Using eBPF for Database Workload Tracing: An Explorative Study

Jörg Domaschka*
Simon Volpert*
joerg.domaschka@uni-ulm.de
simon.volpert@uni-ulm.de
Ulm University, Ulm
Germany

Kevin Maier
Georg Eisenhart
kevin.maier@uni-ulm.de
georg.eisenhart@uni-ulm.de
Ulm University, Ulm
Germany

Daniel Seybold
daniel.seybold@benchant.com
benchANT GmbH
Ulm, Germany

ABSTRACT

Database management systems (DBMS) are crucial architectural components of any modern distributed software system. Yet, ensuring a smooth, high-performant operation of a DBMS is a black art that requires tweaking many knobs and is heavily dependent on the experienced workload. Misconfigurations at production systems have an heavy impact on the overall delivered service quality and hence, should be avoided at all costs. Replaying production workload on test and staging systems to estimate the ideal configuration are a valid approach. Yet, this requires traces from the production systems.

While many DBMS's have built-in support to capture such traces these have a non-negligible impact on performance. eBPF is a Linux kernel feature claiming to enable low-overhead observability and application tracing. In this paper, we evaluate different eBPF-based approaches to DBMS workload tracing for PostgreSQL and MySQL. The results show that using eBPF causes lower overhead than the built-in mechanisms. Hence, eBPF can be a viable baseline for building a generic tracing framework. Yet, our current results also show that additional optimisation and fine-tuning is needed to further lower the performance overhead.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computing methodologies** → *Modeling methodologies.*

KEYWORDS

DBMS, eBPF, Benchmarking, Cloud

ACM Reference Format:
Jörg Domaschka, Simon Volpert, Kevin Maier, Georg Eisenhart, and Daniel Seybold. 2023. Using eBPF for Database Workload Tracing: An Explorative Study. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578245.3584313

*All authors contributed equally to this research.

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04.
https://doi.org/10.1145/3578245.3584313

1 INTRODUCTION

The importance and prevalence of Database Management Systems (DBMS) is ubiquitous. Acting as a backbone of many modern distributed systems they directly impact performance as well as reliability and thus user experience. Consequently, the assessment of performance under various conditions has always been a widely regarded research topic [18] and continues to the present day [3]. Yet, this research is not limited to mere performance engineering, but covers of many different aspects of databases including non-functional ones such as availability and consistency [5].

Most methods for analysing DBMS need some kind of workload that can be enacted on a target DBMS server. Basically, this workload can either be generated synthetically or traced from an actual real workload. While there are some well known and highly regarded synthetic benchmark and workload generator tools available [19], they impose the challenge to model production workload on benchmark parameters. What is more, this modelling will fail, if circumstances require to reproduce a real production workload. In these cases, replaying real workload traces is the only possible approach. While such traces can conceptually be manually crafted, this is seldom a realistic option due to time and effort required, but also because changes in workload cannot be captured easily. Alternatively, a workload trace needs to be gathered from production systems leaving the question of how to acquire such a trace.

Most, if not all production grade DBMS come with support for workload tracing. Yet, this tracing feature is supposed to cause massive overhead and is usually not recommended for running in production [12]. Hence, if traces are collected at all, they span relatively short time spans and cannot be used for continuous monitoring, validation, and optimisation. In this paper, we shall evaluate to what extent Extended Berkeley Packet Filter (eBPF) can be used for gathering workload traces from different DBMS. eBPF is a Linux kernel feature that has been gaining attention in both academia and industry promising low-overhead instrumentation capabilities. Furthermore, it is decoupled from the actual DBMS technology and thus widely applicable. Consequently, this leads to the following research questions:

- **RQ1** Can DBMS be instrumented (using eBPF) in order to trace occurring workload?
- **RQ2** How big is the impact of such an (eBPF) instrumentation on the overall performance?
- **RQ3** How does the eBPF impact compare to native DBMS tracing?

While answering these questions our contribution is an analysis of eBPF measurement overhead for two distinct workloads. These workloads are executed on recent releases of MySQL and

PostgreSQL. We further compare those results to their respective native tracing capabilities.

The remainder of this paper is structured as follows. We introduce important aspects of eBPF in Section 2. Subsequently, we describe the implementation of our eBPF based tracing tool in Section 3. This is followed by an evaluation of the results by presenting the measurement scenarios, environment and results in Section 4. Section 5 then puts our findings in contrast to those found in related research. Finally, we conclude the paper.

