2 probes   <-- DTrace is tracing...
^C                                                             <-- Ctrl-C to interrupt tracing

           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@                     2   <-- Tracing results
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@                     2
              64 |                                         0
```

BPFTRACE CRASH COURSE

```
# bpftrace -e '
    tracepoint:syscalls:sys_exit_read                              <-- Probe
    /comm == "sshd"/                                               <-- Predicate
    {                                                              <-- Action
        @ = hist(args->ret);                                       <-- Map function
    }
'
Attaching 1 probe...                                               <-- bpftrace is tracing...
^C                                                                 <-- Ctrl-C to interrupt tracing

@:                                                                 <-- Tracing results
[2, 4)                 1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4, 8)                 0 |                                                    |
[8, 16)                0 |                                                    |
[16, 32)               0 |                                                    |
[32, 64)               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

Static probes:

- Created during compilation
- Stable interface
- May slightly impact performance even when not attached to

Dynamic probes:

- Created ad-hoc
- Unstable interface
- Unattached probes do not impose performance penalties

	FreeBSD	Linux
Userspace dynamic probes	pid provider[1]	uprobe
Userspace static probes	USDT	USDT
Kernel dynamic probes	fbt provider[2]	kprobe
Kernel static probes	SDT	tracepoint

1: The pid provider and uprobes are very different.

2: Soon also instructions within functions via the kinst provider.

benchmarks

Benchmark 1

- Workload: Read from /dev/zero and write to /dev/null
- Target: Overhead of tracer's basic features
- Based on a benchmark from Brendan Gregg's *BPF Performance Tools*

Benchmark 2

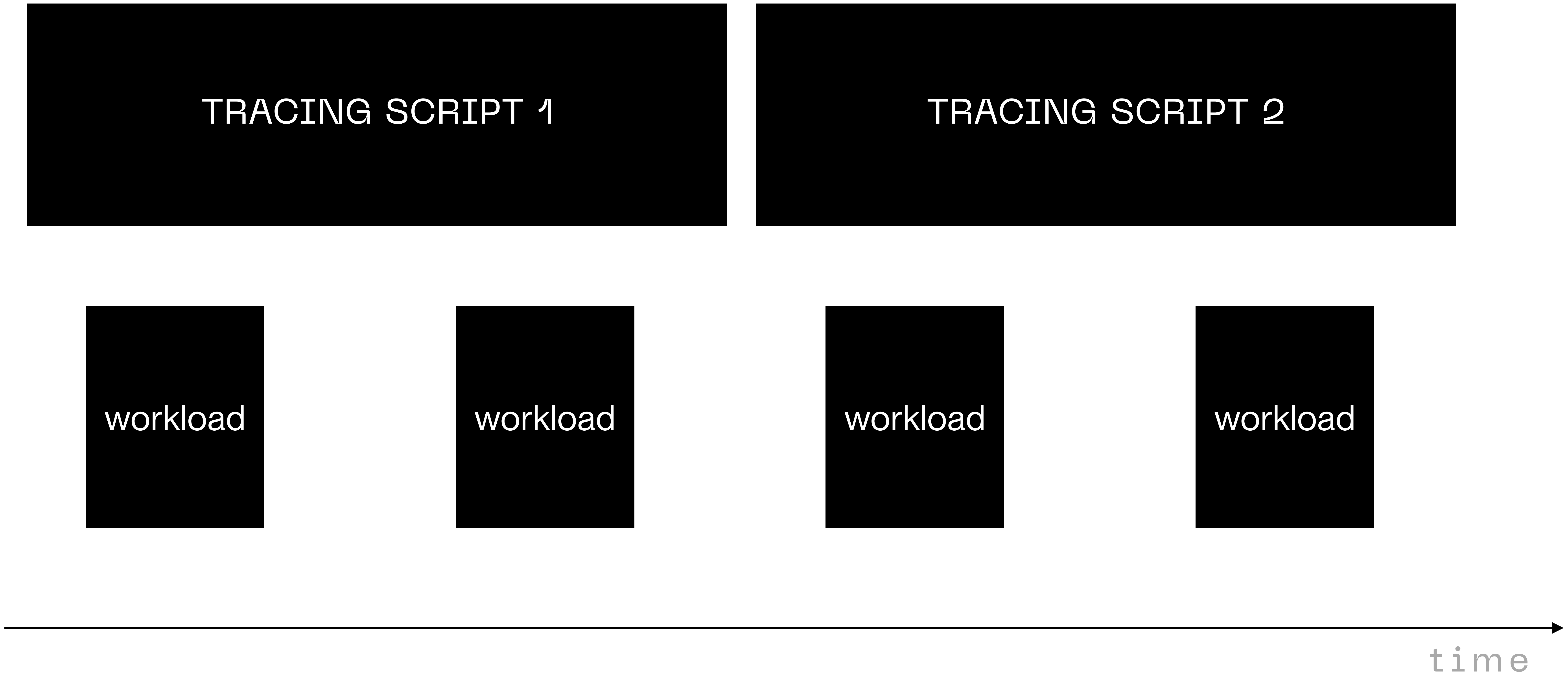
- Workload: FreeBSD's make buildkernel
- Target: Overhead of tracing complex workloads
- Based on the CADETS technical report

BENCHMARKS: HARNESS

Benchmark harness: Hyperfine (https://github.com/sharkdp/hyperfine)

