Benchmarking Performance Overhead of DTrace on FreeBSD and eBPF on Linux

Mateusz Piotrowski
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin
Berlin, Germany
0mp@FreeBSD.org

***Abstract*—DTrace and eBPF are among the most powerful observability tools available on general-purpose operating systems today, with which users can ask arbitrary questions and receive precise answers about any part of the system. Their performance is unmatched by any traditional observability tools.**

Unfortunately, their performance characteristics are not well-researched. As a result, there are few available papers discussing performance overhead of tracers one could reference for research or for production purposes.

In this paper, we review, design, implement, and conduct a microbenchmark and an application benchmark to peak into the performance overhead of DTrace and eBPF.

Our results provide a data point for future reference and use in research and production environments. Additionally, our results show not only similarities and differences but also strengths and shortcomings of DTrace (on FreeBSD) and eBPF (on Linux), two ecosystems catering to a the same niche in observability.

***Index Terms*—DTrace, eBPF, Linux, FreeBSD, performance engineering, benchmarking, observability**

I. Introduction

Observability is the capacity to ask arbitrary questions and receive complex answers from any given system [4, p. 1]. Traditional observability tools like `iostat`, `procstat`, and `top` provide complex answers for specific parts of the operating system (in case of the listed tools, CPU, I/O, and processes respectively). Tools like DTrace and eBPF allow us to ask arbitrary questions across subsystem boundaries [16, pp. 235–236]. Even though those tracing tools are often advertised as having almost no overhead, their overhead is often non-negligible, which limits the complexity of questions users can actually ask. The overhead impacts the capacity of the tool to fulfill its purpose [19, 13.3 How Much Overhead?].

II. Motivation

Traditional observability tools focus on obtaining specific information about the system. They offer only a limited flexbility as they answer predetermined questions about specific subsystems. Additionally, they have to be in sync with new developments in subsystems and applications to keep up with, e.g., the rising number of metrics to collect. Essentially, the subsystem observation is only possible if the subsystem comes with a dedicated observability tool. Performance-wise, the tools were deemed fast enough if they seemed so to the system operator [6, p. 48].

The situation changed drastically with the emergence of tracers, like DTrace [5] and eBPF [20], which offer the flexibility and performance necessary to explore the system in open-ended ways. As they address the needs of modern observability [15], both their popularity and deployment are on a rise.

Flexibility-wise, modern tracers allow for instrumenting almost any piece of code. There are various caveats to that, however. Instrumenting certain functions may jeopardize the stability of the system. Some subsystems are more tracing-friendly than others, which has more to do with usability than flexibility. Ultimately, however, those are edge cases. The anecdotal proof of the tracer's flexibility are tracer-based reimplementations of traditional observability tools (e.g., truss [8]).

In terms of performance, tracers have been praised for their speed since they were introduced [10]. It is not surprising. Tracer-based tools are generally as performant as their traditional counterparts. In other words, as long as a tracer replaces traditional tools, the user will not notice performance hiccups. The moment we start utilizing the tracer's flexibility is when the overhead increases causing performance to drop. The probe effect is inevitable.

Unfortunately, the impact of the probe effect on the popular workloads is not sufficiently researched. There are very few publications discussing tracers' overhead, which users could refer to when designing production systems.

This research aims to contribute to the collective understanding of the user-facing tracing performance overhead. We seek to provide data points that would assist users in managing their expectations regarding the actual capabilities of tracers. Publishing benchmark results is critical because it expands the base of *known workloads*, which others use for verification during their benchmarking [9, Section 4.6].

III. Benchmark Environemt

The benchmarks were executed on two identical amd64 servers (CPU: 32-core Intel Xeon Gold 6226R; RAM: 376 GiB) configured with FreeBSD 13.1-RELEASE-p1 and Ubuntu 20.04.5 (Linux 5.4.0-139-generic). We followed the recommendations from [3, p. 228] to reduce the measurement noise.

IV. Benchmark 1: dd

In [7, Chapter 18], Brendan Gregg describes a benchmark measuring the eBPF performance overhead, which uses the absolute per-event cost of a tracer action as a metric. The description includes detailed benchmark results. Those results provide an invaluable baseline for further benchmarking of eBPF and other tracers. The high quality of the benchmark description offered by the book is the reason why we selected this benchmark as one of two cornerstones of our research.

The following section will discuss Gregg's benchmark background, design, and implementation. We will explain how we recreated the benchmark setup and ported bpftrace scripts from Linux to FreeBSD DTrace.

A. Gregg's Benchmark Background

Gregg's benchmark focuses on illustrating the cost of an action, which in this case is the handler installed by the tracer at a probe. The workload was chosen to make observing overhead easy by generating events at a measurable frequency.

B. Gregg's Benchmark Design and Implementation

The benchmark uses the standard POSIX dd utility, which copies bytes from one file to another. The complete invocation listed in the original publication is as follows:

```
dd if=/dev/zero of=/dev/null bs=1 count=10000k
```

Listing 1. Gregg's Benchmark Workload

The command copies 10000000 1-byte blocks from `/dev/zero` to `/dev/null`. Reads from `/dev/zero` do not call any file-system functions, and the data does not have to be generated (like in the case of random number generators) or read from an existing file. The 1-byte-block read/write operations make the dd invocation CPU-intensive. As a bonus, 10000000 is an arbitrary number that is sufficiently large to keep a modern CPU busy for a few seconds.

The benchmark defines 18 bpftrace tracing scripts. Each tracing script tests a different tracer feature, like the dynamic kernel instrumentation, the histogram aggregation, or the kernel stack printing. Tested features are discussed in detail in Section IV-D.

Every benchmark run starts with the tracing script. When the tracer is done initializing and is ready to trace, the dd workload is started. The time is measured from the moment of the dd command execution until its return. The workload is bound to a single CPU for consistency.

