A Black-box Monitoring Approach to Measure Microservices Runtime Performance

ROLANDO BRONDOLIN and MARCO D. SANTAMBROGIO, Politecnico di Milano, ITA

Microservices changed cloud computing by moving the applications' complexity from one monolithic executable to thousands of network interactions between small components. Given the increasing deployment sizes, the architectural exploitation challenges, and the impact on data-centers' power consumption, we need to efficiently track this complexity. Within this article, we propose a black-box monitoring approach to track microservices at scale, focusing on architectural metrics, power consumption, application performance, and network performance. The proposed approach is transparent w.r.t. the monitored applications, generates less overhead w.r.t. black-box approaches available in the state-of-the-art, and provides fine-grain accurate metrics.

CCS Concepts: • **Networks → Cloud computing**; • **Software and its engineering → Monitors**; • **Hardware → Power and energy**;

Additional Key Words and Phrases: Microservices, power attribution, performance monitoring, network performance monitoring, docker, kubernetes, cloud computing

ACM Reference format:
Rolando Brondolin and Marco D. Santambrogio. 2020. A Black-box Monitoring Approach to Measure Microservices Runtime Performance. *ACM Trans. Archit. Code Optim.* 17, 4, Article 34 (November 2020), 26 pages.
https://doi.org/10.1145/3418899

1 INTRODUCTION

The past ten years of evolution of cloud computing saw the rise of many technologies able to ease the development, deployment, and maintenance of complex applications at scale. One of the main elements that fostered this evolution was the concept of *microservice*. The microservice architectural style [18] is an approach that allows building an application as a suite of small and independent services communicating with each other through lightweight networking mechanisms. Such microservices can be developed with different programming languages, leveraging different storage technologies, and following different business needs. From a technology perspective, Docker [16] is the main contributor to this evolution, along with Kubernetes [8]. Docker containers leverage kernel-level virtualization implemented with Linux *namespaces* and *cGroups* to limit visibility and provide resource isolation between microservices. On top of Docker, Kubernetes orchestrates resources across a cluster of machines, managing the

Authors' address: R. Brondolin and M. D. Santambrogio, Politecnico di Milano, Piazza Leonardo da Vinci 32, Milano, 20133, ITA; emails: {rolando.brondolin, marco.santambrogio}@polimi.it.

1544-3566/2020/11-ART34
https://doi.org/10.1145/3418899

containers' life-cycle and scaling them depending on performance needs. Kubernetes implements microservices as *pods*, where a pod is a group of containers providing a single functionality.

The microservice architectural style combined with the proper technological tools allowed to move away from the standard monolithic pattern. However, the complexity that was previously hidden inside the monolithic application moved as well, abruptly increasing the pressure on the network layer. With the growing complexity of deployments, it is becoming fundamental to monitor and measure the performance of microservices at scale to assess their functionality and to detect performance issues as soon as they appear in the system. Such issues may arise in several layers of the cluster's stack and can affect network performance, system performance, and resource usage. Moreover, as discussed in Reference [19], microservices interact with CPU architectures differently w.r.t. monolithic workloads, and these kinds of interactions can be captured by monitoring performance counters. If we leverage data about performance counters, system performance, network performance, and power consumption, it is then possible to spot bottlenecks, improve performance, and increase energy efficiency. For these reasons, monitoring should take into account all these aspects, focusing on pure software performance metrics as well as on architectural ones.

Within this context, several works [21, 24] addressed the monitoring issue for Linux servers and in general for physical hosts and virtual machines. However, they lack the ability to drill down at the container level. Many commercial tools [12, 31] and open-source tools [35] overcame this limitation by instrumenting application code. Finally, References [15, 38, 40] retrieve data using a black-box approach, however, they only provide metrics about the high-level performance of the applications, neglecting the low-level metrics needed to monitor how the workloads are exploiting the underlying architecture.

In this article, we present a novel black-box approach to monitor microservices at scale in the context of Docker containers managed by a Kubernetes orchestrator. The goal is to build a unified view of a microservice-based application from low-level architectural metrics to network performance. For this reason, the contributions of this article are the following:

(1) The design of a monitoring agent based on extended Berkeley Packet Filter (eBPF) [4, 27] that collects data about application performance (e.g., CPU usage, execution time), low-level performance (e.g., cycles, Instruction Retired (IR), cache references, cache misses, and power consumption), and network activity (e.g., number of requests, bytes sent and received, average latency, and from 50th to 99th percentile latency) for each Docker container, Kubernetes pod, and physical host;
(2) The design of a metrics collection system that retrieves metrics from a set of monitoring agents, groups them depending on container, pod, service, and host, and provides a graph view and analysis of the monitored Kubernetes cluster that encompasses performance, power consumption, and network activity of each microservice.

Within this article, we demonstrate how the proposed methodology generates less overhead on the monitored benchmarks w.r.t. similar approaches in the state-of-the-art. At the same time, we are able to guarantee a reasonable accuracy over the measurements the monitoring system performs. Finally, we show a study of performance and power consumption of one application of the *DeathStarBench* [19] benchmark suite, discussing the possible tradeoffs between optimizing power consumption and maintaining the performance of the applications.

