BENCHMARKING PERFORMANCE OVERHEAD

of DTrace on FreeBSD and eBPF on Linux

0mp@FreeBSD.org

INTRO

WHY?

The Internet

This tracer—so fast!!1 🦄
That tracer—so slow... 🐴

Talk is cheap. Show me the numbers.

The Internet

crickets 🦗🦗🦗

OUTLINE

Background

Observability
Tracers

Benchmarks

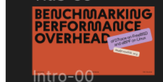
Benchmark Design
Benchmark 1
Benchmark 2
Smoking Gun

Conclusion

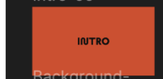
Future Work

OUTLINE

Title-00



Intro-00

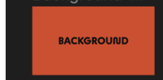


Intro-01

Intro-02

Intro-03

Background-...



Background-...

Background-...

Background-...

Background-...

Background-...

Background-...

Background-...

Background-...

Background-...

Background-...

Background-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-11

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Benchmarks-...

Conclusion-00

Conclusion-01

Conclusion-02

Conclusion-03

Conclusion-04

FutureWork-00

FutureWork-01

ThankYou-00

BACKGROUND

OBSERVABILITY

OBSERVABILITY: INTRO

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?

OBSERVABILITY: 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 [...]
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
```

OBSERVABILITY: MONITORING

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

ISN'T IT SLOW?

OBSERVABILITY: PROBE EFFECT...

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

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

TRACERS

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
```

TRACERS: 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 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

TRACERS: TYPES OF PROBES

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: Also the kinst provider.

BENCHMARKS

BENCHMARK DESIGN

BENCHMARK DESIGN: INTRO

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

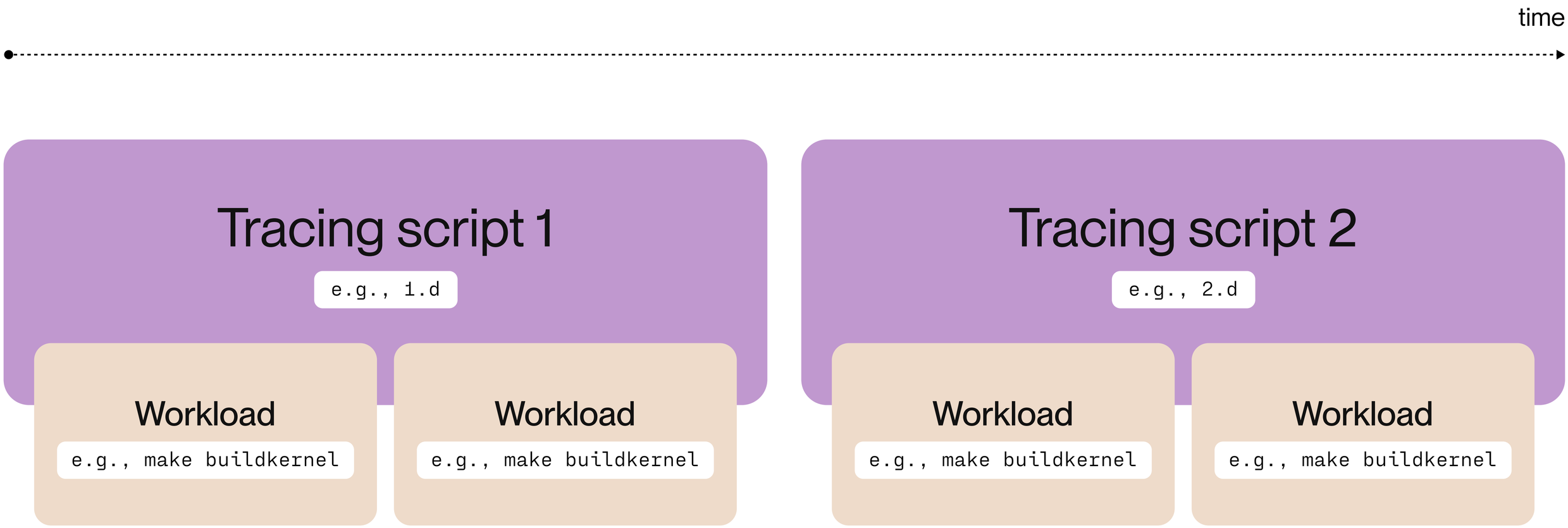
- Workload: FreeBSD's make buildkernel
- Target: Overhead of tracing complex workloads
- Based on the CADETS technical report

BENCHMARK DESIGN: HARNESS

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

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

BENCHMARK DESIGN: EXECUTION



BENCHMARK DESIGN: SYSTEM SETUP

Hardware:

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

Software:

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

Disabled hyperthreading and dynamic frequency scaling

BENCHMARK 1

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

BENCHMARK 1: BACKGROUND

- 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 & URETPROBE

```
# 06.bt
u:libc:__read {
    1
}
```

Uprobes support file-based tracing.