Perhaps surprisingly for such a simple design, this benchmark checks out all the benchmark quality criteria [13, Section 1.5]:

- Relevance. This benchmark is not only popular in the performance engineering world. It also assesses a wide range of properties by testing different tracer features.
- Reproducibility. We will revisit this quality criterion when comparing our results with Gregg's.
- Fairness. [7] provides a fair amount of information on the setup. There are no secrets to how the results were obtained.
- Verifiability. Since the benchmark setup is simple, anyone can rerun it to verify its results.
- Usability. Again, this benchmark is simple to use and requires little configuration. Its setup is accessible both economically and technologically, as it can be executed even on dated consumer machines.

Gregg's benchmark was executed on Linux 4.15 running on Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz CPUs. The published result of each experiment is the fastest of 10 runs [7].

C. Benchmark 1 Background

To reuse Gregg's benchmark in our research, we had to port the bpftrace script to DTrace. Porting was a relatively easy task as the syntax of bpftrace resembles that of DTrace.

D. Benchmark 1 Design and Implementation

We collected the results using a benchmarking tool called Hyperfine [18]. The tracers were started in Hyperfine's setup phase and stopped in Hyperfine's cleanup phase. DTrace scripts required extra flags and were started like this:

```
dtrace -C -q -D DTRACE_SCRIPT="\"$script\"" -s "↩
    $script"
```

Listing 2. Setup Phase DTrace Launching Command

Flags `-C` and `-D DTRACE_SCRIPT="\"$script\""` are required by Listing 12 (and its function-return counter part).

Let us discuss the process of porting bpftrace scripts to DTrace. This section includes both bpftrace and DTrace implementations of each tracing script. It discusses the implementation differences and challenges we faced during porting. Every pair has an associated test purpose as described in [7]. Note that some test purposes refer to features specific to Linux and bpftrace, which may confuse in the context of FreeBSD (e.g., uprobes).

Most of the bpftrace one-liners use kprobes to tap into the read function of the VFS subsystem. Those kprobes are dynamic kernel probes with an unstable interface [7]. In the world of DTrace, the recommended interface for tracing the VFS subsystem is the vfs provider, which

offers a stable interface to the most interesting parts of the VFS subsystem [10, p. 335]. Using the unstable probes of fbt provider instead of the vfs provider for DTrace scripts may seem suboptimal but considering that the primary goal of the scripts is to test the performance of the tracing infrastructure, we decided to deprioritize platform-specific optimizations and conventions.

Experiment 1 provides the baseline for our measurements (Gregg describes the test purpose of this experiment as "control"). The tracers here run in the background and instrument only their special-purpose `BEGIN` probes. The performance impact of the tracers on the dd workload should be almost non-existent.

The second pair (Listing 3 and Listing 4) tests kprobes. Porting this script to DTrace required that we establish what is an equivalent of kprobes in FreeBSD. Since kprobes are dynamic kernel instrumentation, the most appropriate choice is to use the fbt provider on FreeBSD, which serves the same purpose as kprobes. Another obstacle was the lack of `vfs_read()` function on FreeBSD. We decided that `dofileread()` is a good target function on FreeBSD. Both `vfs_read()` and `dofileread()` are functions called right after the `read()` system call enters the kernel. 1 is a syntactically correct way to ensure that the action body does nothing when a probe fires. This way, only the probe instrumentation overhead is measured. The probe handlers can return immediately once a probe fires.

```
1 k:vfs_read {
2     1
3 }
```

Listing 3. 02.bt

```
1 fbt::dofileread:entry {
2     1
3 }
```

Listing 4. 02.d

The third pair of scripts (Listing 5 and Listing 6) is dedicated to testing kretprobes (the dynamic instrumentation of the kernel function return boundaries). The implementation is similar to the second pair (Listing 3 and Listing 4).

```
1 kr:vfs_read {
2     1
3 }
```

Listing 5. 03.bt

```
1 fbt::dofileread:return {
2     1
3 }
```

Listing 6. 03.d

Pairs 4 (Listing 7 and Listing 8) and 5 (Listing 9 and Listing 10) test the performance of tracepoints, the static kernel instrumentation. On FreeBSD, the various DTrace providers implement this type of interface. For example, the read system call probe is available in the syscall provider.

```
1 t:syscalls:sys_enter_read {
2     1
3 }
```

Listing 7. 04.bt

```
1 syscall:freebsd:read:entry {
2     1
3 }
```

Listing 8. 04.d

```
1 t:syscalls:sys_exit_read {
2     1
3 }
```

Listing 9. 05.bt

```
1 syscall:freebsd:read:return {
2     1
3 }
```

Listing 10. 05.d

Pairs 6 (Listing 11) and 7 (Listing 13) were the most challenging to port over to DTrace. Their test purposes are uprobes and uretprobes, respectively, which provide the dynamic user-space instrumentation. Unfortunately, DTrace on FreeBSD does not provide an equivalent of Linux uprobes. The closest thing is the pid provider, which provides only a subset of uprobe functionalities. In particular, it does not support file-based tracing. As a result, it is not possible to dynamically instrument the user-space code of to-be-started processes on FreeBSD with an elegant one-liner. The pid provider supports tracing already existing processes (DTrace's `-p` flag) and processes started explicitly as a child process of DTrace (the `-c` flag). See lines 37 through 40 of Listing 12 for an example of how the pid provider can use the PID of an existing process.

Unfortunately, this approach does not yield the behavior we need. In our benchmarking setup, we would start the tracing script and let it run in the background as we queue up multiple consecutive runs of the dd workload. This requirement prompted us to seek a way to port the `u:libc:__read(){ 1; }` one-liner to DTrace in the same spirit as other we did for other. The DTrace script's main requirement was to trace all the future dd workload invocations. It means that the PID of the dd workload is unknown if the script starts before the workload. We found a workaround and implemented it in the DTrace port.

The following paragraphs describe the implementation of `06.d`.

The script starts with a pragma that enables destructive actions. Destructive actions must be enabled because we need access to the two destructive functions. The first one, `system()`, is needed to spawn a child DTrace script. The other one, `stop()`, is required to stop the dd workload process temporarily.

Lines 3 through 7 define some handy constants. `TARGET_PROCESS_ARGS` is for the process arguments of the dd workload. We use it to identify the dd workload process. `LIBC_PATH_PREFIX` is the prefix of the location of the libc in the file system. This prefix is needed to identify when the dd workload process loads this shared library.