2 BACKGROUND

The origins of eBPF lie in the initial implementation Berkeley Packet Filter (BPF) [16]. The idea behind this technology was the execution of arbitrary package filter expressions which are passed to the kernel in order to be interpreted there. This enabled users to filter packets without transporting every packet from the kernel to user space and back. The initial BPF was implemented as a minimal and very limited Virtual Machine (VM).

eBPF extends this by following up on the concept of VM and significantly improving its capabilities. Beyond the mere executing of functions on package events, it is now possible to observe and manipulate further sources like Performance Monitoring Counters (PMCs), tracepoints, kernel and user functions. These event sources are not necessarily part of eBPF, but eBPF tooling enables approachable exploitation of those. The typical lifecycle of an eBPF program is visualised in Figure 1 as presented by Gregg [8].

2.1 The eBPF Lifecycle

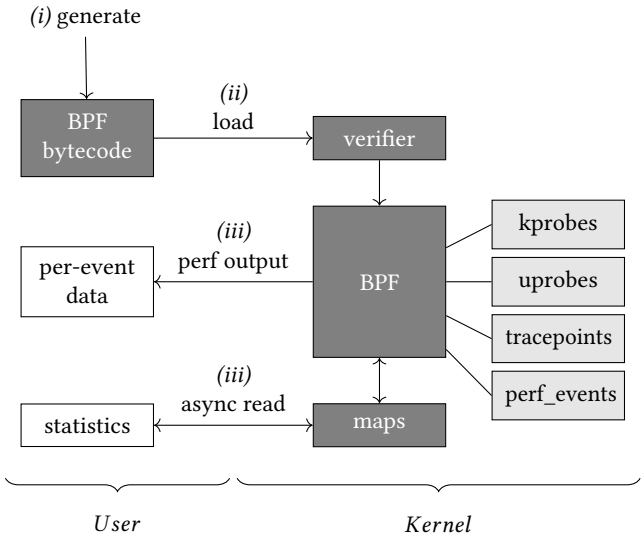

Figure 1: eBPF internals and typical lifecycle according to [8]

The lifecycle of an eBPF program usually begins with *(i)* bytecode for the BPF kernel VM. There is an increasing set of tools for generating such bytecode. These include for instance bcc[1], libbpf[2], and aya[3]. Upon generation, the bytecode is *(ii)* loaded into the kernel for a verification step before it is passed to the actual BPF VM. Finally, any resulting data needs to be transferred back from kernel to user space. While multiple eBPF-based tools only report metrics or findings upon program exit, results can also be transferred to user space at any time, e.g., via *(iii)* `perf_output` and *(iii)* `async read`.

2.2 Linux Profiling Subsystem

The Linux Kernel itself offers vast performance instrumentation within both the kernel and the user space. These instrumentations can be retrieved system-wide, as well as per process and it is thus possible to target specific workloads and applications. Tools for reading such instrumentation are manifold. Yet, they all leverage the same underlying instrumentation by using the Linux profiling subsystem.

The profiling subsystem of Linux is called `perf_events`. It is also known as Performance Counters for Linux (PCL), Linux Performance Events (LPE) and PMC. It supports a multitude of events to be worked with. These events are issued by several mechanisms, particularly: *(a)* counter, *(b)* tracepoints, *(c)* kprobes, *(d)* uprobes and *(e)* User Statically-Defined Tracing (USDT) [9]. They can be divided into the two distinct categories static and dynamic. Static instrumentation happens before compile time and cannot be changed without recompilation. It can be applied to both the kernel and user-space applications. Dynamic instrumentation, however, offers the possibility to attach probes at runtime to arbitrary function calls. The previously mentioned instrumentation points are briefly described in the following paragraphs, accompanied with an overview table in Table 1:

a. (Hardware) Counters represent counts of CPU specific events instrumented by the kernel. Whenever the CPU issues such an event, the counter is incremented. This typically occurs in hardware and is considered to be very lightweight. The available static counters are highly dependant on the CPU manufacturer, as well as its production line and generation. Those are also available as tracepoints.

b. Tracepoints are static kernel instrumentations [23]. They are defined and implemented by the kernel developers and issue an event once a specific call occurs. These signals can be hooked into and thus traced. The events are well documented and range from power management over networking aspects towards filesystem and virtualisation.

c. kprobes relate to tracepoints, bit significantly differ in terms of dynamic instrumentation [11]. Compared to tracepoints they allow dynamically hooking into any arbitrary kernel function call. This significantly increases tracing possibilities. However, the API is rather unstable since it depends on kernel function names. These are not guaranteed to be stable across releases.

