PERFORMANCE ANALYSIS of DTrace on FreeBSD & eBPF on Linux
What overhead does tracing impose on a system?
OUTLINE

About Me

Observability

Tracers

Benchmarks

Benchmark 1

Benchmark 2

Smoking Gun

Conclusion & Future Work
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
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?
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/156295597949902592
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 
    /execname == "sshd"/
    {
        @ = quantize(arg0);
    }
',
dtrace: description 'syscall::read:return ' matched 2 probes ^C

DTrace is tracing...

<table>
<thead>
<tr>
<th>value</th>
<th>Distribution</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@</td>
<td>2</td>
</tr>
</tbody>
</table>
| 64    |              | 0     |<-- Ctrl-C to interrupt tracing
BPFTRACE CRASH COURSE

```bash
# bpftrace -e '
    tracepoint:syscalls:sys_exit_read
    /comm == "sshd"/
    {
        @ = hist(args->ret);
    }
',
Attaching 1 probe...
^C
```

```
Attachimg 1 probe...

^C
```

```
@:
[2, 4)  1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4, 8)  0 |
[8, 16) 0 |
[16, 32) 0 |
[32, 64) 1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

```
Attaching 1 probe...

^C
```

```
# bpftrace -e '
    tracepoint:syscalls:sys_exit_read
    /comm == "sshd"/
    {
        @ = hist(args->ret);
    }
',
Attaching 1 probe...
^C
```

```
@:
[2, 4)  1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4, 8)  0 |
[8, 16) 0 |
[16, 32) 0 |
[32, 64) 1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
```

```
Attaching 1 probe...

^C
```
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

<table>
<thead>
<tr>
<th>Static probes</th>
<th>FreeBSD</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Userspace dynamic probes</td>
<td>pid provider¹</td>
<td>uprobe</td>
</tr>
<tr>
<td>Userspace static probes</td>
<td>USDT</td>
<td>USDT</td>
</tr>
<tr>
<td>Kernel dynamic probes</td>
<td>fbt provider²</td>
<td>kprobe</td>
</tr>
<tr>
<td>Kernel static probes</td>
<td>SDT</td>
<td>tracepoint</td>
</tr>
</tbody>
</table>

¹: The pid provider and uprobes are very different.
²: Soon also instructions within functions via the kinst provider.
benchmarks
BENCHMARKS: OVERVIEW

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

- Warmup runs
- Setup & cleanup scripts
- Outliers detection
- 11/10
BENCHMARKS: BENCHMARK RUNS

- TRACING SCRIPT 1
- TRACING SCRIPT 2

workload workload workload workload

---
time
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
dd if=/dev/zero of=/dev/null bs=1 count=10000000
BENCHMARK 1: BACKGROUND

- Measurement 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.
# 06.bt

```bash
u:libc:__read {
  1
}
```

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:

