Kmon: An In-kernel Transparent Monitoring System for Microservice Systems with eBPF

Tianjun Weng, Wanqi Yang, Guangba Yu, Pengfei Chen*
School of Computer Science and Engineering
Sun Yat-sen University
{wengtj3,yangwq23,yugb5}@mail2.sysu.edu.cn
chenpf7@mail.sysu.edu.cn

Jieqi Cui, Chuanfu Zhang
School of Systems Science and Engineering
Sun Yat-sen University
cuijq5@mail2.sysu.edu.cn, zhangchf9@mail.sysu.edu.cn

***Abstract*—Currently, the architecture of software systems is shifting from "monolith" to "microservice" which is an important enabling technology of cloud native systems. Since the advantages of microservice in agility, efficiency, and scaling, it has become the most popular architecture in the industry. However, as the increase of microservice complexity and scale, it becomes challenging to monitor such a large number of microservices. Traditional monitoring techniques such as end-to-end tracing cannot well fit microservice environment, because they need code instrumentation with great effort. Moreover, they cannot explore the fine-grained internal states of microservice instances. To tackle this problem, we propose Kmon, which is an In-kernel transparent monitoring system for microservice systems with extended Berkeley Packet Filter (eBPF). Kmon can provide multiple kinds of run-time information of micrservices such as latency, topology, performance metrics with a low overhead.**

***Index Terms*—Microservice, Cloud computing, Monitoring, eBPF, Kubernetes**

I. Introduction

Attracted by the characteristics of high flexibility and fast delivery, microservice is widely used in many modern companies now, such as Google and Microsoft. To monitor and manage microservice systems, system operators need to collect performance indicators that reflect system states and persistently store those indicators in a database.

Some conventional monitoring tools (e.g., cAdvisor[1]) can help system operators to collect and aggregate indicators. However, most of them only focus on basic metrics of resource usage(e.g., CPU utilization). Another type of monitoring tools like Istio[2] can monitor more indicators like requests latency at L7 network layer, but they need to change the underlying infrastructure. To gain diversified metrics for existing systems without modification, we propose Kmon, an in-kernel transparent monitoring system for microservice systems.

Kmon can capture conventional metrics more accurately. Furthermore, it can collect fine-grained in-kernel performance indicators(e.g., the number of system calls). In-kernel indicators are useful because they can reflect hidden problems in deep (e.g., *livelock* shown in §II or limplock [14]). It is almost impossible to find out these deeper problems with current monitoring tools since they cannot obtain in-kernel events. Therefore, it is important to collect in-kernel indicators for microservice observability.

Although previous tools like strace[3] can capture in-kernel indicators, they sacrifice the performance of the system and need to aggregate manually for distributed systems. Other methods (e.g., OpenTelemetry[4]) need to change the source code of the user's program, which introduces additional complexity. Kmon chooses to use eBPF, a component of the Linux kernel to collect metrics of programs without instrumentation. It reduces the difficulty of using eBPF and enables eBPF to sense changes in application layer, especially for the changes of microservice. Users can simply use it by write configuration and deploy Kmon to each host. Then Kmon can automatically collect metrics and store them into databases. The configuration specifies metrics users want to gain and server instances in microservice that need to be monitored, which does not require modifying source or provide internal logic information of programs.

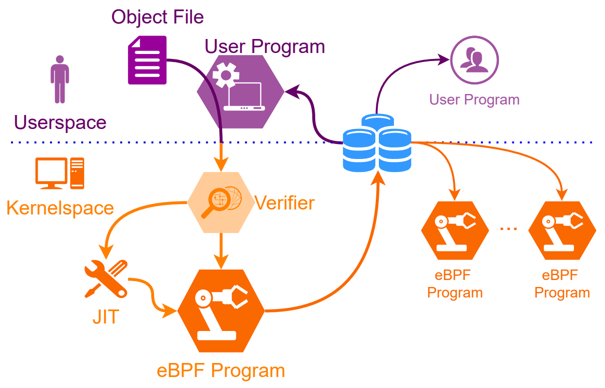


Fig. 1. The Basic Architecture of eBPF.