d. uprobes are very similar to kprobes regarding their possibilities and downsides. These dynamic probes allow hooking into any arbitrary user space application function. Depending on available debug symbols or details of the application to trace, they are a very

[1] https://github.com/iovisor/bcc
[2] https://github.com/libbpf/libbpf
[3] https://github.com/aya-rs/aya

Table 1: Linux performance instrumentation overview

	static	dynamic	userspace	kernelspace
counter	✓	✗	✗	✓
tracepoint	✓	✗	✗	✓
kprobe	✗	✓	✗	✓
uprobe	✗	✓	✓	✗
USDT	✓	✗	✓	✗

powerful tool to add instrumentation to software in post, without altering or even having access to its source code [6].

(e). USDT relate to uprobes like tracepoints relate to kprobes. Directly compared to uprobes these represent static pre-defined tracepoints within applications. This implies, that they need to be included within the application's source code. There are libraries and bindings for many programming languages and frameworks available to conveniently integrate them into a codebase. To make use of them, they usually need be enabled at compile time.

3 EXPERIMENT DESIGN

In this section, we discuss the design of our experiments. These are supposed to evaluate the capabilities of eBPF tracing for capturing DBMS workloads. Besides, the overhead of such tracing shall be compared to native tracing capabilities of the respective DBMS. In the first subjection, we introduce basic challenges we face with our goals. Then, we briefly sketch the implementation of the tracer tool we built for the sake of this experiment. Doing so, we also discuss essential technical details and highlight the benefits, but also limitations. This is followed by a description of the actual experiment conduction. We present a workflow and the involved entities that take part in the experiments. We further elaborate on the reasons why we chose to implement it that way.

3.1 Approach and Implementation

Probe Selection. As described in Section 2 both kinds of user space probes (uprobes and USDT) allow observing an application at runtime. Amongst others, they allow capturing function invocations including parameters and execution time. Depending of the availability and implementation either static USDT probes or dynamic uprobes may be beneficial to achieve the users intent. In our case, we clearly focus on uprobes. This is due to the fact that we want to be as independent of the DBMS developers, source code availability and pre-existing static instrumentation as possible. Furthermore, MySQL has stopped supporting USDT probes which additionally consolidated this decision.

Generally speaking, using uprobes mode requires attaching a `uprobe` for the query start and an `uretprobe` for the query end function that is responsible for handling incoming queries. These probes are attached using the libbcc library. When a function call is detected, the start timestamp and its arguments can be analysed. When the end of the function gets detected, the execution time of the function can be calculated. Finally, together with their timestamp and duration, the queries can be logged to a file.

More specifically, in order to satisfy our requirements, the functions traced with uprobes need to be the functions responsible for SQL query dispatching. MySQL and PostgreSQL both implement such functions, conveniently with the full database query available as arguments. This can be extended to support additional DBMS by finding an appropriate user function to attach a uprobe to.

Baseline Tool. For our evaluation we extend the already existing tool `dbslower`[4]. Originally, dbslower was built to detect slow database queries to find possible bottlenecks within the workload. In its present release state, the tool is capable of tracing specific queries on MySQL and PostgreSQL relational databases. When use the term ANTtrail in the remainder of this paper to refer to "dbslower" including our own extensions.

Tool Extension. As the tool was initially limited to only show the first 256 chars of the query, the char array that the queries are stored in was extended and to support this change the struct containing the data is now stored in a BPF_PERCPU_ARRAY instead of a BPF_HASH. Furthermore, we added support for PostgreSQL uprobes. We added the capability to enable the tracing and logging of any query to a file in order to replay it later. Moreover, we have removed any kind of filters to make sure to also trace faster queries.

Support for further DBMS can be realised by adding further DBMS specific uprobes. The support for tracing PostgreSQL queries with uprobes was added this way.

Limitations. MySQL and PostgreSQL queries that create queries internally like triggers or recursive SQL, are currently not supported due their internal function invocation structure being different. For these queries, only the submitted query is traced. The internally created queries can be traced by attaching an additional uprobe and uretprobe to the user function that is responsible for them, but they cannot be simply replayed, as they depend on the initial query, e.g. by referencing OLD.value in the case of triggers.

In contrast, queries submitted with the Node.js, Python and JDBC drivers, as well as with the MySQL and PostgreSQL shell, are traced correctly. The overhead of the tool could potentially be reduced, e.g. by filtering empty queries as early as possible in the C program. As the filter that does this for MySQL also filters queries within transactions it was removed and empty queries are currently filtered before they are printed.