- Warmup runs
- Setup & cleanup scripts
- Outliers detection
- 11/10

BENCHMARKS: BENCHMARK RUNS



system setup

SYSTEM SETUP

Hardware

- amd64
- 32 CPUs (Intel Xeon Gold 6226R CPU @ 2.90GHz)
- Almost 400 GB RAM

Operating systems

- FreeBSD 13.1-RELEASE-p1
- Ubuntu 20.04.5 (bpftrace 0.17.0)

Disabled hyperthreading and dynamic frequency scaling

1

dd if=/dev/zero of=/dev/null bs=1 count=10000000

BENCHMARK 1: BACKGROUND

- Measurment of per-event cost of different tracer features
- Principle of least perturbation (i.e., pick the fastest run)
- 18 different tracing scripts
- Setup and results described in Brendan Gregg's *BPF Performance Tools*
  - Workload assigned to a single CPU via cpuset(1) and taskset(1)
  - Linux 4.15, Intel Core i7-8650U

scripts

BENCHMARK 1: SCRIPT 01: CONTROL

```
# 01.bt
BEGIN {}
```

```
# 01.d
dtrace:::BEGIN {}
```

BENCHMARK 1: SCRIPTS 02 & 03: KPROBE & KRETPROBE

```
# 02.bt
k:vfs_read {
    1
}

# 03.bt
kr:vfs_read {
    1
}
```

```
# 02.d
fbt::dofileread:entry {
    1
}

# 03.d
fbt::dofileread:return {
    1
}
```

- VFS is usually traced with the vfs provider on FreeBSD. Use fbt instead to use dynamic instrumentation.
- *fbt* cannot reach *vfs_read()* equivalent on FreeBSD. Instrument *dofileread()* instead.

BENCHMARK 1: SCRIPTS 04 & 05: TRACEPOINT ENTRY & TRACEPOINT RETURN

```
# 04.bt
t:syscalls:sys_enter_read {
    1
}

# 05.bt
t:syscalls:sys_exit_read {
    1
}
```

```
# 04.d
syscall:freebsd:read:entry {
    1
}

# 05.d
syscall:freebsd:read:return {
    1
}
```

- Tracing of the kernel with static probes.

BENCHMARK 1: SCRIPTS 06 & 07: UPROBE & URETPROBES

```
# 06.bt
u:libc:__read {
    i
}
```

Uprobes support file-based tracing.

FreeBSD does not have an equivalent yet.

The tracing of functions, which have not started yet, is hard.
Let's try anyway.

The DTrace command is:

```
dtrace -C -q -D DTRACE_SCRIPT="\"06.d\"" -s "06.d"
```

BENCHMARK 1: SCRIPTS 06 & 07: UPROBE & URETPROBES: 06.D (1/3)

```
# 06.d (1/3)
#pragma D option destructive

#define TARGET_PROCESS_ARGS          "dd if=/dev/zero of=/dev/null bs=1 count=10000000"
#define LIBC_PATH_PREFIX              "/lib/libc.so"
#define LIBC_PATH_PREFIX_LEN         (sizeof(LIBC_PATH_PREFIX) - 1)
```

BENCHMARK 1: SCRIPTS 06 & 07: UPROBE & URETPROBES: 06.D (2/3)

```
# 06.d (2/3)
#ifndef READY_TO_ATTACH /* This is the DTrace parent script. */
syscall::open:entry /curpsinfo->pr_psargs == TARGET_PROCESS_ARGS && arg0 != NULL && \
                     substr(copyinstr(arg0), 0, LIBC_PATH_PREFIX_LEN) == LIBC_PATH_PREFIX/ {
            self->path = copyinstr(arg0); /* Save the path. */
}

syscall::open:return /self->path != ""/ {
            self->fd[arg1] = 1; /* Do not forget the file descriptor. */
}

syscall::close:entry /* On successful close of libc, spawn the DTrace child script. */
/self->fd[arg0] > 0 && self->path != ""/ {
            stop();
            system("dtrace -C -D READY_TO_ATTACH -p %d -s %s", pid, DTRACE_SCRIPT);

            /* Clean up variables to prepare for the next workload run. */
            self->path = 0;
            self->fd[arg0] = 0;
}
#endif
```

BENCHMARK 1: SCRIPTS 06 & 07: UPROBE & URETPROBES: 06.D (3/3)

```
# 06.d (3/3)
#ifdef READY_TO_ATTACH
pid$target:libc*:_read:entry
{
            1;
}

proc:::exit
/pid == $target/
{
            exit(0);
}
#endif
```

BENCHMARK 1: SCRIPTS 08 & 09: FILTER & MAP

```
# 08.bt
k:vfs_read /arg2 > 0/ {
    1
}

# 09.bt
k:vfs_read {
    @ = count()
}
```

```
# 08.d
fbt::dofileread:entry /args[3]->uio_resid > 0/ {
    1
}

# 09.d
fbt::dofileread:entry {
    @ = count()
}
```

BENCHMARK 1: SCRIPTS 10, 11, & 12: SINGLE KEY, STRING KEY, & TWO KEYS

```
# 10.bt
k:vfs_read {
    @[pid] = count()
}
```

```
# 11.bt
k:vfs_read {
    @[comm] = count()
}
```

```
# 12.bt
k:vfs_read {
    @[pid, comm] = count()
}
```

```
# 10.d
fbt::dofileread:entry {
    @[pid] = count()
}
```

```
# 11.d
fbt::dofileread:entry {
    @[execname] = count()
}
```

```
# 12.d
fbt::dofileread:entry {
    @[pid, execname] = count()
}
```