As part of the Linux kernel, eBPF provides a possibility to extend the Linux kernel. System operators can use it to capture in-kernel indicators at a low cost. However, write eBPF program directly is hard. Fig. 1 shows the basic architecture and procedure of eBPF. First, a developer should write eBPF code in C, then use llvm to compile it into an object file, which will be loaded into kernelspace; Second, the developer should write a program to load eBPF, this program usually running on userspace. Before the object file is loaded, the kernel runs a verifier to ensure it cannot damage the kernel. Then JIT transfers the object file from byte code to machine instructions. Finally, eBPF program and user program communicate with each other by eBPF maps, the user program can extract kernel runtime information from maps, or ask eBPF program to change the behavior of the kernel or itself by sending signals to maps.

[1]cAdvisor: github.com/google/cadvisor
[2]Istio: istio.io/
[3]strace: strace.io/
[4]OpenTelemetry: opentelemetry.io/

Writing eBPF code is hard because it needs to understand the source code of the Linux kernel and observe the rules of the verifier. The development environment is also complex. In recent years, some eBPF tools (e.g., BCC[5]) have emerged to reduce the difficulty of developing eBPF programs. But it has two challenges to monitoring microservice with eBPF as follows.

- The eBPF program collects many non-numerical indicators (e.g., stack address shown in §III-C) that cannot be utilized directly. Thus, we must translate those unreadable non-numerical data to human-readable indicators.
- The eBPF program captures the in-kernel indicators that are corresponding with the microservice containers' PID. Therefore, eBPF cannot sense the changes of microservices semantically. We must give eBPF the ability to adapt to microservice and other high-level changes(e.g., user configuration changes).

For these challenges, Chang et al. [2] used eBPF to collect indicators for microservice profiling, but they have not considered the changes of microservices. Shiraishi et al. [3] implemented dynamic sensors to microservices through eBPF, but they need to create or delete eBPF programs when adjusting monitor items, which causes extra overload. Viperprobe [1] is a microservices collection framework, which focuses only on numerical metrics and ignores the non-numerical indicators.

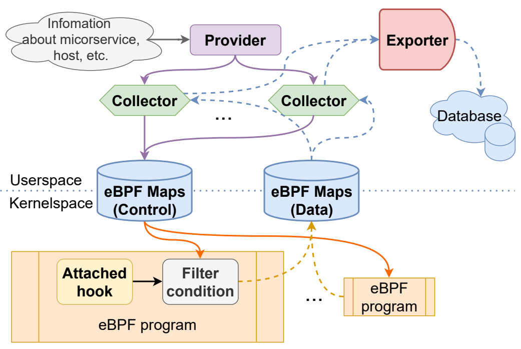


Fig. 2. The Architecture of a Kmon.

Kmon use visualize tools like flamegraph and heatmap to increase readability for visualization high-density and non-numerical indicators. Kmon consists of Provider, Collector, Exporter, and eBPF. Provider collects the high-level information and sends them to Collector for further indicators transformation. Collector controls the behavior of eBPF programs through the eBPF map, transforms and aggregates indicators from eBPF programs. Exporter forms the indicators and stores them into databases. Experiments show that Kmon can collect fine-grained indicators at a low cost.

The contributions of this paper can be summarized as follows:

- It not only enables eBPF to sense the changes of microservice, but also enables it to sense other high-level information like changes of user-defined configuration.
- It is an attempt at storing, handling and visualizing non-numeric in-kernel metrics for the distribution system diagnosis.
- It unifies multiple in-kernel metrics in one system with the same data format, reduces the workload of metrics collecting and translating.

[5]BCC: github.com/iovisor/bcc

II. Motivation

Firstly, capturing all types of indicators with a unified data structure can reduce the cost of indicator aggregation. There are various types of monitoring tools now. There is no doubt they work well in some specific fields. However, it is difficult to obtain a holistic view of a system with only one monitor tool. Moreover, the output format of each tool is quite different. Therefore, it is hard to combine all indicators in one view.

Secondly, a monitoring system that collects fine-grained indicators can help people find out more problems. Here is an example of *livelock*: if a process gets stuck in a *livelock*, it still consumes CPU resources while does nothing useful for its tasks. It cannot reflect a *livelock* problem only with CPU usage. *Livelock* occurs between processes that will increase the number of context switches. If we can obtain the number of context switching or the execution stack of each context switch, we are more likely to recognize such a problem.