3.2 Experiment workflow

In any scenario, a single cloud-hosted virtual machine is used to run the workload generator and a further cloud-hosted virtual machine is used to operate a single-node DBMS instance with default configuration. In all cases the widely-used Yahoo! Cloud Serving Benchmark [2] as workload generator is used for issuing the workload. Every experiment is conducted as a simple workflow. Figure 2 illustrates this workflow in its actual implementation including every involved component.

The workload generator takes a specific *(i)* workload configuration as input. This *(ii)* workload is executed against the database. While the workload is running, the database tracer is executed in

[4] https://github.com/iovisor/bcc/blob/v0.26.0/tools/dbslower.py

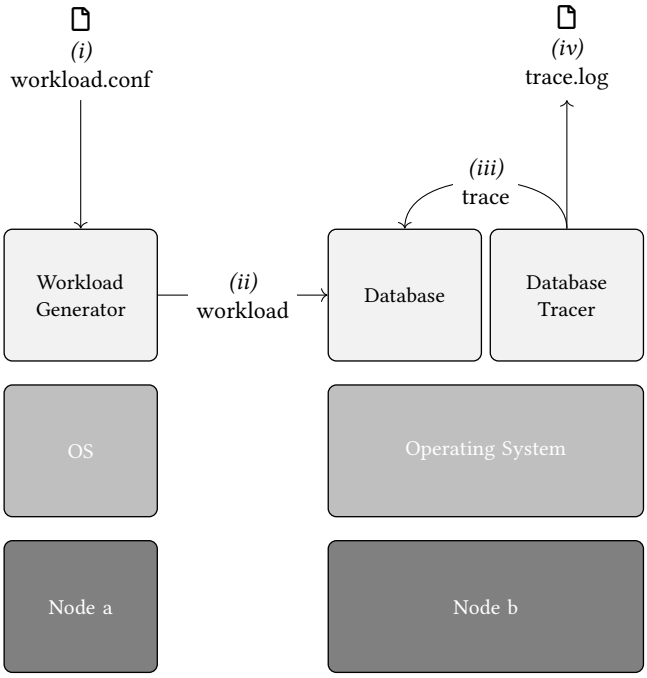

Figure 2: Database tracing architecture

parallel to *(iii)* trace the current workload. While doing so, it *(iv)* logs the trace to a file.

In order to reliably enact this workflow, the benchmark process is executed by benchANT[5] [22], a Benchmarking-as-a-Service platform that builds upon the Mowgli framework [21]. This framework enables the creation of declarative workflows that yield reproducible experimentation results. The benchANT platform takes care of holistically managing the experiment from resource provisioning and benchmark execution to data processing and visualisation.

4 EVALUATION

This section describes the experiment setup to measure the performance overhead of the database tracing for multiple scenarios. First, we introduce the five tracing scenarios, second we provide the technical details for the experiments, and finally, we conclude with discussing the obtained results.

4.1 Tracing Scenarios

We evaluate the performance overhead for four different tracing scenarios against the baseline DBMS performance with the goal to identify the performance overhead using DBMS-specific tracing capabilities versus eBPF based tracing. For the eBPF tracing we analyse three different tracing scenarios to iteratively analyse the performance overhead with increasing processing steps. We assume, that each step increases the performance impact. In particular, our evaluation uses the following tracing scenarios:

Baseline measures the DBMS performance without any tracing in place.

DBMS-Native measures the performance with native DBMS specific tracing capabilities enabled. For MySQL the native logging is enabled by activating the General Query Log[6], for PostgreSQL by setting `log_statement = 'all'`

EBPF-Active detects the queries with ANTtrail, but does not perform any any further processing.

EBPF-Process processes the queries using ANTtrail, but does not persist the captured traces.

EBPF-Persist processes the queries using ANTtrail and persists the queries into a text file.

The implementation for the three different eBPF scenarios using ANTtrail is publicly available[7].

Table 2: Evaluation Environment

	PostgresSQL	MySQL	YCSB
cloud	AWS EC2		
region	eu-central-1		
instance	m5.large	m5.large	c5.4xlarge
storage type	GP2		
OS	Ubuntu 20.04		
version	13.9	8.0.30	0.17.0

4.2 Evaluation Environment

All experiments are conducted through the benchANT platform to ensure a deterministic and reproducible evaluation process together with comprehensive result data [22]. Table 2 provides an overview of the technical details for running the used versions of PostgreSQL, MySQL and YCSB. In depth details can be found in the resulting data sets[8].