FreeBSD does not have an equivalent yet.

The tracing of programs 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 & URETPROBE: 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 & URETPROBE: 06.D (2/3)

```
# 06.d (2/3)
#ifndef READY_TO_ATTACH /* This is the 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; }

syscall::close:entry /self->fd[arg0] > 0 && self->path != ""/ {
    stop();
    system("dtrace -C -D READY_TO_ATTACH -p %d -s %s", pid, DTRACE_SCRIPT);
    self->path = 0;
    self->fd[arg0] = 0;
}
#endif
```

BENCHMARK 1: SCRIPTS 06 & 07: UPROBE & URETPROBE: 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()
}
```

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

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

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

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

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

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

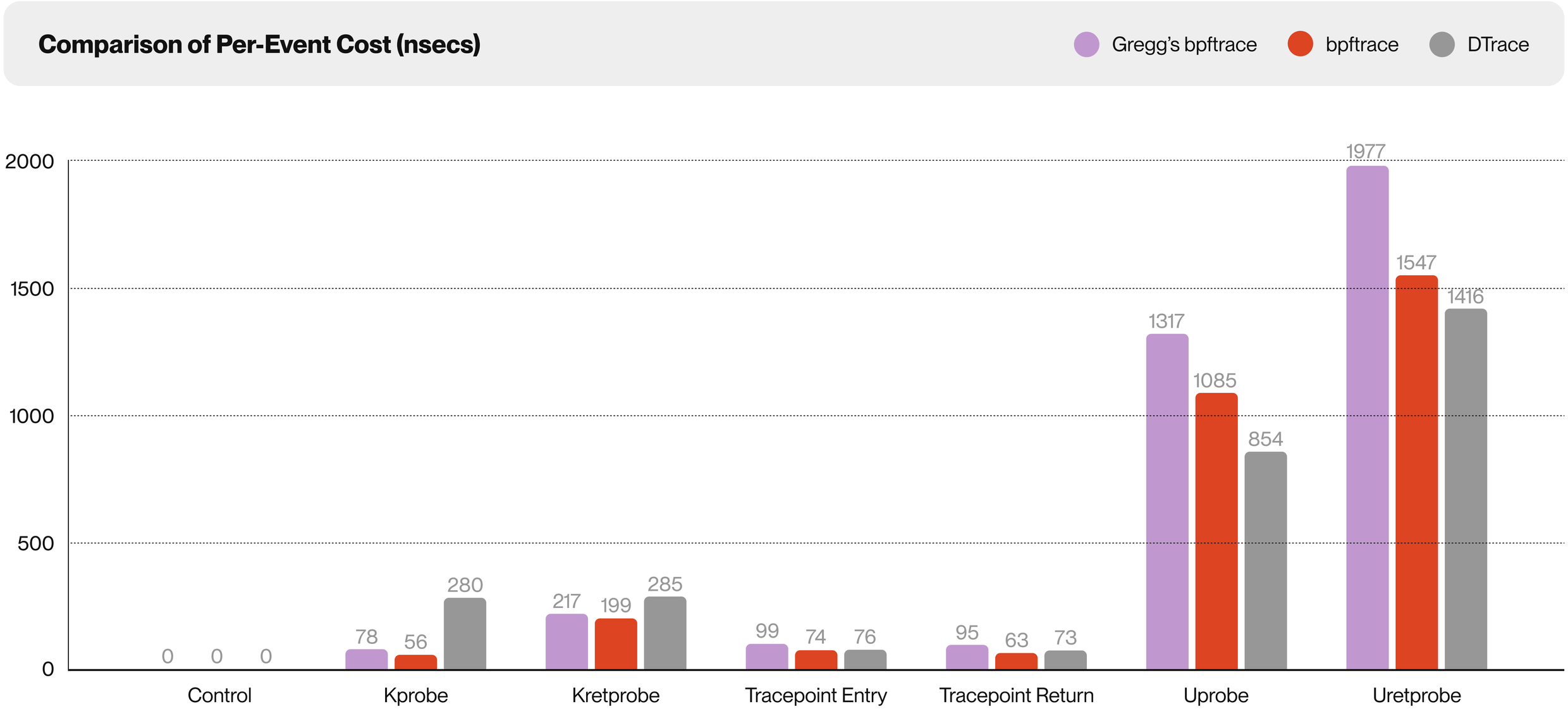
BENCHMARK 1: SCRIPT 18: PER EVENT

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

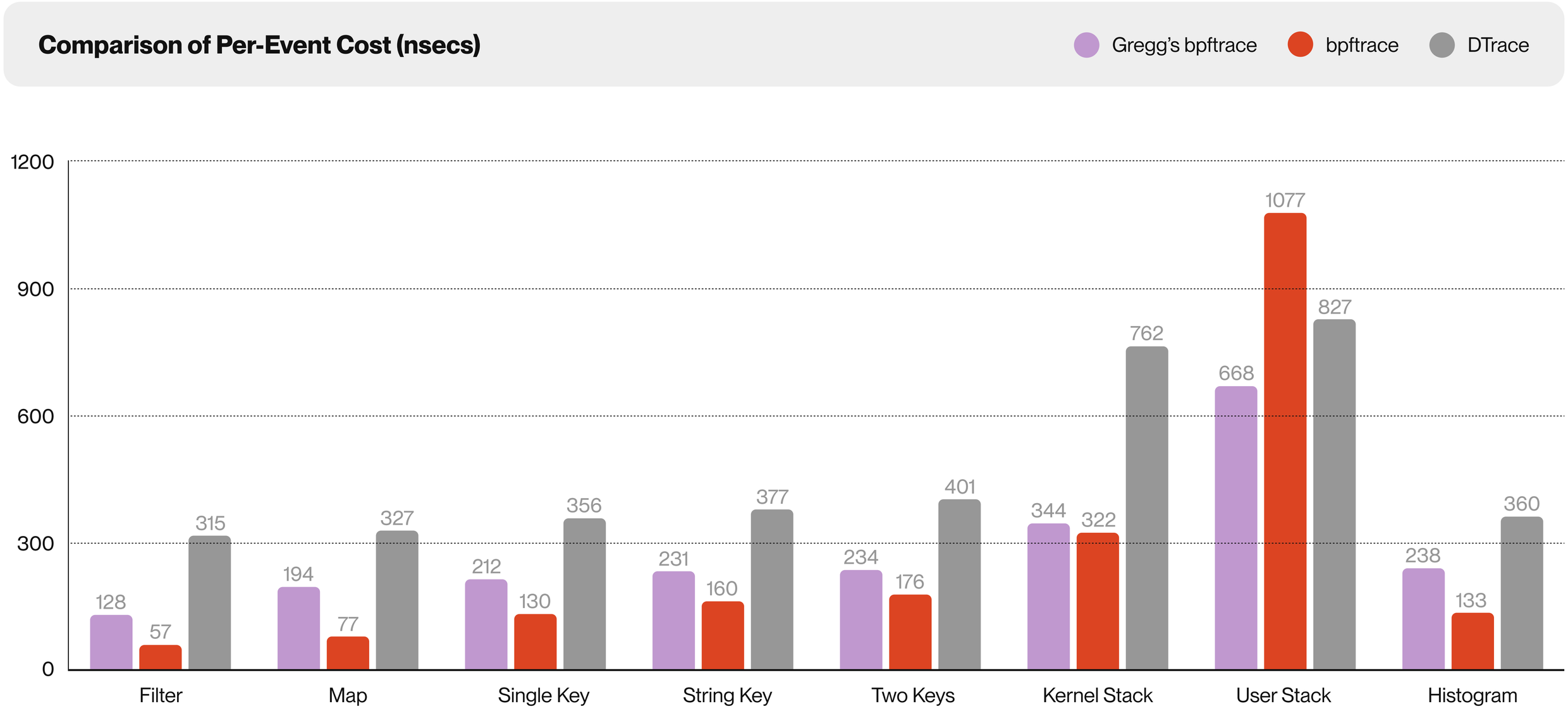
```
# 18.d
fbt::dofileread:entry {