Thirdly, a system that does not need instrumentation can reduce the cost of development and deployment. Some metrics in microservice, such as the latency of TCP messages, are hard to collect. They can be captured via code instrument, or by deploying software that results in an additional performance loss of users' programs like Istio. Kmon is designed without any code instrument. Thus it has little impact on the new applications and runtime applications.

III. Approach

A. System Architecture Overview

Kmon monitors microservices at three levels, which share the same architecture when implemented. That is because all indicators can be similarly collected by eBPF. Before describing methods of monitoring at each level, the shared architecture is introduced first. Fig. 2 illustrates the architecture of Kmon.

- **Provider**: It collects high-level information which are sent to Collectors for updating monitoring strategy.
- **Collector(s)**: They load eBPF program and communicate with them by eBPF maps. Each Collectors are responsible for a type of indicator.
- **eBPF Maps**: eBPF maps are the key/value-based storage structure in the kernel. In Kmon, they are used to send control messages from Collector to eBPF programs (called control maps), and collect metrics from eBPF program to Collector (called data maps).
- **eBPF Program(s)**: They run and hook some specific events and functions in kernel to collects and store data into data maps.
- **Exporter**: It receives the metrics from Collectors. Then it aggregates those metrics and sends them to the database.

B. Provider

Provider (Fig. 3) is responsible for collecting high-level information. The initial eBPF is used at the kernel level, so it is hard to capture high-level information, such as microservice information and user-defined configuration. However, as Kmon is designed for microservice, it is necessary to provide this information for eBPF. Without PID provided to identify the program in a container, eBPF does not know which program it needs to monitor; With notice of configuration changing by a user, Kmon node can change its monitoring strategy without restarting the node or the eBPF program.

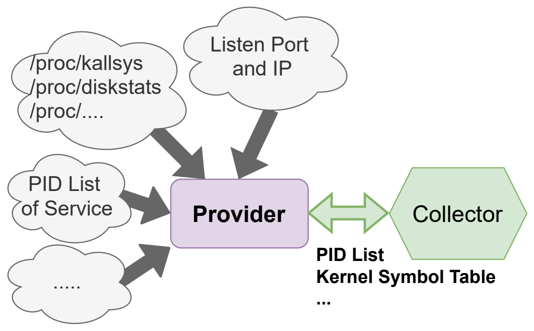


Fig. 3. The high-level information probed by Provider for data parsing

Provider also needs to provide necessary information for Collector to parse data. For example, eBPF can collect kernel stack in hook trigger, and stores addresses in form of unsigned long integer, which is unreadable for humans, and hard to analyze for algorithms without a kernel symbol table. So if an indicator contains kernel stacks, Provider should collect the kernel symbol table for Collector to convert addresses to symbol names.

C. Collector

The Collector (Fig. 4) interacts with eBPF programs. It loads the eBPF program into kernel and communicates with eBPF programs via eBPF maps. Communications contain two parts: to control behaviors of eBPF programs and to receive data collected by eBPF programs.

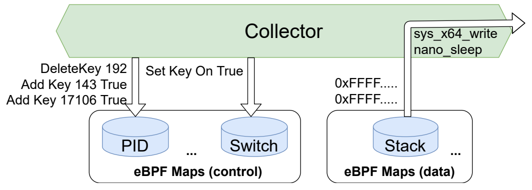


Fig. 4. The basic workflow of Collector

For the former, Collectors analyze high-level changes from the Provide. For example, if services are deleted or created, Collectors should immediately notify eBPF programs of changing their monitoring PID list. If a user changes configurations that are relative to some Collectors, these Collectors need to reflect these changes to control maps. For the latter, Collectors receive data from data maps. This raw data (bitmap, bytes data, address, etc.) are usually human unreadable, so Collectors should convert them to an appropriate form. These indicators are sent to Exporter and stored in databases.