Since the benchmarks are executed in the cloud, a certain volatility in the cloud resource performance needs to be taken into account. In earlier work, we could show that three repetitions of DBMS performance benchmarks provide sufficiently stable results if the cloud environment is sufficiently stable [10]. Further, Scheuner has evaluated that AWS EC2 IaaS services do have the required stable cloud resource performance [20]. Therefore, we execute each scenario three times on that cloud platform and report the average together with the standard deviation for each reported performance metric.

We evaluate the performance overhead for the defined tracing scenario under two workloads generated by the YCSB as described in Table 3. For each of the runs, we measure both throughput and latency in intervals of 10 seconds. Any further computations for e.g. median and mean are based on these roundabout 540 data points per configuration (3 runs à 30 minutes with 10 second intervals).

[5]https://benchant.com

[6]https://dev.mysql.com/doc/refman/8.0/en/query-log.html

[7]https://github.com/benchANT/dbms-tracing-overhead/tree/main/code

[8]https://github.com/benchANT/dbms-tracing-overhead/tree/main/results/ycsb

Table 3: Workloads

	read-heavy	write-heavy
YCSB instances	1	
threads	50	
inital data size	10 GB	
write proportion	0.1	0.9
read proportion	0.9	0.1
runtime	30 minutes	

4.3 Results

We have visualised every possible manifestation resulting from the combination of workload type as described in Table 3 and DBMS type as described in Table 2 as a boxplot in Figure 3. Each subplot presents the DBMS throughput as operations per second on its y-axis, whereas each measurement scenario as described in Section 4.1 is represented on the respective x-axis. These further share their colour encoding among all initially mentioned manifestations.

Furthermore, these visualisations are presented in tabular form as seen Table 4 and Table 5. Each table is separated in two overarching columns along their workload type. These are further subdivided into specific metrics aligned to all scenarios. We present both the median and the mean throughput to make sure that these are not too distant from each other. To quickly grasp the performance degradation, we additionally calculate the performance loss induced by each tracing method in percent. In order to highlight the degradation even more, we have colour coded the median throughput loss to quickly spot the worst impact.

4.3.1 DBMS-native Tracing. Focusing on only the degradation induced by DBMS native tracing, we can see that this highly depends on both the workload type and the database implementation. For MySQL, the impact on read-heavy workloads (14.2%) is way higher, than the impact on a write heavy workload (8.6%). One has to consider though, that the throughput of read-heavy workload is roughly three times as high as in the write-heavy case. Interestingly though, it is the other way around for PostgreSQL and generally speaking much worse. Here, the degradation for a read-heavy workload (21.6%) is much lower that for a write-heavy workload (63.5%). The latter is exceptionally high and a convincing reason to not enable its tracing in production. Considering these experiments, we can clearly see that general statements regarding native tracing performance impact cannot be easily made and that any extrapolations may hardly be possible. In order to be certain about the native tracing impact, measurements are necessary since it is highly dependant on the workload and the DBMS in use.

4.3.2 eBPF Tracing. The most interesting following comparison is the impact of an eBPF based measurement in contrast to the baseline throughput, but also to the DBMS native one. As we can see there is always an impact on the baseline performance. It ranges from 3.6% to 16.5%. Notably though, the impact is always less or equal than the impact induced by DBMS native tracing. Especially the huge performance impact in the write-heavy PostgreSQL scenario (63.5%) can be reduced to (12.6%). While still being a noticeable impact this is a significant improvement.

4.3.3 eBPF Overhead. As described in Section 4.1, we performed measurements for an increasing amount of work performed by eBPF. We assumed, that the performance impact is much lower when only triggering on query events (EBPF-Active) and should increase upon processing these events (EBPF-Processing) and finally stronger increase upon persisting (EBPF-Persist) those. Yet, the performance impact of the EBPF-Active scenario was either higher or similar to the EBPF-Persist scenario. We can not coherently argue, why this is so. The eBPF buffer polling mechanism can for instance have an impact here, though, we cannot prove that yet. Other causes are possible as well. A detailed investigation of this aspect is ongoing work.

To summarise, we can show, that eBPF based profiling to derive database traces can yield better performance compared to their native counterpart. The performance degradation is either similar or less. However, it is still not negligible. We expect that this overhead can be further reduced as has been shown by other authors who identified a high potential for optimisation and best practices for these kind of tools [14].

4.4 Evaluation Summary

In this section, we answer the research questions from the introduction based on the results of the evaluation.