```bash
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)

```c
#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]->ui->ui_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()
}

# 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]->ui0_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;
}
# 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)**

<table>
<thead>
<tr>
<th></th>
<th>Gregg's bpftrace</th>
<th>bpftrace</th>
<th>DTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Control</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Kprobe</td>
<td>78</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>Kretprobe</td>
<td>217</td>
<td>199</td>
</tr>
<tr>
<td>4</td>
<td>Tracepoint Entry</td>
<td>99</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>Tracepoint Return</td>
<td>95</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>Uprobe</td>
<td>1317</td>
<td>1085</td>
</tr>
<tr>
<td>7</td>
<td>Uretprobe</td>
<td>1977</td>
<td>1547</td>
</tr>
<tr>
<td>8</td>
<td>Filter</td>
<td>128</td>
<td>57</td>
</tr>
<tr>
<td>9</td>
<td>Map</td>
<td>194</td>
<td>77</td>
</tr>
<tr>
<td>10</td>
<td>Single Key</td>
<td>212</td>
<td>130</td>
</tr>
<tr>
<td>11</td>
<td>String Key</td>
<td>231</td>
<td>160</td>
</tr>
<tr>
<td>12</td>
<td>Two Keys</td>
<td>234</td>
<td>176</td>
</tr>
<tr>
<td>13</td>
<td>Kernel Stack</td>
<td>344</td>
<td>322</td>
</tr>
<tr>
<td>14</td>
<td>User Stack</td>
<td>668</td>
<td>1077</td>
</tr>
<tr>
<td>15</td>
<td>Histogram</td>
<td>238</td>
<td>133</td>
</tr>
<tr>
<td>16</td>
<td>Timing</td>
<td>651</td>
<td>473</td>
</tr>
<tr>
<td>17</td>
<td>Multiple</td>
<td>856</td>
<td>1264</td>
</tr>
<tr>
<td>18</td>
<td>Per Event</td>
<td>870</td>
<td>1539</td>
</tr>
</tbody>
</table>
## BENCHMARK 1: RELATIVE SLOWDOWN

<table>
<thead>
<tr>
<th></th>
<th>Gregg’s bpftrace</th>
<th>bpftrace</th>
<th>DTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Control</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>2</td>
<td>Kprobe</td>
<td>13 %</td>
<td>21 %</td>
</tr>
<tr>
<td>3</td>
<td>Kretprobe</td>
<td>36 %</td>
<td>76 %</td>
</tr>
<tr>
<td>4</td>
<td>Tracepoint Entry</td>
<td>17 %</td>
<td>28 %</td>
</tr>
<tr>
<td>5</td>
<td>Tracepoint Return</td>
<td>16 %</td>
<td>24 %</td>
</tr>
<tr>
<td>6</td>
<td>Uprobe</td>
<td>221 %</td>
<td>414 %</td>
</tr>
<tr>
<td>7</td>
<td>Uretprobe</td>
<td>331 %</td>
<td>591 %</td>
</tr>
<tr>
<td>8</td>
<td>Filter</td>
<td>21 %</td>
<td>22 %</td>
</tr>
<tr>
<td>9</td>
<td>Map</td>
<td>33 %</td>
<td>29 %</td>
</tr>
<tr>
<td>10</td>
<td>Single Key</td>
<td>36 %</td>
<td>49 %</td>
</tr>
<tr>
<td>11</td>
<td>String Key</td>
<td>39 %</td>
<td>61 %</td>
</tr>
<tr>
<td>12</td>
<td>Two Keys</td>
<td>39 %</td>
<td>67 %</td>
</tr>
<tr>
<td>13</td>
<td>Kernel Stack</td>
<td>58 %</td>
<td>123 %</td>
</tr>
<tr>
<td>14</td>
<td>User Stack</td>
<td>112 %</td>
<td>411 %</td>
</tr>
<tr>
<td>15</td>
<td>Histogram</td>
<td>40 %</td>
<td>51 %</td>
</tr>
<tr>
<td>16</td>
<td>Timing</td>
<td>109 %</td>
<td>181 %</td>
</tr>
<tr>
<td>17</td>
<td>Multiple</td>
<td>143 %</td>
<td>483 %</td>
</tr>
<tr>
<td>18</td>
<td>Per Event</td>
<td>146 %</td>
<td>588 %</td>
</tr>
</tbody>
</table>
BENCHMARK 1: COMPARISON OF PER-EVENT COST: EXPERIMENTS 01–07

Comparison of Per-Event Cost (nsecs)

- Gregg's bpftrace
- bpftrace
- DTrace

<table>
<thead>
<tr>
<th>Method</th>