Lines 9 through 34 implement the parent DTrace script. That script is what is launched by the Hyperfine. When a new dd workload program appears, it aims to spawn a child DTrace script with the dynamic user-space instrumentation (lines 36–47).

Our prototype of the parent DTrace script used the proc provider to instrument the `proc:::exec-success` probe that fires when a process successfully executes a file. The action body of the `proc:::exec-success` probe spawns the child process. Intuitively, the process tracing should start as soon as the PID of the process is known. The `proc:::exec-success` probe sounds like the right moment to instrument the user-space bits with the pid provider. Unfortunately, this approach was wrong because the child DTrace script could not attach to `pid$target:libc*:_read:entry`. The reason was that the libc probes were not yet available. The freshly spawned process had no time to load its shared libraries.

The parent DTrace process must spawn the child DTrace process once libc is loaded to work around the timing issue. In our implementation, we instrument three probe points to detect that. The first probe is `syscall::open:entry` (line 10), which we instrument in such a way that we only execute the action if the process arguments match `TARGET_PROCESS_ARGS` and if the opening file is the library matching `LIBC_PATH_PREFIX`. If those predicates hold, we capture the path of the library in a thread-local `self->path` variable. Thanks to using a thread-local variable, we no longer need to perform expensive string comparisons to identify our process of interest. It is enough to check if the variable is set. The second probe is `syscall::open:return` (line 18). In this probe, we obtain the file descriptor of the opened libc file. The third probe needs the file descriptor because we cannot operate on file names (partially due to the unavailability of the handy `fds` array in the FreeBSD implementation of DTrace [10, p. 323]. The third probe instruments the closing of the libc file via `syscall::close:entry`. Under the assumption that the libc is now loaded by the process and its functions are available in the address space, the parent DTrace process stops the dd workload process. The next step is spawning the child DTrace script. The PID of the dd workload process is passed via the `-p` flag, and `READY_TO_ATTACH` is defined via the `-D` flag for the C preprocessor enabled via the `-C` flag. The child DTrace script is now working, and the parent DTrace script reset the thread-local variables in preparation for another dd workload process.

Let us now turn our focus to the child DTrace script. The tracing of the libc functions is done via the pid provider probes. Most importantly, the child DTrace script exits with the dd workload process. It does so by instrumenting `proc:::exit`. This way, the child DTrace script runs only as long as process it traces.

```
1 u:libc:__read {
2     1
3 }
```

Listing 11. 06.bt

```
1  #pragma D option destructive
2
3  #define TARGET_PROCESS_ARGS     \
4                          "dd if=/dev/zero of=/dev/null↩
       bs=1 count=10000000"
5
6  #define LIBC_PATH_PREFIX        "/lib/libc.so"
7  #define LIBC_PATH_PREFIX_LEN    (sizeof(↩
       LIBC_PATH_PREFIX) - 1)
8
9  #ifndef READY_TO_ATTACH
10 syscall::open:entry
11 /curpsinfo->pr_psargs == TARGET_PROCESS_ARGS && \
12     arg0 != NULL && \
13     substr(copyinstr(arg0), 0, LIBC_PATH_PREFIX_LEN) ↩
       == LIBC_PATH_PREFIX/
14 {
15         self->path = copyinstr(arg0);
16 }
17
18 syscall::open:return
19 /self->path != ""/
20 {
21         self->fd[arg1] = 1;
22 }
23
24 syscall::close:entry
25 /self->fd[arg0] > 0 && self->path != ""/
26 {
27         stop();
28         system("dtrace -C -D READY_TO_ATTACH -p %d -s↩
       %s", \
29             pid, DTRACE_SCRIPT);
30
31         self->path = 0;
32         self->fd[arg0] = 0;
33 }
34 #endif
35
36 #ifdef READY_TO_ATTACH
37 pid$target:libc*:_read:entry
38 {
39         1;
40 }
41
42 proc:::exit
43 /pid == $target/
44 {
45         exit(0);
46 }
47 #endif
```

Listing 12. 06.d

Experiment 7 (`07.bt` and `07.d`) test uretprobes. As a result, their implementation is very similar to Listing 11 and Listing 12. Therefore, the listing of `07.d` is omitted from this section for brevity.

```
1 ur:libc:__read {
2     1
3 }
```

Listing 13. 07.bt

All the remaining experiments focus on the performance of the tracer's features.

The test purpose of experiment 8 is filter performance. In the DTrace world, filters are called predicates.

The functionality selected for experiment 8 is filtering out reads of 0 bytes. In contrast to Linux, where the read size is stored in `arg2`, scripts on FreeBSD must obtain that information from `args[3]->uio_resid` [10].

```
k:vfs_read /arg2 > 0/ {
    1
}
```

Listing 14. 08.bt

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

Listing 15. 08.d

Experiments 9 through 15 test various elements of aggregations. Experiment 9 focuses on maps (called aggregations in DTrace). Experiment 10 introduces a single numerical key to the equation. Experiment 11 tests the performance of an aggregation where the key is a string. At least in DTrace, strings are known to be noticeably less performant than numerical types like integers. Experiment 12 evaluates the performance overhead of using maps with two keys. Experiment 13 counts how often identical kernel stack traces lead to the execution of the `vfs_read()` function. Experiment 14 does the same, but for user stack traces. Experiment 15 tests the performance of the histogram aggregation. This type of aggregation is more complex than the simple count aggregation. The following listings present their implementations.

```
k:vfs_read {
    @ = count()
}
```