D. eBPF Maps and eBPF Program

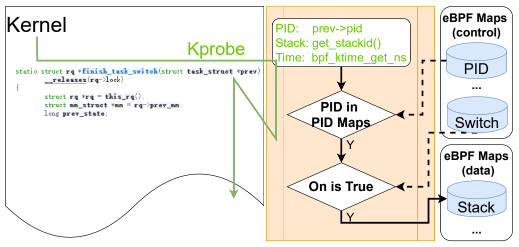

```
static struct rq *finish_task_switch(struct task_struct *prev)
        __releases(rq->lock)
{
        struct rq *rq = this_rq();
        struct mm_struct *mm = rq->prev_mm;
        long prev_state;
```

Fig. 5. The control maps and data maps in Kmon

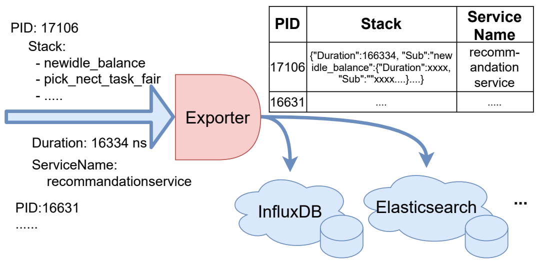
To make Kmon a flexible system, eBPF programs(Fig. 5) need to adjust their behaviors when microservice or configuration changes.

One way for eBPF programs to adjust their behavior is to recompile and reload them, which are used in BCC. It needs to install the heavy compiling environment (e.g., LLVM, Clang, etc.) in production environment. As microservice changes frequently, recompiling of eBPF program for each change causes much overload.

Kmon adjusts in another way, namely storing the control message into eBPF maps. eBPF programs check maps for choosing executive branch. The usage of these eBPF maps is different from eBPF maps that store collected data. Therefore, we name the former control maps and data maps respectively. In this way, eBPF can be precompiled before deployed in the system, avoiding the heavy compiling environment and the recompilation overload in run-time.

E. Exporter

Exporter (Fig. 6) sends indicators collected by Collector to databases. Exporter supports different databases by transfer data forms to meet the requirements of databases.



PID	Stack	Service Name
17106	{"Duration":166334, "Sub":"newidle_balance":{"Duration":xxxx, "Sub":""xxxx....}....}	recomm-andation service
16631	....	.....

Fig. 6. The primary workflow of Exporter

IV. Implementation of In-kernel monitoring

Kmon can collect various types of indicators via its architecture. There are too many types to decide which should be focused on when monitoring, so a hierarchy of these indicators for locating the problem is helpful.

Kmon classifies indicators into three categories(Fig. 7). The first is TCP request level. TCP requests is common in microservice and many abnormal detecting algorithms for microservice depend on it. In this level, Kmon captures indicators of each TCP connection such as quaternion and latency.

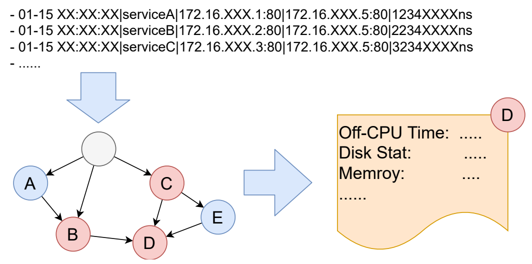

```
- 01-15 XX:XX:XX|serviceA|172.16.XXX.1:80|172.16.XXX.5:80|1234XXXXns
- 01-15 XX:XX:XX|serviceB|172.16.XXX.2:80|172.16.XXX.5:80|2234XXXXns
- 01-15 XX:XX:XX|serviceC|172.16.XXX.3:80|172.16.XXX.5:80|3234XXXXns
- ......
```

Fig. 7. Three levels of performance indicators.

The second level is the topology level. Dynamic service graph are formed by TCP request data. Nodes in the graph represent service instances, and the link between nodes reflects flows of recent network requests. It aggregates information of each host from the whole microservice system. Data at this level is intuitive for humans and algorithms to identify the relation between services. According to the latency information and topology, it is convenient for system operators to estimate if there is an abnormal node in the graph.

The third level contains other fine-grained indicators for further investigation. It includes metrics relative to CPU, memory, block I/O, etc. The source code of eBPF programs for Kmon refer to BCC examples, with some changes to match Kmon architecture.

A. TCP Request monitoring

Here is an example of collecting indicators of TCP request. Tbl. I shows what should be provided by Provider and Tbl. II shows what kernel function should eBPF program to monitor. The request latency is calculated by subtracting the first received message timestamp from the last sending message timestamp(Fig. 8), so eBPF should hook relative function call to capture the time.

TABLE I
HIGH-LEVEL INFORMATION

PID	Identify receiver of requests.
Service name	Convert PID to service name for readability.
IP listen list	Identify server side for persistent connection.