BENCHMARK 1: SCRIPTS 13 & 14: USER STACK & KERNEL STACK

```
# 13.bt
k:vfs_read {
    @[kstack] = count()
}
```

```
# 14.bt
k:vfs_read {
    @[ustack] = count()
}
```

```
# 13.d
fbt::dofileread:entry {
    @[stack()] = count()
}
```

```
# 14.d
fbt::dofileread:entry {
    @[ustack()] = count()
}
```

BENCHMARK 1: SCRIPT 15: HISTOGRAM

```
# 15.bt
k:vfs_read {
    @ = hist(arg2)
}
```

```
# 15.d
fbt::dofileread:entry {
    @ = quantize(args[3]->uio_resid)
}
```

BENCHMARK 1: SCRIPT 16: TIMING

```
# 16.bt
k:vfs_read {
    @s[tid] = nsecs
}

kr:vfs_read /@s[tid]/ {
    @ = hist(nsecs - @s[tid]);
    delete(@s[tid]);
}
```

```
# 16.d
fbt::dofileread:entry {
    self->s = timestamp
}

fbt::dofileread:return /self->s/ {
    @ = quantize(timestamp - self->s);
    self->s = 0;
}
```

BENCHMARK 1: SCRIPT 17: MULTIPLE

```
# 17.bt
k:vfs_read {
    @[kstack, ustack] = hist(arg2)
}
```

```
# 17.d
fbt::dofileread:entry {
    @[stack(), ustack()] = quantize(args[3]->uio_resid)
}
```

BENCHMARK 1: SCRIPT 18: PER EVENT

```
# 18.bt
k:vfs_read {
    printf("%d bytes\n", arg2)
}
```

```
# 18.d
fbt::dofileread:entry {
    printf("%d bytes\n", args[3]->uio_resid);
}
```

results

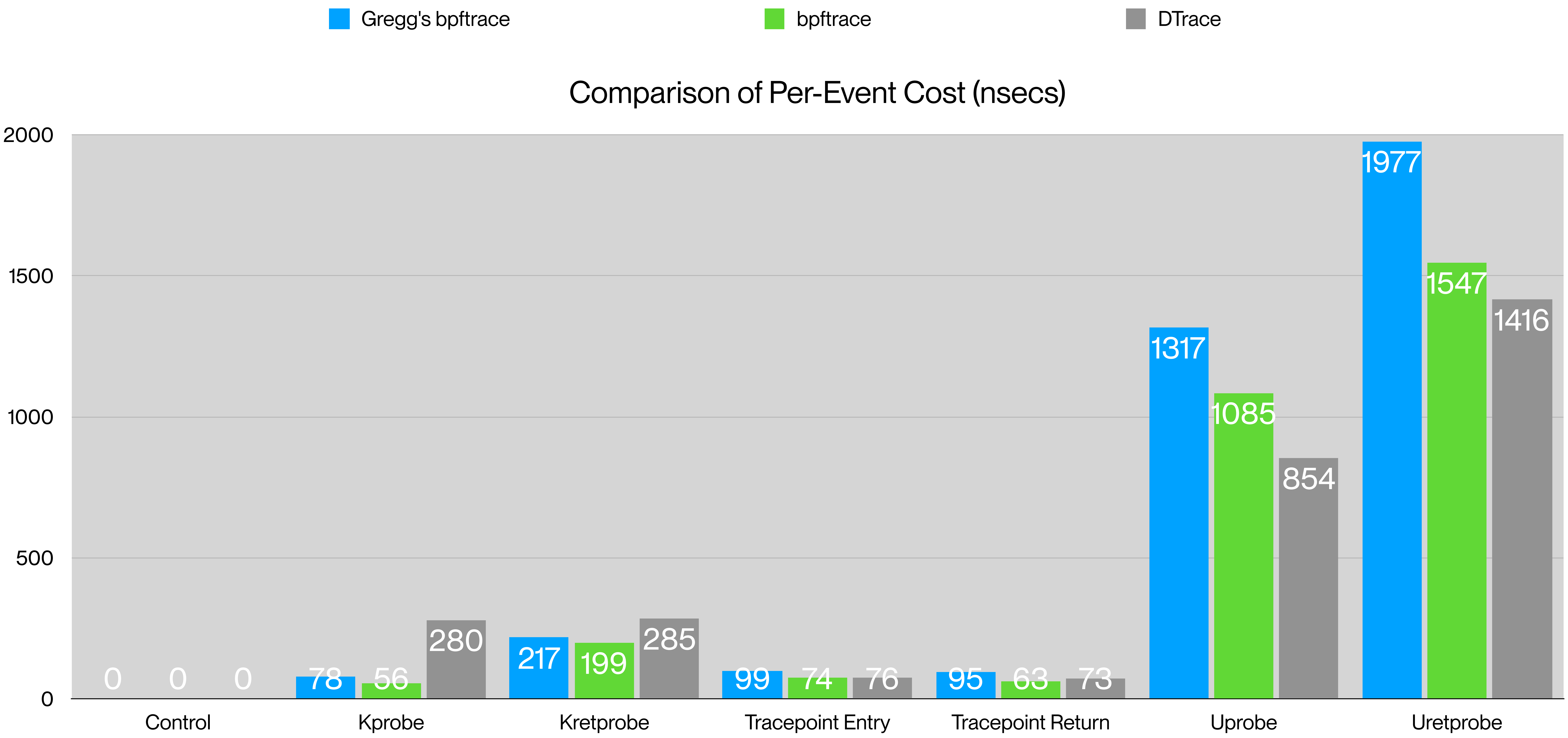
BENCHMARK 1: PER-EVENT COST (NSECS)