Listing 16. 09.bt

```
fbt::dofileread:entry {
    @ = count()
}
```

Listing 17. 09.d

```
k:vfs_read {
    @[pid] = count()
}
```

Listing 18. 10.bt

```
fbt::dofileread:entry {
    @[pid] = count()
}
```

Listing 19. 10.d

```
k:vfs_read {
    @[comm] = count()
}
```

Listing 20. 11.bt

```
fbt::dofileread:entry {
    @[execname] = count()
}
```

Listing 21. 11.d

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

Listing 22. 12.bt

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

Listing 23. 12.d

```
k:vfs_read {
    @[kstack] = count()
}
```

Listing 24. 13.bt

```
fbt::dofileread:entry {
    @[stack()] = count()
}
```

Listing 25. 13.d

```
k:vfs_read {
    @[ustack] = count()
}
```

Listing 26. 14.bt

```
fbt::dofileread:entry {
    @[ustack()] = count()
}
```

Listing 27. 14.d

```
k:vfs_read {
    @ = hist(arg2)
}
```

Listing 28. 15.bt

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

Listing 29. 15.d

Experiment 16 recreates one of the commonly used idioms in the bpftrace/DTrace programmer's toolkit. The timing of a function execution is an example of how tracers can answer complex questions about the system's performance. Tracers can precisely measure time between events. In the case of experiment 16, the tracer can calculate the time between two events.

```
k:vfs_read {
    @s[tid] = nsecs
}

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

Listing 30. 16.bt

```
fbt::dofileread:entry {
    self->s = timestamp
}

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

Listing 31. 16.d

The test purpose of experiment 17 is "multiple". This pair of scripts measures the combined performance impact of using some computation-heavy elements of the tracer's language, such as histograms and stack traces at the same time

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

Listing 32. 17.bt

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

Listing 33. 17.d

Finally, experiment 18 looks into the performance overhead of per-event printing. This one is another commonly employed use case.

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

Listing 34. 18.bt

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

Listing 35. 18.d

E. Benchmark 1 Results and Analysis

Table I and Table II contain our Benchmark 1 results. Table III is a copy of Gregg's results from [7]. The published result of each experiment is the fastest of all runs, to follow the principle of least perturbations [7].

Table IV contains the table of the per-event cost results of both our and Gregg's results.

Table V illustrates the relative slowdown.

Let us consider Table IV.

It is important to remember that we cannot tell which system is faster than another. There are many factors we did not take into consideration (e.g., compiler version). Any performance comparison between DTrace and bpftrace would be unfair and misinformed.

Our bpftrace results show that most experiments show a lower per-event cost than Gregg's results. We expected that because tracing infrastructure is improving fast and getting better and better with each release. Additionally, we use bpftrace 0.17.0, which is most likely more optimized than the older versions used by Gregg. Gregg describes the cost of around 80 nanoseconds as "fast" [7]. It is excellent that the kprobe experiment is still under 80 nanoseconds. Also, none of the experiments fell below that arbitrary threshold. The performance of instrumenting tracepoint entries and returns is improving. Another good sign is that the kprobes are still less costly than kreprobes. Similarly, tracepoint entries are still more expensive than tracepoint returns. Uprobes, famously slow on Linux, perform better in our runs. Moreover, uretprobes are still slower than uprobes, which is to be expected.

The troubling results are those of experiment 14 (User Stack). It needs to be clarified why reading the user stack is so slow in the case of our bpftrace runs. Based on the standard deviation of this experiment, we suspect that something unusual is happening on the system. The measurements of individual runs are always around 13 or 17 seconds and never in between. Experiment 17 (Multiple) is also most likely impacted by the slow user stack functionality.

Experiment 18 (Per Event) is way slower than in Gregg's case. We suspect that this might be a telltale of a misconfigured experiment case. The redirection instructions in Greggs book for experiment 18 were not clear to us.

The standard deviation of DTrace results is within reasonable according to Hyperfine. The per-event cost of DTrace runs is surprising. DTrace performed much worse than bpftrace, but as we mentioned earlier, we cannot draw any interesting conclusions from that alone. There are promising bits in DTrace's results, however. The kernel instrumentation (dynamic and static) performs very similarly for both entry and return probes. This result contrasts with Linux readings. We suspect this results from a different design of the in-kernel instrumentation. Also, the static probes are more performant than the dynamic ones (fbt), which was expected. By comparing the experiments with and without actions in their clauses, we can see that actions cause extra overhead in addition to probes. Amazingly, experiments 6 (Uprobe) and 7 (Uretprobe), which used a nested DTrace invocation to trace libc functions, are still more performant regarding the absolute number of nanoseconds than uprobes on Linux, which contrasts with other experiments. We expected to see an enormous performance hit in the case of experiments 6 and 7, as the traced program was temporarily stopped to let the child DTrace script spawn and attach to the traced process.

Table V shows the relative slowdown of experiments relative to the baseline run. From this perspective, our bpftrace runs exhibited a more considerable slowdown than Gregg's results. There might be a couple of reasons for this. E.g., perhaps there are bottlenecks in the BPF subsystem, which result in BPF not benefitting from faster hardware.

Experiment	Runtime (s)	Per-Event Cost (ns)
01 Control	2.61958	0
02 Kprobe	3.17729	56
03 Kretprobe	4.61308	199
04 Tracepoint Entry	3.36088	74
05 Tracepoint Return	3.24673	63
06 Uprobe	13.46601	1085
07 Uretprobe	18.08874	1547
08 Filter	3.19123	57
09 Map	3.39113	77
10 Single Key	3.91503	130
11 String Key	4.22169	160
12 Two Keys	4.37811	176
13 Kernel Stack	5.83544	322
14 User Stack	13.38925	1077
15 Histogram	3.94975	133
16 Timing	7.35197	473
17 Multiple	15.25990	1264
18 Per Event	18.01106	1539

Table I
Benchmark 1 bpftrace Results

Experiment	Runtime (s)	Per-Event Cost (n)
01 Control	2.31770	0
02 Kprobe	5.12040	280
03 Kretprobe	5.16942	285
04 Tracepoint Entry	3.07915	76
05 Tracepoint Return	3.05219	73
06 Uprobe	10.85424	854
07 Uretprobe	16.48251	1416
08 Filter	5.47263	315
09 Map	5.58700	327
10 Single Key	5.88025	356
11 String Key	6.08666	377
12 Two Keys	6.32500	401
13 Kernel Stack	9.94135	762
14 User Stack	10.58532	827
15 Histogram	5.91704	360
16 Timing	9.13756	682
17 Multiple	15.44620	1313
18 Per Event	5.43515	312