<th>Control</th>
<th>Kprobe</th>
<th>Kretprobe</th>
<th>Tracepoint Entry</th>
<th>Tracepoint Return</th>
<th>Uprobe</th>
<th>Uretprobe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gregg's bpftrace</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bpftrace</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DTrace</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Gregg's bpftrace

bpftrace

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

Comparison of Per-Event Cost (nsecs)

- Gregg's bpftrace
- bpftrace
- DTrace

Filter: 128, 315, 327
Map: 194, 327, 356
Single Key: 212, 130, 356
String Key: 231, 160, 377
Two Keys: 234, 176, 401
Kernel Stack: 344, 322, 762
User Stack: 1077, 827, 668
Histogram: 238, 360
BENCHMARK 1: COMPARISON OF PER-EVENT COST: EXPERIMENTS 16–18

Comparison of Per-Event Cost (nsecs)

- **Gregg's bpftrace**
- **bpftrace**
- **DTrace**

<table>
<thead>
<tr>
<th>Category</th>
<th>Gregg's bpftrace</th>
<th>bpftrace</th>
<th>DTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing</td>
<td>651</td>
<td>473</td>
<td>682</td>
</tr>
<tr>
<td>Multiple</td>
<td>856</td>
<td>1264</td>
<td>1313</td>
</tr>
<tr>
<td>Per Event</td>
<td>682</td>
<td>1539</td>
<td>312</td>
</tr>
</tbody>
</table>
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.

- bpftrace seems to have a better performance overall than DTrace.

- Per-event cost (last experiment) is surprisingly low on FreeBSD... 😕
make -j 32 buildkernel
BENCHMARK 2: BACKGROUND

- Measurement 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

<table>
<thead>
<tr>
<th>#</th>
<th>fbt</th>
<th>syscall</th>
<th>vfs</th>
<th>sched</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>UFS</td>
<td>all</td>
<td>all</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>UFS-occ</td>
<td>entry</td>
<td>wroc</td>
<td>—</td>
</tr>
<tr>
<td>3</td>
<td>UFS-occ</td>
<td>all</td>
<td>wroc</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>UFS</td>
<td>all</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>5</td>
<td>UFS-occ</td>
<td>entry</td>
<td>wroc</td>
<td>all</td>
</tr>
<tr>
<td>6</td>
<td>UFS-occ</td>
<td>all</td>
<td>wroc</td>
<td>all</td>
</tr>
<tr>
<td>7</td>
<td>UFS-a</td>
<td>all</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8</td>
<td>UFS-abv</td>
<td>all</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>—</td>
<td>all</td>
<td>—</td>
</tr>
</tbody>
</table>
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)

# UFS-occ (6)

# UFS-a (3588)

# UFS-abv (8040)
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

# bpctrace

# all (24)
tracepoint:sched::*

# DTrace

# all (13)
sched:::
results
<table>
<thead>
<tr>
<th></th>
<th>fbt</th>
<th>syscall</th>
<th>vfs</th>
<th>sched</th>
<th>CADETS</th>
<th>DTrace</th>
<th>bpftrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>460</td>
<td>32.92</td>
<td>43.28</td>
</tr>
<tr>
<td>1</td>
<td>UFS</td>
<td>all</td>
<td>all</td>
<td>—</td>
<td>530</td>
<td>36.43</td>
<td>44.05</td>
</tr>
<tr>
<td>2</td>
<td>UFS-occ</td>
<td>entry</td>
<td>wroc</td>
<td>—</td>
<td>460</td>
<td>33.38</td>
<td>43.32</td>
</tr>
<tr>
<td>3</td>
<td>UFS-occ</td>
<td>all</td>
<td>wroc</td>
<td>—</td>
<td>470</td>
<td>33.58</td>
<td>43.46</td>
</tr>
<tr>
<td>4</td>
<td>UFS</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>570</td>
<td>36.62</td>
<td>44.51</td>