		Gregg's bpftrace	bpftrace	DTrace
1	Control	0	0	0
2	Kprobe	78	56	280
3	Kretprobe	217	199	285
4	Tracepoint Entry	99	74	76
5	Tracepoint Return	95	63	73
6	Uprobe	1317	1085	854
7	Uretprobe	1977	1547	1416
8	Filter	128	57	315
9	Map	194	77	327
10	Single Key	212	130	356
11	String Key	231	160	377
12	Two Keys	234	176	401
13	Kernel Stack	344	322	762
14	User Stack	668	1077	827
15	Histogram	238	133	360
16	Timing	651	473	682
17	Multiple	856	1264	1313
18	Per Event	870	1539	312

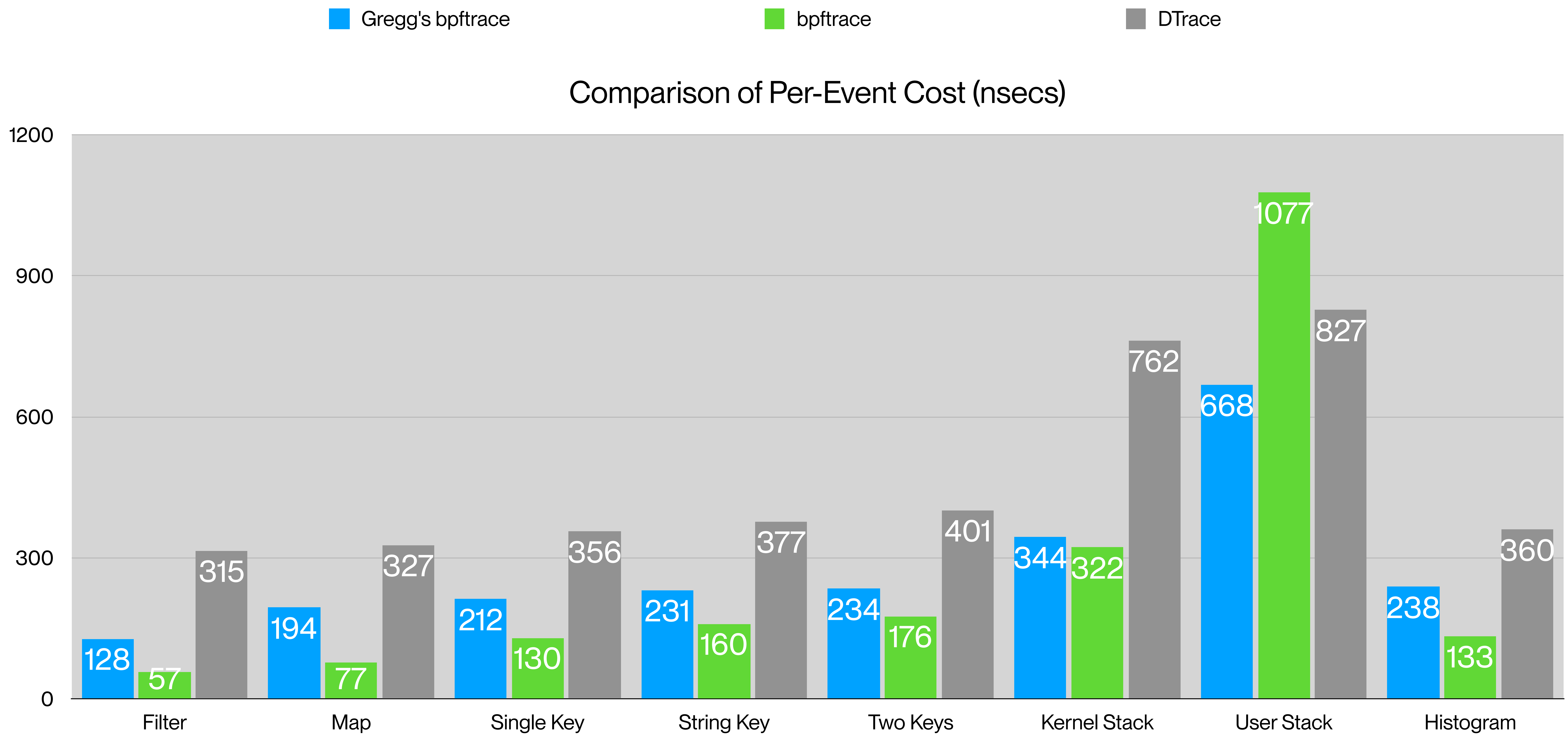
BENCHMARK 1: RELATIVE SLOWDOWN

		Gregg's bpftrace	bpftrace	DTrace
1	Control	0 %	0 %	0 %
2	Kprobe	13 %	21 %	121 %
3	Kretprobe	36 %	76 %	123 %
4	Tracepoint Entry	17 %	28 %	33 %
5	Tracepoint Return	16 %	24 %	32 %
6	Uprobe	221 %	414 %	368 %
7	Uretprobe	331 %	591 %	611 %
8	Filter	21 %	22 %	136 %
9	Map	33 %	29 %	141 %
10	Single Key	36 %	49 %	154 %
11	String Key	39 %	61 %	163 %
12	Two Keys	39 %	67 %	173 %
13	Kernel Stack	58 %	123 %	329 %
14	User Stack	112 %	411 %	357 %
15	Histogram	40 %	51 %	155 %
16	Timing	109 %	181 %	294 %
17	Multiple	143 %	483 %	566 %
18	Per Event	146 %	588 %	135 %