RQ1. For the case of PostgreSQL and MySQL an instrumentation with eBPF using uprobes is possible. There is no technical constraint that hinders following the same approach for other DBMS be they relational or NoSQL. Yet, all uprobe instrumentation is specific for a distinct release of a DBMS. Hence, supporting new DBMS or new versions of PostgreSQL and MySQL requires additional effort. In ongoing work we evaluate further approaches for eBPF-based tracing in order to lower the limitations of our current approach.

RQ2. Surprising for us the impact generated with eBPF-based workload tracing is not stable across different workloads and we see a wide range of different overheads. Less surprising, the overhead also depends on the DBMS technology to be traced.

RQ3. In all cases under investigation, the eBPF-based approach is able to compete with the DBMS-native tracing and in two cases it clearly outperforms the DBMS-native approach. Yet, measuring the overhead also yields some results as doing more work in the eBPF handling not necessarily induces more overhead.

5 RELATED WORK

Since eBPF makes leveraging of uprobes and USDT probes approachable, fellow researchers investigated its possibilities for user space application tracing. Their potential due imposing comparatively low overhead has been clear since more than a decade as determined by Keniston et al. [12]. A survey conducted by Gebai et al. extends those findings for most more recent implementations [7].

The overhead of an arbitrary eBPF application however, cannot be generalised and highly depends on the manifold use cases. More specifically, Levin states, that an often overlooked fact is that the overheads of metrics collection is a function of the type, number, and instrumentation for the collected metrics. Nevertheless he built the eBPF based observability tool "ViperProbe" with defined critical metrics for monitoring compute systems. He finds the performance

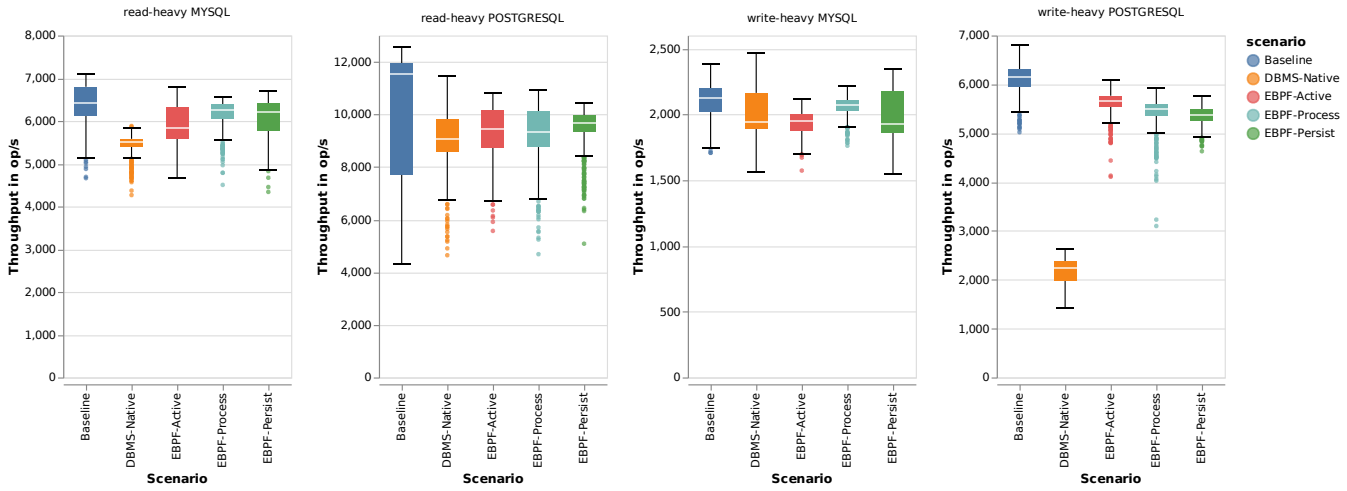

Figure 3: Throughput for each combination of workload and database type for each distinct measurement scenario

Table 4: Degradation for tracing a workload on MYSQL

type	read-heavy				write-heavy			
variable	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput
scenario								
Baseline	0.0	6432.00	6384.03	0.0	0.0	2127.40	2102.87	0.0
DBMS-Native	14.2	5518.80	5456.49	14.5	8.6	1945.50	1999.43	4.9
EBPF-Persist	3.6	6200.65	6070.62	4.9	9.4	1928.00	1988.55	5.4
EBPF-Process	2.5	6268.65	6162.64	3.5	2.7	2070.45	2059.28	2.1
EBPF-Active	9.2	5839.10	5921.38	7.2	8.5	1946.15	1936.57	7.9