TABLE II
HOOKED KERNEL FUNCTIONS

tcp_sendmsg ***tcp_cleanup_rbuf***	Capture the time when TCP message sent and received, for latency timing.
security_socket_accept ***tcp_close***	Capture the time when TCP connection accept and close, for identify server side for short connections.

Kmon only collects requests sent to servers, but it is difficult to distinguish the direction after a socket being accepted. Kmon uses the hook about "accept" and "close" operation to decide short connection direction, which assumes that the server-side executes the "accept" system call.

However, it is not suitable for a persistent connection. If a connection was made before Kmon has been deployed, Kmon cannot sense it. Hence Provider should collect the listen port infomation. Kmon assumes that if the local host is on the server-side, its IP and port should be in the local host's IP listening list.

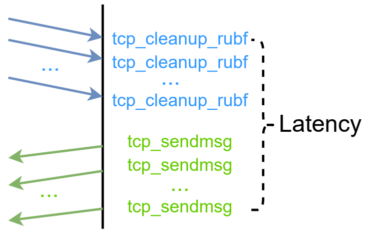


Fig. 8. TCP latency calculation.

B. Service Topology Monitoring

Service topology can be constructed by the data from TCP requests. It searches all requests that occur in a period, draws node and edge according to its source, and destination. It is a dynamic topology as different topologies are constructed in different periods, which is suitable for microservice for its frequent changes. The data from TCP requests to construct topology is sufficient, so no more Collectors are needed.

C. Fine-grained Performance Indicators

Other indicators share the same architecture of Kmon, but with different eBPF programs and Collectors. Tbl. III and Tbl. IV show the information about some implemented indicators that the Provider and the eBPF program collect.

Network: we choose tcp_drop as an example. It can capture each packet dropped by kernel, with a stack for each dropping. It can debug high-rate of drops. There are many indicators about network like tcp_connect and tcp_accept. We introduce tcp_drop because others are similar and we have only implemented tcp_drop now.

CPU: Off-CPU time[6] can capture what and when a process is blocked. It is useful to analyze details of the process's or kernel's behaviors[7]. Because capturing this indicator has high cost as tracing scheduler is called frequently[8], Kmon just stores data for those who let processes go to "sleep" state instead of each scheduling (While it is still high-cost.). This indicator needs to store stacks of each schedule, which need high memory. Inspired by Flame Graph, Kmon translate and stores stack in trees. Each function call is a node of the tree with its total durations. Collector constructs trees for each interval (for example, 5s) to trace changes of a process.

Block I/O: Many metrics can be captured from three hooks in Fig. III, such as I/O type and throughput. These metrics are mainly gain from pointer of "request" struct, which is the parameter of hooked function. It can also capture I/O latency by subtracting timestamps of "blk_account_io_start" from timestamps of "blk_account_io_done" It is useful to optimize I/O performance, for example, placing service instances that always write at the same time to different machines.

TABLE III
HIGH-LEVEL INFORMATION

Configuration	For changing Kmon's behaviour.
PID	Identify and filter program of monitoring.
Service name	Convert PID to service name for readability.
Kernel symbol table	Convert stack address to symbol name.
Disk name	Convert major and minor number of disk to name.

TABLE IV
HOOKED KERNEL FUNCTION

Type	***Hook***	***describtion***
Network	tcp_drop	Capture TCP packets or segments that were dropped by the kernel.
Block I/O	blk_account_io_start blk_mq_start_request blk_account_io_done	Capture indicator about block I/O, such as read-write type, throughput, latacy, I/O.
CPU	activate_task deactivate_task	Capture the on and off CPU counts, time and its stack for specific program.

We also implement a generic system call hook to count the number of specific kernel functions called(e.g. tcp_connect, tcp_drop, write, and read) specified by user. Some kernel functions like system calls can reflect the type of program, for example, the number of execution "read" or "write" may be high for an I/O frequent service.

V. EXPERIMENT SETTING

Our experiment uses four Linux virtual machines with Linux kernel v5.4 to make up a Kubernetes cluster. we use "Hipster shop"[9], a cloud-native microservices application demo from google as a benchmark. It compromises 10-tier microservice on which users can browse items, add them to the cart, and purchase them. We change its load generator to k6, which can get more metrics after load having been generated. The load of k6 is set to be 100 users in all experiments.