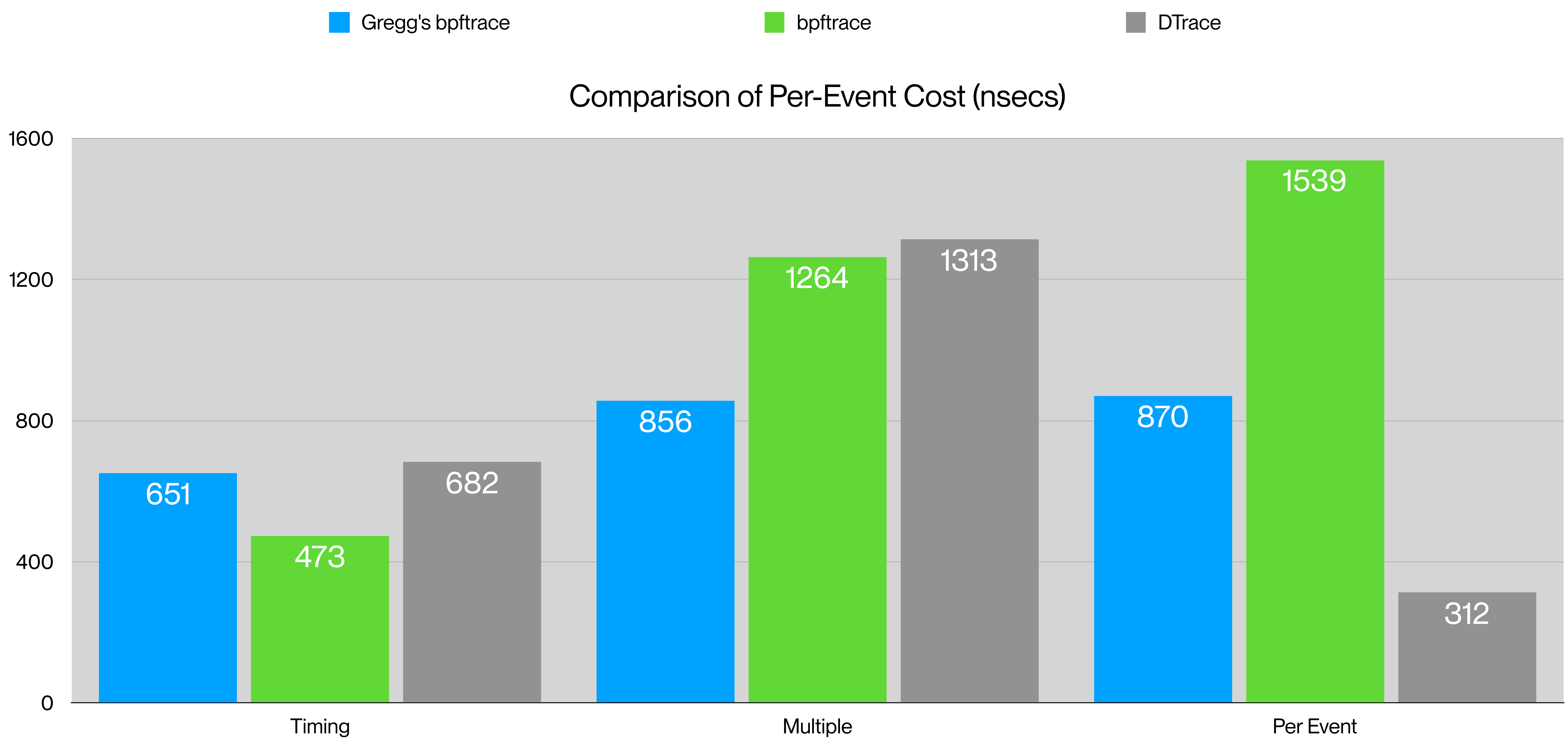
BENCHMARK 1: COMPARISON OF PER-EVENT COST: EXPERIMENTS 01–07



BENCHMARK 1: COMPARISON OF PER-EVENT COST: EXPERIMENTS 08–15



BENCHMARK 1: COMPARISON OF PER-EVENT COST: EXPERIMENTS 16–18



BENCHMARK 1: SUMMARY

- When tracing frequent events like system calls, the overhead can be as high as 600%.
- Implementation of probes has a huge imact on performance
    - Return probes are not as expensive on FreeBSD as they are on Linux.
- bpftrace seems to have a better performance overall than DTrace.
- Per-event cost (last experiment) is surprisingly low on FreeBSD...