The rest of this article is organized as follows: Section 2 analyzes the related works in the field with a focus on power, performance, and network traffic monitoring in microservices and cloud-based environments. Section 3 describes the proposed solution to monitor microservices behavior from the low-level eBPF module to the remote metrics collection system. Section 4 shows the experimental results we obtained with the proposed monitoring approach in terms of overhead and precision of the collected metrics. Section 4 derives also insights on how the benchmarks behaved during the experimental campaign, thanks to our monitoring service. Finally, Section 5 draws the conclusion and derives future works.

2 RELATED WORK

According to the Cloud Native Computing Foundation (CNCF) Cloud Computing Landscape [11], observability of cloud-native applications can be divided into three main areas: *monitoring*, *logging*, and *tracing*. Within this article, we focus on monitoring microservices power and performance. *Monitoring* cloud applications allows capturing the runtime behavior of the system, enabling a thorough analysis of the collected data and highlighting performance issues that can lead to a poor end-user experience. Here, we provide a brief view of the state-of-the-art for power monitoring as well as application performance and network performance monitoring.

2.1 Power Monitoring

Power consumption can be measured at different levels of the data-center stack, from the power grid level arriving to the thread level in a single host. In the past few years, some works focused on fine-grain power monitoring, from Virtual Machines (VMs) to containers, to threads. One of the first works in this area is represented by Bellosa et al. [5]. Within this work, the authors found a first correlation between the power consumption of the server and some hardware performance counter measuring data at the level of the whole machine. At the Virtual Machine (VM) level, a notable work is represented by *XeMPower* [17]. *XeMPower* allows to precisely attribute power consumption to each VM in a given host with low overhead on the system. The tool monitors both Running Average Power Limit (RAPL) and various performance counters that are traced at the context switch of the *Xen* vCPUs. *Power Containers* [37] works at the per-request level but does not take into account Hyper Thread (HT). This last aspect was taken into account by *HaPPy* [41] with a proportional power attribution at the thread level based on Running Average Power Limit (RAPL) [36] and *cycles*.

Moving to containers and microservices, Piraghaj et al. [34] proposed a framework and algorithm for energy-efficient container consolidation in cloud data centers. *DockerCap* [1] proposes an Observe Decide Act (ODA) loop to cap power consumption while assigning resources to guarantee a Service Level Agreement (SLA). Power monitoring in this context is done through RAPL at the socket level. Brondolin et al. [7], instead, build on top of *HaPPy* [41] and perform power attribution at the container level. The proposed work improves accuracy over References [7, 41] while providing analysis also on performance monitoring.

2.2 Application Performance Monitoring and Network Performance Monitoring

To monitor hardware performance counters, two main tools are available: *Linux perf* [13] and *PAPI* [29]. Linux Perf is a performance tool that can measure performance counters, tracepoints, hardware, and software events, and it can provide statistical profiling of the entire system performance. *PAPI*, instead, is a library to collect performance counters from within the monitored application. *Linux perf* is the standard tool for performance counter profiling, and, for this reason, we decided to use *perf arrays* within the eBPF code to retrieve them.

As for application performance monitoring, several works exist in the state-of-the-art, both at the academic level as well as at the commercial level. For what concerns commercial and open source tools, here, we describe briefly the most relevant. *Sysdig* [38] is a performance monitoring tool that collects data about processes, containers, and Kubernetes clusters without instrumenting the user code. *Sysdig* introduces a kernel driver with several buffers that collect and analyze all the system calls to provide metrics about CPU usage, memory usage, network I/O, and file I/O. Although our tool does not collect memory usage and file I/O, with Sysdig, we are not able to collect low-level performance metrics and latency percentiles. *Weave scope* [40] collects data only about CPU usage and network connections without latency measures with eBPF and builds a network topology graph on top of it. *Datadog* [12] instruments the applications with custom integrations. Network analysis is performed through eBPF and it is based on *Weave scope*, however, it does not account for latency measures but just for bandwidth. Finally, *Prometheus* [35] is an open-source project within the Cloud Native Computing Foundation (CNCF) that provides Application Programming Interfaces (APIs) to instrument the user code, a time-series database, and a visualization interface to show the metrics.