Table II
Benchmark 1 DTrace Results

Experiment	Runtime (s)	Per-Event Cost (ns)
01 Control	5.97243	0
02 Kprobe	6.75364	78
03 Kretprobe	8.13894	217
04 Tracepoint Entry	6.95894	99
05 Tracepoint Return	6.92440	95
06 Uprobe	19.14660	1317
07 Uretprobe	25.74360	1977
08 Filter	7.24849	128
09 Map	7.91737	194
10 Single Key	8.09561	212
11 String Key	8.27808	231
12 Two Keys	8.31670	234
13 Kernel Stack	9.41422	344
14 User Stack	12.64800	668
15 Histogram	8.35566	238
16 Timing	12.48160	651
17 Multiple	14.53060	856
18 Per Event	14.67190	870

Table III
Benchmark 1 Gregg's Results

V. Benchmark 2: FreeBSD Kernel Compilation

The second benchmark we selected was a FreeBSD kernel compilation. Even though this may sound like an unusual pick, it is one of the few recently published DTrace benchmarks we could find. The benchmark is contained in the CADETS technical report [23] and designed for measuring DTrace's performance overhead on FreeBSD.

The following sections describe the CADETS benchmark's background, design, and implementation. Later, we introduce our benchmark and explain how we recreated and expanded the CADETS benchmark. Ultimately, we present our results and compare them to those of CADETS'.

A. CADETS Benchmark Background

The CADETS technical report [23] describes a DTrace-based system providing provenance data for causal backtracking and temporal pattern matching. What is of great relevance to our research (and Benchmark 2 in particular) is that the system's implementation described in the report uses DTrace as an always-on observability mechanism to collect the necessary provenance data by continuously tracing a large number of probes. This use case naturally tests DTrace's reliability, performance, and ability to collect data from multiple sources simultaneously.

Experiment	Gregg's bpftrace (ns)	bpftrace (ns)	DTrace (ns)
01 Control	0	0	0
02 Kprobe	78	56	280
03 Kretprobe	217	199	285
04 Tracepoint Entry	99	74	76
05 Tracepoint Return	95	63	73
06 Uprobe	1317	1085	854
07 Uretprobe	1977	1547	1416
08 Filter	128	57	315
09 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

Table IV
Comparison of Per-Event Costs

Experiment	Gregg's bpftrace	bpftrace	DTrace
01 Control	0%	0%	0%
02 Kprobe	13%	21%	121%
03 Kretprobe	36%	76%	123%
04 Tracepoint Entry	17%	28%	33%
05 Tracepoint Return	16%	24%	32%
06 Uprobe	221%	414%	368%
07 Uretprobe	331%	591%	611%
08 Filter	21%	22%	136%
09 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%

Table V
Relative Slowdown

Regarding the workload, the FreeBSD kernel compilation has a long history of being used as a benchmark workload [24, 17, 21, 23, 12, 25].

It is not surprising:

- The kernel compilation is familiar. It is a frequently performed operation in the development process of an operating system. At the same time, it is a common step during the system configuration step among downstream consumers. As a result, the kernel recompilation process and characteristics are commonly understood by both the developers and operators.
- The kernel compilation performance is relevant. The reason is that it is rebuilt not only in development but also later on during system integration. This trend leads to the speed of kernel compilation performance being the focal point of various projects (e.g., [2] is a recent example in the Linux kernel community). Also, the kernel compilation performance can be the smoking gun evidence of a recent performance regression [11].
- The kernel compilation puts various parts of the operating system under test.

The kernel compilation tests the following system characteristics [24]: effective CPU utilization, I/O performance, and file system metadata performance. Its performance is the wall clock execution duration of the `make buildkernel` command.

The FreeBSD kernel compilation fulfills the following benchmark quality criteria from [13, Section 1.5]:

- Relevance. The FreeBSD kernel compilation is a real-world workload [14]. Additionally, it may act as a macrobenchmark for system throughput evaluation [24].
- Reproducibility. The FreeBSD kernel compilation process is designed to produce the same results every time. Additionally, the FreeBSD build process can be configured to provide reproducible builds. This guarantees that not only the process of building the kernel is the same every time, but also the build artifacts are always bit-for-bit identical.
- Fairness. Both the code and the documentation of the build process are open-source. The FreeBSD build process has no secrets that would disadvantage anyone willing to execute this benchmark.
- Verifiability. The setup is simple and well-documented, which improves verifiability. Unfortunately, since the FreeBSD kernel compilation is not designed to be a benchmark, it does not store any extra statistics about its run that would help verify the published results.
- Usability. The FreeBSD project supports building the FreeBSD kernel on various architectures and platforms. The kernel is expected to be buildable even on low-end consumer-market machines like older laptops. Additionally, FreeBSD can be built on Linux and macOS.

B. CADETS Benchmark Design and Implementation

The authors measured the performance overhead of the DTrace instrumentation by conducting a series of measurements of the FreeBSD kernel compilation time in four different tracing setup variants [23, pp. 40–42]:

- Auditing Script Variant is a kernel compilation with a special auditing script. The script can be viewed in one of the CADETS' Git repositories [1].
- Baseline Variant has tracing turned off. This is considered the baseline measurement to compare other runs against.
- HyperTrace Variant traces the kernel compilation, but the tracer runs on the host and traces the virtual machine via a mechanism called HyperTrace [23, 22].
- Tracing-Enabled Variant traces the kernel compilation. In this case, the tracer and the compilation process run on the same machine, the most common tracing setup described in this benchmark. The machine is a virtual machine.

We are going to focus only on the baseline and tracing-enabled variants. The other two are outside our research.

Unfortunately, the report does not explain many essential details of the system setup. The authors only mention that they used a bhyve virtual machine with 4 CPUs and 16 GB RAM for the scenario involving the auditing script. Additional information can be implied from the table listing different tracing scripts [23, Figure 16 Varying Levels of Granularity for Different Traces]. For example, we suspect the file system used in the setup was UFS.