Table 5: Degradation for tracing a workload on POSTGRESQL

type	read-heavy				write-heavy			
variable	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput
scenario								
Baseline	0.0	11546.90	10076.10	0.0	0.0	6149.80	6109.23	0.0
DBMS-Native	21.6	9057.00	9080.74	9.9	63.5	2245.75	2180.93	64.3
EBPF-Persist	16.4	9656.45	9405.72	6.7	12.6	5374.05	5347.53	12.5
EBPF-Process	19.2	9331.80	9209.57	8.6	10.7	5490.00	5413.67	11.4
EBPF-Active	18.1	9460.30	9297.67	7.7	7.9	5666.45	5620.40	8.0

impact when collecting all configured metrics to be between 10-15% [15]. Within the same domain but with some improvements, Amaral et al. developed "MicroLens". They determined a similar performance impact with it being 18% across multiple nodes and 9% on a single host [1]. Furthermore Krahn et al. also experienced a similar impact with their tool named "TEEMon" for Redis[9], MongoDB[10] and Nginx[11] [13].

Apart from the more generic metric collection for the purpose of monitoring, literature also provides insights for tracing specific applications. Nisbet et al. determined the impact of various profiling tools for a java application. He found that this impact with 3.6% is rather low for the bcc-java implementation that leverages eBPF [17].

In earlier work, we sketched how workload traces from production systems can be clustered, modified, mixed and scaled to perform load testing and what-if analysis [4].

[9]https://redis.io/
[10]https://www.mongodb.com/
[11]https://www.nginx.com/

6 CONCLUSION

We can conclude, that eBPF has the potential to improve the already existing tracing capabilities DBMS have to offer. Our experiments show, that such an approach either has a similar or a lower impact on database performance. We can further see, that this impact highly depends on the applied workload and is very specific to the database implementation itself. In some cases the performance impact is so low, it may be usable in production. Judging that however, is the responsibility of database operators, though we can provide the means and a method to file that decision. Traces and insights gained hereby can help those operators to improve overall performance, troubleshoot issues or generally observe workload.

From an academic point of view, the generation of real world database traces can be very valuable. Only few datasets exist, and the availability of a low performance impact and non-intrusive tracing tool might help to improve this situation.

Nevertheless, chances are, that the performance penalty induced by eBPF tracing might improve in the future. One the one hand, the implementation of our tooling is certainly not perfect. Furthermore, fellow researchers have shown development methods, best practices and optimisation, that potentially lower performance bottlenecks caused by eBPF.

Within this work we focused on the usage of user space tracing with uprobes and USDT. Kernel instrumentation with tracepoints and kprobes might be another promising direction to pursue and possibly gain performance benefits. Especially analysing network packets and possibly offloading this to network interface cards supporting this may be of interest.

These optimisations, as well as the implementation for further DBMS including NoSQL based ones will be our next steps. Moreover an investigation on alternative instrumentation points may be valuable here.

Acknowledgements

Research leading to this work has received funding from the German Federal Ministry for Economic Affairs and Climate Action under grant 03EFPBW217, BaaS.

REFERENCES

[1] Marcelo Amaral, Tatsuhiro Chiba, Scott Trent, Takeshi Yoshimura, and Sunyanan Choochotkaew. 2022. MicroLens: A Performance Analysis Framework for Microservices Using Hidden Metrics With BPF. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022 IEEE 15th International Conference on Cloud Computing (CLOUD). (July 2022), 230–240. DOI: 10.1109/CLOUD55607.2022.00043.

[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (SoCC '10). Association for Computing Machinery, Indianapolis, Indiana, USA, 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152.

[3] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. 2021. DIAMetrics: Benchmarking Query Engines at Scale. *ACM SIGMOD Record*, 50, 1, (June 17, 2021), 24–31. DOI: 10.1145/3471485.3471492.

[4] Jörg Domaschka, Mark Leznik, Daniel Seybold, Simon Eismann, Johannes Grohmann, and Samuel Kounev. 2021. Buzzy: Towards Realistic DBMS Benchmarking via Tailored, Representative, Synthetic Workloads: Vision Paper. In *Companion of the ACM/SPEC International Conference on Performance Engineering* (ICPE '21). Association for Computing Machinery, New York, NY, USA, (Apr. 19, 2021), 175–178. ISBN: 978-1-4503-8331-8. DOI: 10.1145/3447545.3451175.