Kmon node is deployed to each host via Kubernetes with a new ClusterRole. ClusterRole is necessary to get permission for Kmon node in a container to execute eBPF and collect information of Kubernetes (e.g., namespace, pod name, service name, nodes, etc.). The resource usage of service and Kmon node is recorded by a metric server.

[6]www.brendangregg.com/blog/2016-01-20/ebpf-offcpu-flame-graph.html
[7]www.brendangregg.com/blog/2017-08-08/linux-load-averages.html
[8]github.com/iovisor/bcc/blob/master/tools/offcputime_example.txt
[9]Hipstershop: github.com/GoogleCloudPlatform/microservices-demo

VI. Experiment and Result

A. Performance metrics

To estimate the resource usage of an indicator of a Kmon node, the first experiment runs Kmon node program manually without the help of Kubernetes(K8s), and with the Exporter disabled. It avoids the network overload of storage, so we can focus on the indicator monitoring part.

The host which runs the Kmon node contains 3 of 10 services ("paymentservice", "emailservice", and "frontend"). In the first experiment, Tools "top" in Linux is used to measure the resource usage of Kmon nodes.

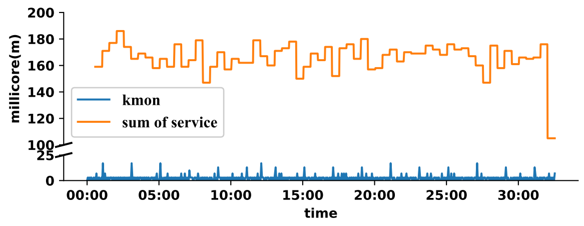


Fig. 9. CPU usage for micrioserive and Kmon node in single host.

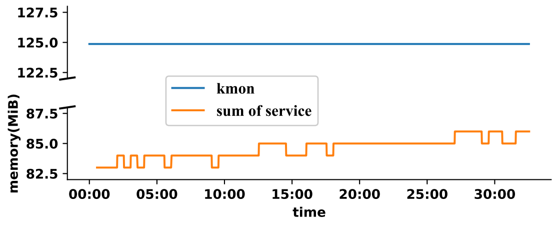


Fig. 10. Memory usage for micrioserive and Kmon node in a single host.

Fig. 9 and Fig. 10 show the CPU and memory usage of the Kmon node. The CPU usage (exclude eBPF program) is negligible while memory usage is relatively high. It is possible to have more optimization for memory usage. We will do this in the future.

After measuring the resource usage of one Kmon node, it is necessary to estimate Kmon's performance in the whole microservice. In this part, Kmon is deployed in each host, with 4 kinds of indicators collected, which is Network (TCP request latency, drop message), Block I/O state (throughput, count, type, stack), Function counter (system call "write" and "read"), and Off-CPU time (stack, duration). Tbl. V shows some of the results of exported TCP request data, which contains timestamp, latency source, and destination.

The average resource usage for Kmon node in 30 minutes is 9.55% in CPU usage) and 616.11MiB in Memory usage. Tbl. VI shows the influence in response time.

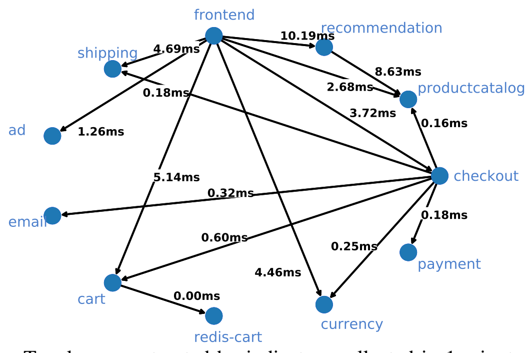


Fig. 11. Topology constructed by indicators collected in 1 minutes.

Fig. 11 shows the topology visualization of requests by network data collected from this part of the experiment. For a more fine-grained indicator in a pod, like "recommendation service", Fig. 13 shows the state at the beginning of K6 starts to generate load. The program in the pod sleeps longer and more frequently. Combine with Fig. 12, which shows the duration when the program is off-CPU in each type of stack. Sleep (do_nano_sleep), synchronize (futex_wait_queue_me) and epoll (poll_schedule_...) may cause its increasing.