    printf("%d bytes\n", args[3]->uio_resid);
}
```

RESULTS

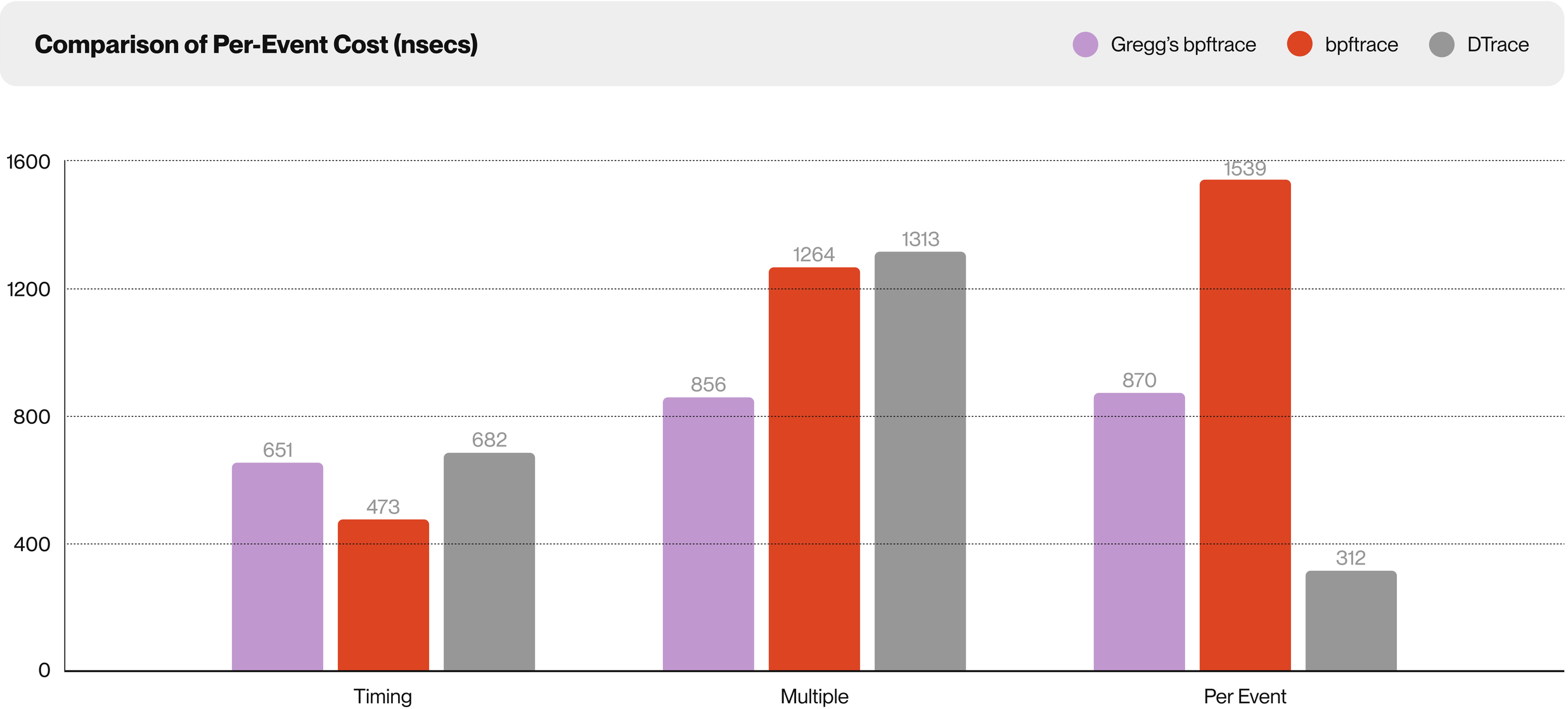
BENCHMARK 1: COMPARISON OF PER-EVENT COST: EXPERIMENTS 01-07



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



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



SUMMARY

BENCHMARK 1: SUMMARY

- When tracing frequent events like system calls, the overhead can be as high as 600%.
- Implementation of probes has a huge impact on performance
- Return probes are not as expensive on FreeBSD as they are on Linux.
- Per-event cost (last experiment) is surprisingly low on FreeBSD... 🤔

BENCHMARK 2