🤔

2

make -j 32 buildkernel

BENCHMARK 2: BACKGROUND

- Measurment of tracing impact on complex workloads
- Setup and results described in the CADETS technical report
  - Only DTrace (FreeBSD 11, 12, or 13)
  - 9 different tracing scenarios (tracing action: counting the number of probe activations)
- Kernel build on an in-memory disk formatted with UFS or XFS.
  - With kernel-toolchain prebuilt
  - Had to work around bpftrace limits:
    - Increase the limit of allowed open file descriptors to 200000 (that's a lot of /dev/null's).
    - Set BPFTRACE_MAX_BPF_PROGS and BPFTRACE_MAX_PROBES to 22000.

scripts

BENCHMARK 2: SCRIPTS

#	fbt	syscall	vfs	sched
0	—	—	—	—
1	UFS	all	all	—
2	UFS-occ	entry	wroc	—
3	UFS-occ	all	wroc	—
4	UFS	all	all	all
5	UFS-occ	entry	wroc	all
6	UFS-occ	all	wroc	all
7	UFS-a	all	—	—
8	UFS-abv	all	—	—
9	—	—	all	—

BENCHMARK 2: FBT PROVIDER

```
# bpftrace

# UFS (3684)
kprobe:xfs_*, kretprobe:xfs_*

# UFS-occ (8)
kprobe:xfs_dir_open, kretprobe:xfs_dir_open,
kprobe:xfs_file_open, kretprobe:xfs_file_open,
kprobe:fput, kretprobe:fput,
kprobe:xfs_create, kretprobe:xfs_create

# UFS-a (11086)
kprobe:xfs_*, kretprobe:xfs_*,
kprobe:a*, kretprobe:a*

# UFS-abv (18268)
kprobe:xfs_*, kretprobe:xfs_*,
kprobe:a*, kretprobe:a*,
kprobe:b*, kretprobe:b*,
kprobe:v*, kretprobe:v*
```

```
# DTrace

# UFS (129)
fbt::ufs_*:

# UFS-occ (6)
fbt::ufs_open:,
fbt::ufs_close:,
fbt::ufs_create:

# UFS-a (3588)
fbt::ufs_*:,
fbt::a*:

# UFS-abv (8040)
fbt::ufs_*:,
fbt::a*:,
fbt::b*:,
fbt::v*:
```

BENCHMARK 2: SYSCALL PROVIDER

```
# bpftrace

# all (574)
tracepoint:syscalls:*

# entry (287)
tracepoint:syscalls:sys_enter_*
```

```
# DTrace

# all (2296)
syscall:::

# entry (1148)
syscall:::entry
```

BENCHMARK 2: VFS PROVIDER

```
# bpftrace

# all (134)
kprobe:vfs_*, kretprobe:vfs_*

# wroc (8)
kprobe:vfs_write, kretprobe:vfs_write,
kprobe:vfs_read, kretprobe:vfs_read,
kprobe:vfs_open, kretprobe:vfs_open,
kprobe:__close_fd, kretprobe:__close_fd
```

```
# DTrace

# all (181)
vfs:::

# wroc (8)
vfs::vop_write:,
vfs::vop_read:,
vfs::vop_open:,
vfs::vop_close:
```

BENCHMARK 2: SCHED PROVIDER

```
# bpftrace

# all (24)
tracepoint:sched:*
```

```
# DTrace

# all (13)
sched:::
```

results

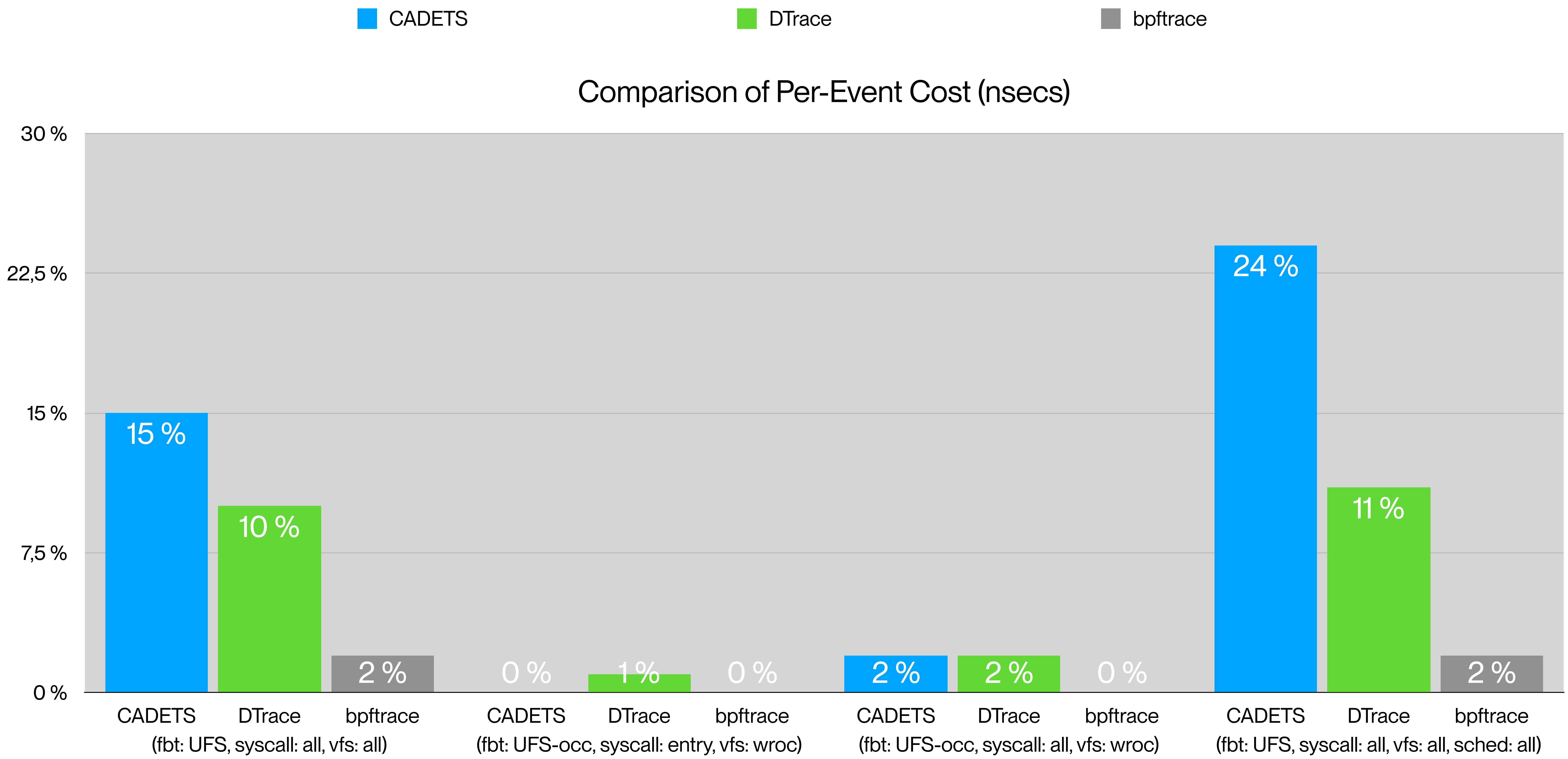
BENCHMARK 2: AVERAGE KERNEL BUILD TIMES (SECONDS)