VII. Related Work

A. Intrusive Monitor Framework

The intrusive monitor frameworks need to change source code or binary file of the user's programs, which introduces addtional overhead. Pythia [5] focuses on *where, what, and when* to instrumentation in a distributed application in an automatic way. Seer [6] and MicroRank [9] instrument trace API to microservice applications for request information to match patterns of services. The work [10] increases observability by inserting API hooks into source code for distributed system diagnosis. Compared with the previous works, users can get the detailed monitor information with the help of Komn without any instrumentation.

B. Non-intrusive Monitor Framework

Contemporary in-kernel monitoring tools such as ftrace[10] and sysdig [4] can gain fine-grained indicators on a single host, but they are hard to collect and aggregate indicators from all nodes in microservice environment. Microscope [11] captures the network connection information of microsercice systems to monitor the changes of service dependency graphs. Microscaler [12], [13] uses the Service Mesh to monitor metrics of microservice systems. However, Microscope and Microscaler cannot get the system-lever metrics.

Chang et.al., [2] use the Bayesian model to analyze data collecting by eBPF collected by *valtrace*. The study in [7] uses the random forest model to analyses network-related metrics from eBPF in virtual machines. Both studies are focus on algorithms rather than fitting dynamic environment. Compared with the previous works, Kmon is more suitable for a dynamical microservice environment and can gain fine-grained indicators efficiently.

VIII. Conclusion

To create a transparent monitoring system with fine-grained indicators, we introduce Kmon, an eBPF based system. It transparently captures various types of indicators and organizes them in three levels, which is convenient for system operators and algorithms. Experiments show it has low CPU usage and little influence on service response time.

In the future, we aim to find a better way to capture TCP connection information on fewer assumptions. The current assumption of TCP requests is not suitable for some specific type of service like Message Queue. The memory usage of Kmon also needs to be reduced. Two directions are considered. One is to use library libbpf instead of libbcc, which uses less memory in run-time with portability, The other is to find a better representation of indicators to compress their size.

Data Availability

The source code of Kmon system can be found in zenodo: https://zenodo.org/record/4596298.

[10]ftrace: www.kernel.org/doc/Documentation/trace/ftrace.txt

TABLE V
RECORD OF TCP MESSAGE SENDING

time	*service*	*latency (ns)*	*myIP:port*	*peerIP:port*
Jan 15, 2021 @ 21:56:13.206	emailservice	375170	172.20.1.132:8080	172.20.2.134:51614
Jan 15, 2021 @ 21:56:13.200	shippingservice	676425	172.20.1.129:50051	172.20.2.134:50330
Jan 15, 2021 @ 21:56:13.199	cartservice	594035	172.20.1.132:7070	172.20.2.134:53604

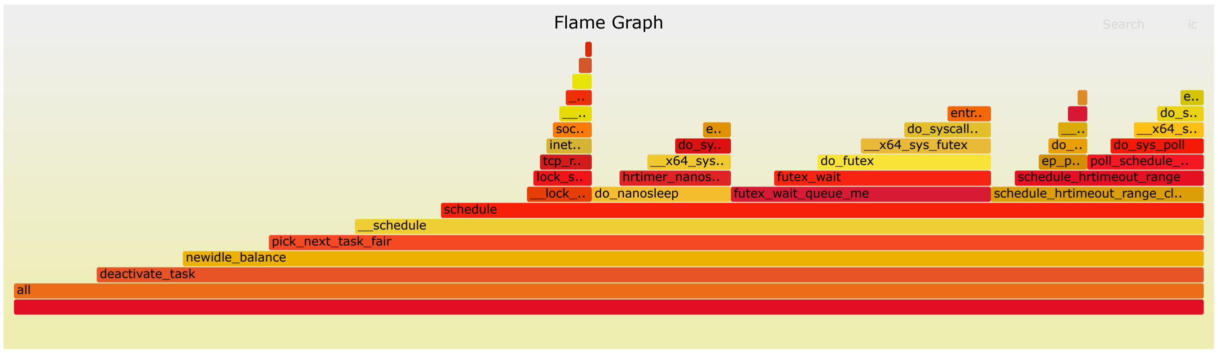


Fig. 12. The Flame graph of off-CPU time of one process. A flame graph can reflect how a program spends its time. Each bar is at random warm colors and ordered alphabetically, the top one shows the duration of off-CPU time, and beneath is its ancestry. The Y-axis of the bar shows stack depth. In this picture, most of the off-CPU time is obtained from function futex (synchronize) and poll (network).