</tr>
<tr>
<td>5</td>
<td>UFS-occ</td>
<td>entry</td>
<td>wroc</td>
<td>all</td>
<td>480</td>
<td>33.54</td>
<td>43.41</td>
</tr>
<tr>
<td>6</td>
<td>UFS-occ</td>
<td>all</td>
<td>wroc</td>
<td>all</td>
<td>500</td>
<td>33.69</td>
<td>43.52</td>
</tr>
<tr>
<td>7</td>
<td>UFS-a</td>
<td>all</td>
<td>—</td>
<td>—</td>
<td>570</td>
<td>35.61</td>
<td>49.97</td>
</tr>
<tr>
<td>8</td>
<td>UFS-abv</td>
<td>all</td>
<td>—</td>
<td>—</td>
<td>1210</td>
<td>160.36</td>
<td>62.14</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>—</td>
<td>all</td>
<td>—</td>
<td>550</td>
<td>35.09</td>
<td>43.15</td>
</tr>
</tbody>
</table>
## BENCHMARK 2: RELATIVE SLOWDOWN (OF BEST RUNS)

<table>
<thead>
<tr>
<th></th>
<th>fbt</th>
<th>syscall</th>
<th>vfs</th>
<th>sched</th>
<th>CADETS</th>
<th>DTrace</th>
<th>bpftrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0 %</td>
<td>0 %</td>
<td>0 %</td>
</tr>
<tr>
<td>1</td>
<td>UFS</td>
<td>all</td>
<td>all</td>
<td>—</td>
<td>15 %</td>
<td>10 %</td>
<td>2 %</td>
</tr>
<tr>
<td>2</td>
<td>UFS-occ</td>
<td>entry</td>
<td>wroc</td>
<td>—</td>
<td>0 %</td>
<td>1 %</td>
<td>0 %</td>
</tr>
<tr>
<td>3</td>
<td>UFS-occ</td>
<td>all</td>
<td>wroc</td>
<td>—</td>
<td>2 %</td>
<td>2 %</td>
<td>0 %</td>
</tr>
<tr>
<td>4</td>
<td>UFS</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>24 %</td>
<td>11 %</td>
<td>2 %</td>
</tr>
<tr>
<td>5</td>
<td>UFS-occ</td>
<td>entry</td>
<td>wroc</td>
<td>all</td>
<td>4 %</td>
<td>2 %</td>
<td>0 %</td>
</tr>
<tr>
<td>6</td>
<td>UFS-occ</td>
<td>all</td>
<td>wroc</td>
<td>all</td>
<td>9 %</td>
<td>2 %</td>
<td>1 %</td>
</tr>
<tr>
<td>7</td>
<td>UFS-a</td>
<td>all</td>
<td>—</td>
<td>—</td>
<td>24 %</td>
<td>8 %</td>
<td>16 %</td>
</tr>
<tr>
<td>8</td>
<td>UFS-abv</td>
<td>all</td>
<td>—</td>
<td>—</td>
<td>163 %</td>
<td>386 %</td>
<td>44 %</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>—</td>
<td>all</td>
<td>—</td>
<td>20 %</td>
<td>7 %</td>
<td>0 %</td>
</tr>
</tbody>
</table>
BENCHMARK 2: COMPARISON OF PER-EVENT COST: EXPERIMENTS 1–4

Comparison of Per-Event Cost (nsecs)

CADETS | DTrace | bpftrace
-------|--------|--------
(fbt: UFS, syscall: all, vfs: all) | 15% | 10% | 2%
(fbt: UFS-occ, syscall: entry, vfs: wroc) | 0% | 1% | 0%
(fbt: UFS-occ, syscall: all, vfs: wroc) | 2% | 2% | 0%
(fbt: UFS, syscall: all, vfs: all, sched: all) | 24% | 11% | 2%
BENCHMARK 2: COMPARISON OF PER-EVENT COST: EXPERIMENTS 5–9

Comparison of Per-Event Cost (nsecs)

- CADETS
- DTrace
- bpftrace

(fbt: UFS-occ, syscall: entry, vfs: wroc, sched: all)
(fbt: UFS-occ, syscall: all, vfs: wroc, sched: all)
(fbt: UFS-abv, syscall: all)
(fbt: all)
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

```c
# 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
```

```c
#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

```bash
# 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
```

```c
tracepoint:syscalls:sys_enter_$1 /pid == cpid/ {  
  ...  
}
kprobe:$2 /pid == cpid && @tracing[tid]/ {  
  ...  
}
kretprobe:$2 /pid == cpid && @tracing[tid]/ {  
  ...  
}
tracepoint:syscalls:sys_exit_$1  
/pid == cpid && @tracing[tid]/ {  
  ...  
}
```
conclusion & future work
Overhead is significant but not necessarily expensive ($\leq 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
SPECIAL THANKS

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