	fbt	syscall	vfs	sched	CADETS	DTrace	bpftrace
0	—	—	—	—	460	32.92	43.28
1	**UFS**	**all**	**all**	—	530	36.43	44.05
2	**UFS-occ**	**entry**	**wroc**	—	460	33.38	43.32
3	**UFS-occ**	**all**	**wroc**	—	470	33.58	43.46
4	**UFS**	**all**	**all**	**all**	570	36.62	44.51
5	**UFS-occ**	**entry**	**wroc**	**all**	480	33.54	43.41
6	**UFS-occ**	**all**	**wroc**	**all**	500	33.69	43.52
7	**UFS-a**	**all**	—	—	570	35.61	49.97
8	**UFS-abv**	**all**	—	—	1210	160.36	62.14
9	—	—	**all**	—	550	35.09	43.15

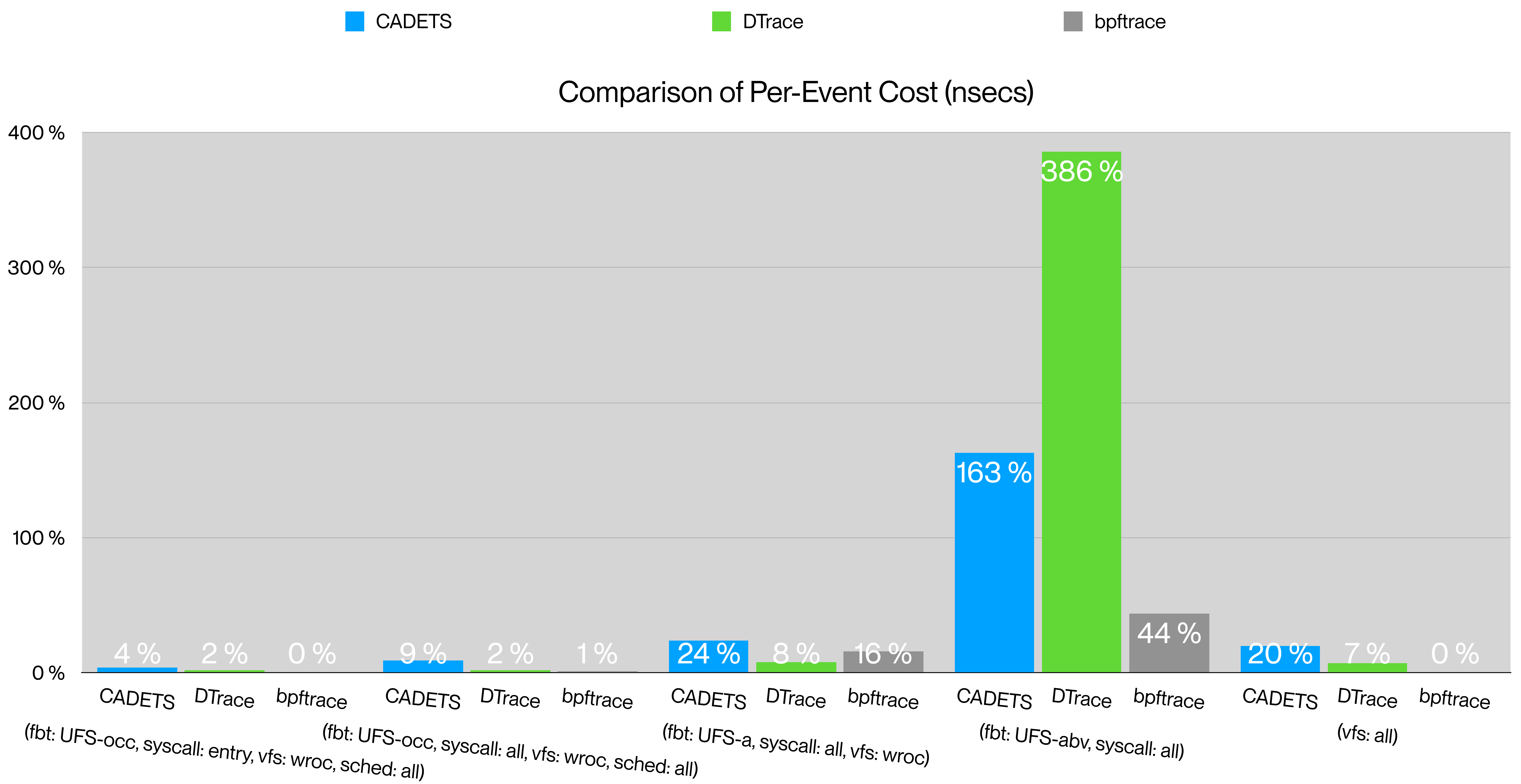
BENCHMARK 2: RELATIVE SLOWDOWN (OF BEST RUNS)