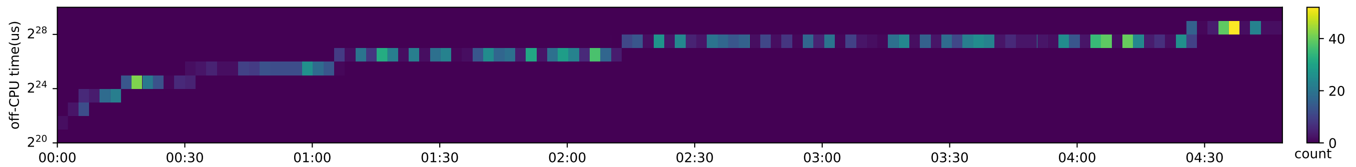


Fig. 13. Heatmap of off-cpu time (only with sleep state) of service "recommendation service" in 4 minutes at the start of load generator running.We can easily notice the increase of off-CPU time.

TABLE VI
RESPONSE INFLUENCE FOR MICROSERVICE IN 30 MINUTES

	With Kmon Node	*Without Kmon Node*
Response-time(Avg)	361.97ms	348.97ms
Response-time(P95)	790.09ms	784.09ms

ACKNOWLEDGMENT

The work was supported by the Research Development Plan of Key Areas in Guangdong Province (2020B010165002), the Natural Science Foundation of China (U1811462, 61802448), the Basic and Applied Basic Research of Guangzhou (202002030328), and the Natural Science Foundation of Guangdong Province (2019A1515012229). The corresponding author is Pengfei Chen.

REFERENCES

[1] Levin J. "ViperProbe: Using eBPF Metrics to Improve Microservice Observability" Technical Report, 2020.
[2] Chang, H., Kodialam, M., Lakshman, T. V., & Mukherjee, S. (2019). Microservice Fingerprinting and Classification using Machine Learning. In 2019 IEEE 27th International Conference on Network Protocols (ICNP),pp. 1–11.
[3] Shiraishi, T. , Noro, M. , Kondo, R. , Takano, Y. , & Oguchi, N. . (2020). Real-time Monitoring System for Container Networks in the Era of Microservices. 2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS).
[4] Gianluca Borello. 2015. System and Application Monitoring and Troubleshooting with Sysdig. USENIX Association, Washington, D.C
[5] Ates, E., Sturmann, L., Toslali, M., Krieger, O., Megginson, R., Coskun, A. K., & Sambasivan, R. R. (2019). An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In Proceedings of the ACM Symposium on Cloud Computing,pp. 165–170.
[6] Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., & Delimitrou, C. (2019). Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,pp. 19–33.
[7] Wallschläger, M., Gulenko, A., Schmidt, F., Acker, A., & Kao, O. (2018). Anomaly Detection for Black Box Services in Edge Clouds Using Packet Size Distribution. In 2018 IEEE 7th International Conference on Cloud Networking (CloudNet) pp. 1-6.
[8] Bertrone, M., Miano, S., Risso, F., & Tumolo, M. (2018). Accelerating Linux Security with eBPF iptables. In Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos,pp. 108–110.
[9] Yu, G., Chen, P., Chen, H. (2021). MicroRank:End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. In Proceedings of the Web Conference 2021, doi:10.1145/3442381.3449905
[10] Huang, P., Guo, C., Lorch, J. R., Zhou, L., Dang, Y. (2018). Capturing and enhancing in situ system observability for failure detection. In OSDI'18 Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (pp. 1–16).
[11] Lin, J., Chen, P., Zheng, Z. (2018). Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments. In International Conference on Service-Oriented Computing (pp. 3–20).
[12] Yu, G., Chen, P., Zheng, Z. (2019). Microscaler: Automatic Scaling for Microservices with an Online Learning Approach. In 2019 IEEE International Conference on Web Services (ICWS) (pp. 68–75).
[13] Yu, G., Chen, P., Zheng, Z.(2020) Microscaler: Cost-effective Scaling for Microservice Applications in the Cloud with an Online Learning Approach. In IEEE Transactions on Cloud Computing, doi: 10.1109/TCC.2020.2985352.
[14] Do, T., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Gunawi, H. S. (2013). Limplock: understanding the impact of limpware on scale-out cloud systems. In Proceedings of the 4th annual Symposium on Cloud Computing (p. 14).