The workload of the benchmark consists of two elements. There is the FreeBSD kernel compilation job and the tracing script. The FreeBSD kernel compilation job is not explained in detail. Based on the FreeBSD documentation, we can assume that the command used was `make buildkernel`. The FreeBSD version used for building is not mentioned explicitly (the CADETS project mentions using FreeBSD versions 11, 12, and 13 at different stages of their research).

The only documentation of the tracing scripts is the table listing various probes to which each script must attach [23, Figure 16]. The probe definitions could be more precise but are descriptive enough for us to recreate those probe definitions in the D language. However, one crucial bit is missing. Namely, we do not have the action implementation. The report does not explain what action was performed by the scripts when a probe matched. In a private communication with a CADETS contributor, we learned that the action was likely a simple operation. In other words, it was neither an empty nor a complex action that would be a significant source of overhead.

The effect of many attached probes is what was meant to be benchmarked. Every tracing script is described in terms of four DTrace providers: fbt, syscall, vfs, and

sched. The description mentions what probes a script is attaching to for a given provider. For example, the "Trace 1" tracing script is described in the following way: fbt: *UFS*; syscall: *all*; vfs: *all*; sched: *none*. Those descriptions are generally easy to understand for users with some DTrace experience. The "UFS" description for the fbt provider requires slightly more knowledge of the FreeBSD kernel internals. We assume that the authors meant a DTrace probe description of `fbt::ufs_*` because most of the UFS-related functions in the FreeBSD kernel start with `ufs_`. This aligns with how other published UFS tracing scripts [10, pp. 352–354].

According to the technical report [23, p. 41], the average overhead across all the tested tracing scripts was 28% as compared to the baseline measurements with tracing disabled.

C. Benchmark 2 Background

We created a benchmark following the design and implementation described in the CADETS technical report.

D. Benchmark 2 Design and Implementation

We had two machines available for the benchmark. They are described in detail in Section III.

We decided to use the source code of FreeBSD 13.1-RELEASE associated with commit hash `fc952ac2212b121aa6eefc273f5960ec3e0a466d`. This version of FreeBSD was selected for various reasons. First, this is the most recent release of FreeBSD, containing all sorts of improvements for building FreeBSD on Linux. Second, it was the version that could be easily compiled on the Ubuntu version we had configured. Lastly, FreeBSD 13 was the newest major release at the time of writing, making it easier for interested parties to reproduce our results on a supported release.

The kernel compilation job was the `make buildkernel` command executed in the FreeBSD source directory. In order to remove the physical disk I/O perturbations from the performance overhead equation, we decided to use memory-backed disks. FreeBSD command providing this functionality is `mdconfig`. In the Linux world, `modprobe brd` is a functional equivalent. The source directory was extracted onto a 40 GB memory-backed disk formatted with a traditional file system native to the operating system. On FreeBSD, that file system was UFS (via `newfs`). On Linux, it was XFS (via `mkfs.xfs`). The rationale for selecting XFS instead of the vastly more popular ext4 is given later in this section.

Regarding the tracing scripts, we recreated the DTrace scripts based on the descriptions from the CADETS report.

The following listings contain our interpretation of the CADETS probe descriptions. The bpftrace probe descriptions were developed by us to match the DTrace probes descriptions functionality-wise as close as possible.

Let us start with the fbt provider (Listing 37, Listing 38, Listing 39, Listing 40, Listing 41, Listing 42, Listing 43, and Listing 44). The CADETS authors use the fbt provider to instrument two categories of functions: the UFS file system functions (either all or a small subset) and any functions starting with a given letter. To port fbt provider probes to Linux, we use kprobes, the dynamic kernel instrumentation equivalent for Linux. Since our fbt probes attach implicitly to both function entries and returns, we must also instrument kretprobes. These functionalities translate easily to bpftrace.

The real obstacle was translating UFS tracing. Even though Linux offers some support for this file system, it cannot be considered the equivalent of UFS on FreeBSD. For that purpose, ext4 would be a much more appropriate choice. Unfortunately, we could not use ext4 because the server's root partition was already using ext4. Using ext4 would result in more perturbations in measurements. The instrumented probes would match file-system function calls not only from the workload but also from the system services. A remedy for that was choosing XFS as the file system for the benchmarking setup.

Instrumenting open, close, and create function calls of a file system is more complex in the case of XFS (and of ext4 equally so) as it is in the case of UFS. The equivalent of `ufs_open()` in the world of XFS is not a single function, but two: `xfs_file_open()` and `xfs_dir_open()`. The porting of `ufs_close()` is even more problematic as Linux file systems usually do not define close functions. The simple closing of a file is handled without ever calling file-system-specific code and is done by `fput()`. Only when the last reference to a file is released the Linux kernel calls the release function of the file system. The lack of a file-system-specific function is unfortunate as `fput()` will also match for non-XFS files, increasing the performance overhead. Listing 36 showcases a potential workaround for that problem. To keep the setup lean, we decided to let our benchmark traces XFS close operations with `kprobe:fput` without additional filtering.

```
#include <linux/fs.h>

kprobe:fput
/kaddr("xfs_file_operations") == ((struct file *)arg0)->f_op/
{
        @[probe] = count();
}
```

Listing 36. Tracing Closing of XFS Files Only

```
fbt::ufs_*:
```

Listing 37. DTrace: fbt: UFS

```
kprobe:xfs_*,
kretprobe:xfs_*
```

Listing 38. bpftrace: fbt: UFS

```
fbt::ufs_open:,
fbt::ufs_close:,
fbt::ufs_create:
```

Listing 39. DTrace: fbt: UFS (open, close, create)

```
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
```

Listing 40. bpftrace: fbt: UFS (open, close, create)

```
fbt::ufs_*:,
fbt::a*:
```

Listing 41. DTrace: fbt: UFS, a*

```
kprobe:xfs_*,
kretprobe:xfs_*,
kprobe:a*,
kretprobe:a*
```

Listing 42. bpftrace: fbt: UFS, a*

```
fbt::ufs_*:,
fbt::a*:,
fbt::b*:,
fbt::v*:
```