```
make -j 32 buildkernel
```

BENCHMARK 2: BACKGROUND

- 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:
```

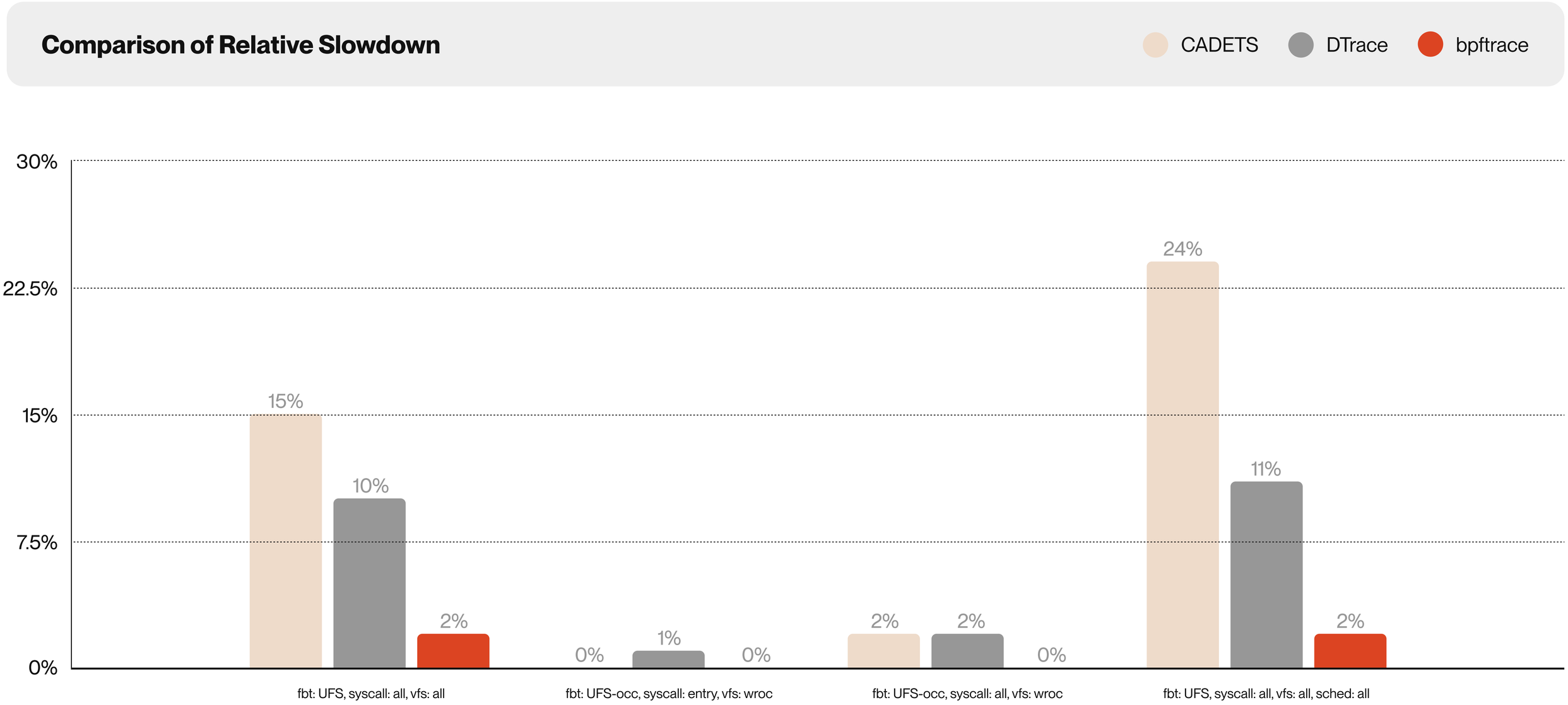
BENCHMARK 2: SCHED PROVIDER

```
# bpftrace

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

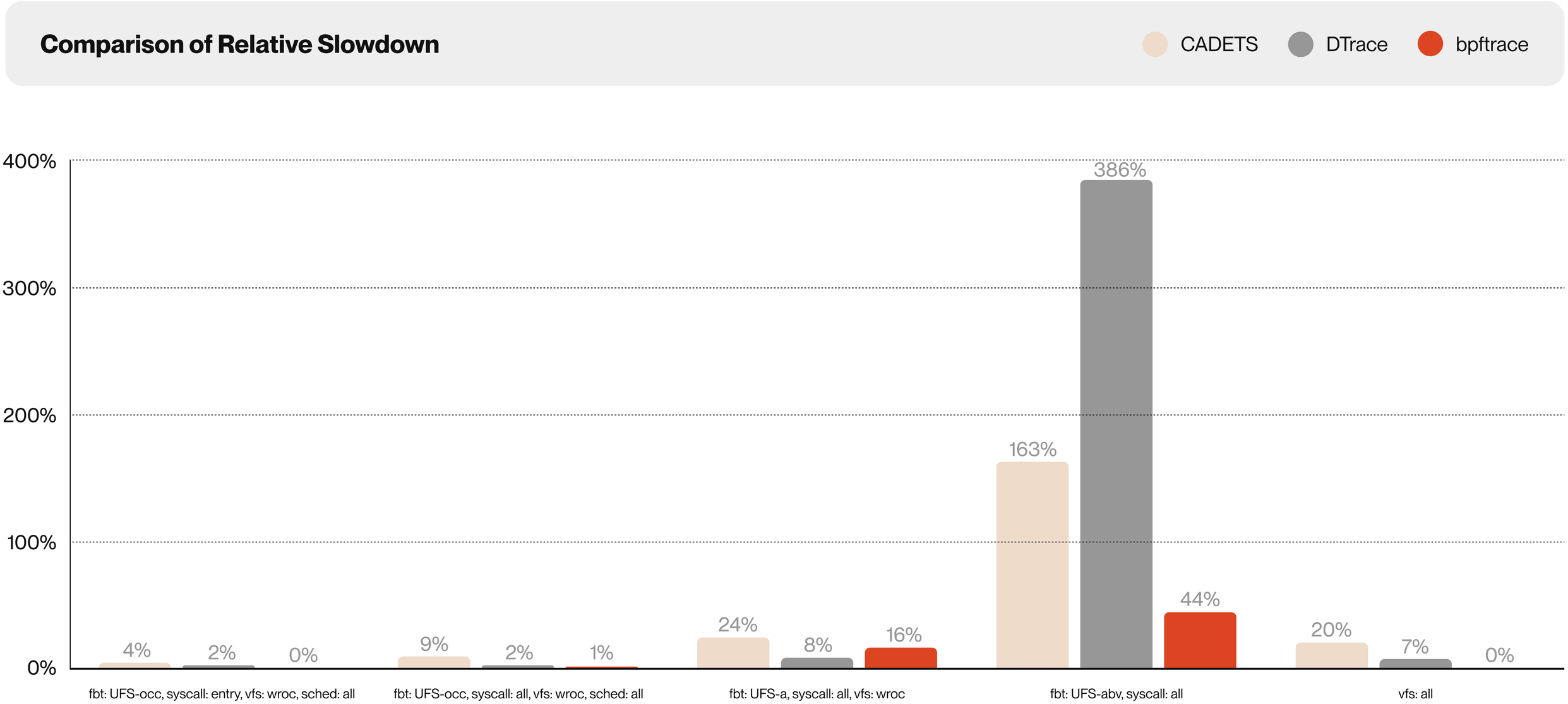
```
# DTrace

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

RESULTS

BENCHMARK 2: COMPARISON OF RELATIVE SLOWDOWN: EXPERIMENTS 1-4



BENCHMARK 2: COMPARISON OF RELATIVE SLOWDOWN: EXPERIMENTS 5-9



SUMMARY

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 observed that bpftrace needed ~10 minutes to stop when signaled at the end of experiment runs
  - DTrace stopped in 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

CONCLUSION

**OVERHEAD IS SIGNIFICANT,
BUT NOT NECESSARILY EXPENSIVE (≤30%).**

CONCLUSION

A LOT DEPENDS ON THE FREQUENCY OF THE TRACED EVENTS.

CONCLUSION

DTRACE'S PERFORMANCE IS MORE PREDICTABLE.

CONCLUSION

(CHECK OUT KUTRACE FOR ≤1% OVERHEAD.)

FUTURE WORK

FUTURE WORK

STABILITY OF TRACERS

PROBES IN BENCHMARK 2

KUTRACE

THANK YOU

Omp@FreeBSD.org