[5] Jörg Domaschka, Simon Volpert, and Daniel Seybold. 2020. Hathi: An MCDM-based Approach to Capacity Planning for Cloud-hosted DBMS. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). (Dec. 2020), 143–154. DOI: 10.1109/UCC48980.2020.00033.

[6] Srikar Dronamraju. 2022. Uprobe-tracer: Uprobe-based Event Tracing — The Linux Kernel documentation. Accessed: 2023-02-27. Retrieved Feb. 27, 2023 from https://docs.kernel.org/trace/uprobetracer.html.

[7] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *ACM Computing Surveys*, 51, 2, (Mar. 12, 2018), 26:1–26:33. DOI: 10.1145/3158644.

[8] Brendan Gregg. 2022. Linux eBPF Tracing Tools. Retrieved Sept. 9, 2022 from https://www.brendangregg.com/ebpf.html.

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

[10] Johannes Grohmann, Daniel Seybold, Simon Eismann, Mark Leznik, Samuel Kounev, and Jörg Domaschka. 2020. Baloo: measuring and modeling the performance configurations of distributed DBMS. In *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020*. IEEE, 1–8. DOI: 10.1109/MASCOTS50786.2020.9285960.

[11] Masami Hiramatsu. [n. d.] Kprobe-based Event Tracing — The Linux Kernel documentation. Accessed: 2023-02-27. (). Retrieved Feb. 27, 2023 from https://docs.kernel.org/trace/kprobetrace.html.

[12] Jim Keniston, Ananth Mavinakayanahalli, Vara Prasad, and Prasanna Panchamukhi. 2007. Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps. In *Proceedings of the 2007 Linux Symposium*.

[13] Robert Krahn, Donald Dragoti, Franz Gregor, Do Le Quoc, Valerio Schiavoni, Pascal Felber, Clenimar Souza, Andrey Brito, and Christof Fetzer. 2020. TEEMon: A continuous performance monitoring framework for TEEs. In *Proceedings of the 21st International Middleware Conference*. Middleware '20: 21st International Middleware Conference. ACM, Delft Netherlands, (Dec. 7, 2020), 178–192. ISBN: 978-1-4503-8153-6. DOI: 10.1145/3423211.3425677.

[14] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. 2022. Verified programs can party: optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22: Seventeenth European Conference on Computer Systems. ACM, Rennes France, (Mar. 28, 2022), 283–299. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519562.

[15] Joshua Levin. 2020. ViperProbe: Using eBPF Metrics to Improve Microservice Observability. Honors Thesis. (2020). https://cs.brown.edu/research/pubs/theses/ugrad/2020/levin.joshua.pdf.

[16] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (USENIX'93). USENIX Association, USA, (Jan. 25, 1993), 2.

[17] Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján. 2019. Profiling and Tracing Support for Java Applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19: Tenth ACM/SPEC International Conference on Performance Engineering. ACM, Mumbai India, (Apr. 4, 2019), 119–126. ISBN: 978-1-4503-6239-9. DOI: 10.1145/3297663.3309677.

[18] Rasha Osman and William J. Knottenbelt. 2012. Database system performance evaluation models: A survey. *Performance Evaluation*, 69, 10, (Oct. 1, 2012), 471–493. DOI: 10.1016/j.peva.2012.05.006.

[19] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. 2017. On the State of NoSQL Benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17: ACM/SPEC International Conference on Performance Engineering. ACM, L'Aquila Italy, (Apr. 18, 2017), 107–112. ISBN: 978-1-4503-4899-7. DOI: 10.1145/3053600.3053622.

[20] Joel Scheuner and Philipp Leitner. 2018. Estimating cloud application performance based on micro-benchmark profiling. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 90–97. DOI: 10.1109/CLOUD.2018.00019.

[21] Daniel Seybold. 2021. *An Automation-Based Approach for Reproducible Evaluations of Distributed DBMS on Elastic Infrastructures*. Ph.D. Dissertation. Universität Ulm, (May 14, 2021). ISBN: 9781757899956. DOI: 10.18725/OPARU-37368.

[22] Daniel Seybold and Jörg Domaschka. 2021. Benchmarking-as-a-service for cloud-hosted dbms. In *Proceedings of the 22nd International Middleware Conference: Demos and Posters* (Middleware '21). Association for Computing Machinery, Virtual Event, Canada, 12–13. ISBN: 9781450391542. DOI: 10.1145/3491086.3492473.

[23] Theodore Ts'o. 2022. Event Tracing — The Linux Kernel documentation. Accessed: 2023-02-27. Retrieved Feb. 27, 2023 from https://docs.kernel.org/trace/events.html.