Listing 43. DTrace: fbt: UFS, a*, b*, v*

```
kprobe:xfs_*,
kretprobe:xfs_*,
kprobe:a*,
kretprobe:a*,
kprobe:b*,
kretprobe:b*,
kprobe:v*,
kretprobe:v*
```

Listing 44. bpftrace: fbt: UFS, a*, b*, v*

Scheduler instrumentation naturally translates from the DTrace sched provider static instrumentation to Linux `tracepoint:sched` (Listing 45 and Listing 46).

```
sched:::
```

Listing 45. DTrace: sched: all

```
tracepoint:sched:*
```

Listing 46. bpftrace: sched: all

System call instrumentation relies on static kernel instrumentation. As a result porting related probe descriptions to Linux was as easy as in the case of scheduler probes.

```
syscall:::
```

Listing 47. DTrace: syscall: all

```
tracepoint:syscalls:*
```

Listing 48. bpftrace: syscall: all

```
syscall:::entry
```

Listing 49. DTrace: syscall: all (entry)

```
tracepoint:syscalls:sys_enter_*
```

Listing 50. bpftrace: syscall: all (entry)

The translation of VFS probe description to Linux was more challenging than expected. Even though VFS is one of the essential subsystems of a modern general-purpose kernel, the Linux kernel currently does not provide a stable tracing interface to VFS functions. In contrast, FreeBSD ships with a dedicated vfs provider. However, on Linux, the VFS function must be instrumented with kprobes. The last obstacle was selecting a function to trace that would be the equivalent of FreeBSD's `vfs_close()`. We decided to select `__close_fd()`, as `fput()` is already instrumented by the XFS probes.

```
vfs:::
```

Listing 51. DTrace: vfs: all

```
kprobe:vfs_*,
kretprobe:vfs_*
```

Listing 52. bpftrace: vfs: all

```
vfs::vfs_write:,
vfs::vfs_read:,
vfs::vfs_open:,
vfs::vfs_close:
```

Listing 53. DTrace: vfs: write, read, open, close

```
kprobe:vfs_write,
kretprobe:vfs_write,
kprobe:vfs_read,
kretprobe:vfs_read,
kprobe:vfs_open,
kretprobe:vfs_open,
kprobe:__close_fd,
kretprobe:__close_fd
```

Listing 54. bpftrace: vfs: write, read, open, close

Similarly to Benchmark 1, we used Hyperfine [18] to conduct Benchmark 2. The benchmark executed Hyperfine once for each tracing script. The setup phase of Hyperfine downloaded the FreeBSD source tree, extracted it onto the in-memory disk, and bootstrapped the necessary tools by executing the `kernel-toolchain` target. The last step of the setup phase was to launch the tracing script. Interestingly, the bpftrace scripts reached the safety limits of the interpreter. To proceed with Benchmark 2, we had to increase the system limit on the number of allowed open file descriptors to 200000 and set environment variables `BPFTRACE_MAX_BPF_PROGS` and `BPFTRACE_MAX_PROBES` to 22000. We configured Hyperfine to schedule at least one warmup run to warm up the cache and give the tracer more time to attach. The tracer startup takes up to a few seconds, but based on our observations, the tracers were attached and ready when the measured runs started.

With the setup finished, Hyperfine would start running

the kernel compilation jobs. The compilation jobs used all 32 cores.

Before each new kernel compilation run, Hyperfine would clean the object directory and rerun the `kernel-toolchain` target. This way every run started with the same object directory.

The tracing script would terminate in the cleanup phase of Hyperfine.

On FreeBSD, the compilation job used the LLVM toolchain shipped with FreeBSD 13.1-RELEASE. On Linux, LLVM 12 was selected out of necessity as we experienced build failures when using other LLVM versions.

E. Benchmark 2 Results and Analysis

Table VI contains the average kernel build time of our and CADETS' results. Table VII illustrates the relative slowdown.

Naturally, our results are way ahead of the CADETS results because we built the kernel with 32 CPUs instead of 4. The FreeBSD kernel build parallelizes to 32 cores, as we see over eight times faster build times.

We can make a couple of observations when comparing the CADETS' results with our DTrace results. Experiments 1, 4, and 9 (vfs: all) are the most challenging to FreeBSD. The performance of Linux does not suffer as much as FreeBSD in those experiments. One of the potential explanations is that the vfs matches many more functions on FreeBSD than kprobes do on Linux.

In Table VII, the negative slowdowns of bpftrace are visible. Of course, it is doubtful that the FreeBSD kernel compilation speeds up when bpftrace traces over 1000 probes. We suspect that there is something in the architecture of bpftrace that causes results to show such a low overhead. During the benchmark setup, we discovered that at the end of all experiment 8 runs, bpftrace needed 10 minutes to terminate. DTrace, in comparison, needed only a couple of seconds. If Benchmark 2 was modified to require tracers to report collected statistics, we would likely see completely different results.

	fbt	syscall	vfs	sched	CADETS	bpftrace	DTrace
0	—	—	—	—	460.00	43.28	32.92
1	UFS	all	all	—	530.00	44.05	36.43
2	UFS-occ	entry	wroc	—	460.00	43.32	33.38
3	UFS-occ	all	wroc	—	470.00	43.46	33.58
4	UFS	all	all	all	570.00	44.51	36.62
5	UFS-occ	entry	wroc	all	480.00	43.41	33.54
6	UFS-occ	all	wroc	all	500.00	43.52	33.69
7	UFS-a	all	—	—	570.00	49.97	35.61
8	UFS-abv	all	—	—	1210.00	62.14	160.36
9	—	—	all	—	550.00	43.15	35.09

Table VI
Average Kernel Build Time (seconds)

VI. Conclusion

Our results show that DTrace and bpftrace are entirely different pieces of software, even though they might look similar.

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

Table VII
Relative Slowdown

Many engineers talk about the performance of observability tools but, unfortunately, rarely publish their measurements. We believe that our performance overhead benchmarking results will help users design better systems and allow developers to have a more open discussion about the performance overhead of tracers.