	fbt	syscall	vfs	sched	CADETS	DTrace	bpftrace
0	—	—	—	—	0 %	0 %	0 %
1	**UFS**	**all**	**all**	—	15 %	10 %	2 %
2	**UFS-occ**	**entry**	**wroc**	—	0 %	1 %	0 %
3	**UFS-occ**	**all**	**wroc**	—	2 %	2 %	0 %
4	**UFS**	**all**	**all**	**all**	24 %	11 %	2 %
5	**UFS-occ**	**entry**	**wroc**	**all**	4 %	2 %	0 %
6	**UFS-occ**	**all**	**wroc**	**all**	9 %	2 %	1 %
7	**UFS-a**	**all**	—	—	24 %	8 %	16 %
8	**UFS-abv**	**all**	—	—	163 %	386 %	44 %
9	—	—	**all**	—	20 %	7 %	0 %

BENCHMARK 2: COMPARISON OF PER-EVENT COST: EXPERIMENTS 1–4



BENCHMARK 2: COMPARISON OF PER-EVENT COST: EXPERIMENTS 5–9



BENCHMARK 2: SUMMARY

- When tracing complex workloads, the overhead of tracing is measurable (≥ 1%) and significant (≥ 5%) but not necessarily too expensive (still ≤ 30%).
- bpftrace seems to outperform DTrace but…
  - I observered that bpftrace needed ~10 minutes to stop when signalled at the end of experiment runs; DTrace stopped in way less than half a minute…

smoking gun

SMOKING GUN: KTRACE.D

```
# time dtrace -s ./ktrace.d -c 'cat /x' read
dtrace: script './ktrace.d' matched 51486 probes
CPU FUNCTION
  1  -> sys_read
  1    -> fget_read
  1      -> fget_unlocked
  1      <- fget_unlocked
  1    <- fget_read
...
  1          -> doselwakeup
  1          <- doselwakeup
  1          -> knote
  1          <- knote
  1        <- tty_wakeup
  1      <- ttydisc_getc_uio
  1    <- ptsdev_read
  1  <- dofileread
  1  <- sys_read
  1  <= read
...
real    0m1.069s
```

```
#pragma D option flowindent

syscall::$1:entry
{
        self->flag = 1;
}

fbt::: /self->flag/
{
}

syscall::$1:return
/self->flag/
{
        self->flag = 0;
        exit(0);
}
```

SMOKING GUN: KTRACE.BT

```
# export BPFTRACE_MAX_PROBES=5000
# export BPFTRACE_MAX_BPF_PROGS=2000
# ulimit -n 100000
# time bpftrace ./ktrace.bt -c '/bin/cat /x' read ext4_*
Attaching 1090 probes...
 =>tracepoint:syscalls:sys_enter_read
  ->kprobe:ext4_file_read_iter
  <-kretprobe:ext4_file_read_iter
 <=tracepoint:syscalls:sys_exit_read
 =>tracepoint:syscalls:sys_enter_read
  ->kprobe:ext4_file_read_iter
  <-kretprobe:ext4_file_read_iter
 <=tracepoint:syscalls:sys_exit_read
 =>tracepoint:syscalls:sys_enter_read
  ->kprobe:ext4_file_read_iter
  <-kretprobe:ext4_file_read_iter
 <=tracepoint:syscalls:sys_exit_read

real    0m44.312s
```

```
tracepoint:syscalls:sys_enter_$1 /pid == cpid/ {
            ...
}

kprobe:$2 /pid == cpid && @tracing[tid]/ {
            ...
}

kretprobe:$2 /pid == cpid && @tracing[tid]/ {
            ...
}

tracepoint:syscalls:sys_exit_$1
/pid == cpid && @tracing[tid]/ {
            ...
}
```

conclusion & future work

Overhead is significant but not necessarily expensive (≤30%).

A lot depends on the frequency of the traced events.

DTrace's performance is more predictable.

(See KUtrace for ≤1% overhead.)

Boldly go where no one has gone before.

special thanks

Devin Teske, George V. Neville-Neil, Mark Johnston, Domagoj Stolfa, Benedict Reuschling, Jan Nordholz, Ania Bui

OTTAWA 19.05.2023

thank you

MATEUSZ PIOTROWSKI