VII. Future Work

A. Stability of Tracers

In Section V, we focused on the runtime overhead of the workload program caused by the enabled instrumentation. What needed to be verified was whether tracers stayed usable under load. For example, during initial experimentation, we discovered that it took about 10 minutes for bpftrace to print its collected statistics and quit. We noticed that our scripts for the cleanup phase had to wait way longer on Linux than on FreeBSD for the tracer to quit. In comparison, when signaled, DTrace would terminate in a few seconds. Measuring and understanding this behavior might benefit the bpftrace community if the problematic bottleneck gets identified.

B. Count the Number of Probes in Benchmark 2

Our design of Benchmark 2 did not consider that the number of probe activations should be recorded. As a result, we cannot calculate the per-event cost for Benchmark 2. This was done to minimize the number of perturbations. We did not want tracers to pollute the results by writing their output to files. Instead, their output was discarded.

Future iterations of Benchmark 2 should provide for collecting those statistics.

C. Measuring DTrace and bpftrace with KUtrace

KUtrace is a new generation of observability tools. It is designed to troubleshoot performance issues of highly optimized and complex systems. It could be used to measure the overheads of DTrace and bpftrace.

References

[1] Thomas Arun, Amanda Strnad, George V. Neville-Neil, and Jonathan Anderson. *cadets/dtrace-scripts: A collection of DTrace scripts*. Jan. 2019. URL: https://github.com/cadets/dtrace-scripts (visited on 01/25/2023).

[2] Jean-Luc Aufranc. "The Linux kernel could soon be 50 to 80% faster to build". In: *CNX Software - Embedded Systems News* (Jan. 2022). URL: https://www.cnx-software.com/2022/01/04/the-linux-kernel-could-soon-be-50-to-80-faster-to-build/ (visited on 03/07/2023).

[3] Denis Bakhvalov. *Performance Analysis and Tuning on Modern CPUs*. Nov. 2020. URL: https://book.easyperf.net/perf_book.

[4] David Calavera and Lorenzo Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. First edition. Beijing [China] ; Sebastopol, CA: O'Reilly Media, Inc, 2019. ISBN: 978-1-4920-5020-9.

[5] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. "Dynamic Instrumentation of Production Systems". In: (2004).

[6] Adrian Cockcroft and Richard Pettit. *Sun Performance and Tuning: Java and the Internet*. 2nd ed. Englewood Cliffs, N.J: Prentice Hall PTR, 1998. ISBN: 978-0-13-095249-3.

[7] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. 1st ed. Hoboken: Pearson Education, Inc, 2019. ISBN: 978-0-13-655482-0. URL: https://www.brendangregg.com/bpf-performance-tools-book.html (visited on 01/17/2023).

[8] Brendan Gregg. *dtruss_example.txt*. Apr. 2016. URL: https://github.com/opendtrace/toolkit/blob/master/Examples/dtruss_example.txt (visited on 03/09/2023).

[9] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Second. Boston: Addison-Wesley, 2020. ISBN: 978-0-13-682015-4.

[10] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. OCLC: ocn693205642. Upper Saddle River, NJ: Prentice Hall, 2011. ISBN: 978-0-13-209151-0.

[11] Mateusz Guzik. *Abysmal performance [...]* Sept. 2022. URL: https://github.com/openzfs/zfs/issues/13932 (visited on 03/11/2023).

[12] Tom Jones. *FreeBSD on the Intel 10th Gen i3 NUC*. June 2020. URL: https://adventurist.me/posts/00300 (visited on 01/25/2023).

[13] Samuel Kounev, Klaus-Dieter Lange, and Jóakim Gunnarsson von Kistowski. *Systems Benchmarking: For Scientists and Engineers*. Cham, Switzerland: Springer, 2020. ISBN: 978-3-030-41704-8. DOI: 10.1007/978-3-030-41705-5.

[14] Scott Long. *Modifying the FreeBSD kernel Netflix streaming*. Mar. 2014. URL: https://www.youtube.com/watch?v=4sZZN8Szh14 (visited on 03/11/2023).

[15] Charity Majors, Liz Fong-Jones, and George Miranda. *Observability Engineering: Achieving Production Excellence*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2022. ISBN: 978-1-4920-7644-5.

[16] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. 1st ed. Sun Microsystems Press/Prentice Hall, 2006. ISBN: 978-0-13-156819-8.

[17] George Mitchell. *SCHED_ULE should not be the default*. Dec. 2011. URL: https://lists.freebsd.org/pipermail/freebsd-stable/2011-December/064773.html (visited on 01/25/2023).

[18] David Peter. *hyperfine*. original-date: 2018-01-13T15:49:54Z. Sept. 2022. URL: https://github.com/sharkdp/hyperfine (visited on 03/10/2023).

[19] Richard L. Sites. *Understanding Software Dynamics*. First edition. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2021. ISBN: 978-0-13-758973-9.

[20] Alexei Starovoitov. *extended BPF*. Sept. 2013. URL: https://lkml.org/lkml/2013/9/30/627 (visited on 01/26/2023).

[21] Murray Stokely. *FreeBSD on Intel NUCs*. Oct. 2016. URL: http://freebsd.stokely.org/2016/10/freebsd-on-intel-nucs.html (visited on 01/25/2023).

[22] Domagoj Stolfa. "DTrace: New Additions to an Old Tracing System". In: *FreeBSD Journal* Observability and Metrics: November/December 2022 (Jan. 2023), pp. 13–21. URL: https://freebsdfoundation.org/wp-content/uploads/2023/01/stolfa_dtrace.pdf (visited on 01/25/2023).

[23] Amanda Strnad, Quy Messiter, Robert Watson, Lucian Carata, Jonathan Anderson, and Brian Kidney. *Casual, Adaptive, Distributed, and Efficient Tracing System (CADETS)*. Tech. rep. BAE Systems, Sept. 2019. URL: https://apps.dtic.mil/sti/citations/AD1080643 (visited on 08/11/2022).

[24] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0". In: 2003.

[25] Weixi Zhu, Alan L. Cox, and Scott Rixner. "A Comprehensive Analysis of Superpage Management Mechanisms and Policies". In: (July 2020).