If we consider instead the research works in the field, one of the first works is represented by *Ganglia* [24], which is a distributed monitoring system for clusters and grids typically used in High Performance Computing (HPC). Another interesting work is *Nagios* [21], which is an open-source monitoring tool for grid computing. Both *Ganglia* and *Nagios* focus on host metrics and grid metrics and do not take into account application-related metrics. Moving to microservice-aware research works, Noor et al. [32] designed a monitoring framework that can work in a multi-cloud, multi-virtualization, and multi-microservice setting. The authors retrieve microservice's performance from the *Linux* Operating System (OS), group everything by microservice, and expose the data through a REST interface. Although this work measures data at the microservice level, network performance is not considered. Pina et al. [33] instead focus mainly on monitoring the network performance without user code instrumentation. To avoid instrumentation they introduce infrastructural components that are used by the application such as the Application Programming Interface (API) gateway, the service discovery, and the load balancers. Unfortunately, these components are not mandatory in a microservice environment: Our approach can retrieve the network metrics without instrumenting them. Chang et al. [9] specifically target Kubernetes and collect metrics about CPU usage, Memory usage, and Quality of Service (QOS) violation metrics. The proposed approach reads aggregate resource usage data from the *Linux* Operating System (OS) as Reference [32] and retrieves network performance by stressing the applications with *Apache jMeter.*[1] Although this approach obtains network metrics similar to the one we provide, jMeter is extremely invasive w.r.t. the performance of the applications.

Another class of interesting work focuses on pure network performance monitoring. *Ntopng* [14] is a system for network traffic monitoring and characterization. It is mainly based on *libpcap* and *tcpdump* [39] and provides a *Lua* based scripting engine as well as a web server to observe network traffic and a data exporter. *Ntopng* can work efficiently with 10 GbE networks without performance loss. The work of Deri et al. [15] is a Kubernetes and container-aware runtime security tool that leverages eBPF to gather connections' open and close events. It works both with Transfer Control Protocol (TCP) and **User Datagram Protocol (UDP)** and retrieves data exchange with *Ntopng*. The work of Deri et al. uses different Linux kprobes and tracepoints w.r.t. our work obtaining similar information. Unfortunately, given that the scope of Reference [15] is on network security, no information about accuracy and overhead is provided. The work of Cinque et al. [10] provides a tool for network traffic analysis that accompanies a system for log collection that involves application instrumentation. Its *MetroFunnel* component is based on *pcap* and can pinpoint abnormal service operation with small overhead on the performance of the applications. Finally, *ConMon* [28] monitors functions inside adjacent containers. Passive monitoring uses tools like *tcpdump*, while active monitoring injects network traffic to test the response of the workloads.

[1] https://jmeter.apache.org.

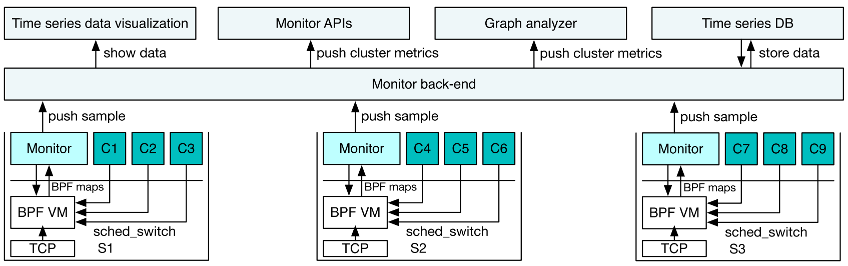


Fig. 1. Overall infrastructure of the proposed monitoring system with kernel level data collection through BPF, user-space monitoring agent that collects metrics and Docker and Kubernetes data, and back-end infrastructure with data visualization, monitor APIs, and graph analyzer.

Among the reviewed state-of-the-art so far, none of the described works was able to combine monitoring data on resource usage, application performance, network performance, and application power consumption to provide a unified view of the application. A work that comes close to it is *Seer* [20]. *Seer* leverages application performance, resource usage, host power consumption, and network tracing data to build a neural network model able to predict Quality of Service (QoS) violations across a cluster of microservices. Our work focuses more on data acquisition rather than performance predictions. Moreover, our work requires no instrumentation of the user code, while *Seer* tracing system requires a hook on the code that performs network operations. Another work in this area is *Rusty* [23], which uses power consumption and performance counters to predict the behavior of low-level metrics of dockerized workloads. We believe that our approach can be used within *Rusty* to also integrate network performance in the prediction framework.

3 MONITORING INFRASTRUCTURE

Within this article, we propose a generalized black-box monitoring approach to measure the performance of cloud-native applications. We currently focus on microservices deployed with Docker containers and managed by a Kubernetes orchestrator, although the proposed methodology can be adapted to other container engines as well as orchestrators. For each container, we collect data about low-level performance (e.g., Instruction Retired (IR), cycles, cache references, cache misses, and power consumption), application performance (e.g., CPU usage and total execution time), and network performance (e.g., number of connections, number of requests, bytes sent and received, average latency, and from 50th to 99th percentile latency). We designed the proposed monitoring tool following three main principles: *transparency*, *performance*, and *accuracy*. *Transparency* means to monitor cloud-native applications without requiring user intervention in terms of configurations and user-code instrumentation. *Performance* means to introduce the least overhead possible on the monitored applications and to use as few resources as possible on the monitored hosts. Finally, *accuracy* means to retrieve metrics in the most precise way without data loss.

Figure 1 shows the main components of the monitoring infrastructure: kernel-level instrumentation, user-space agent, and remote monitoring backend. For each host, we deploy a kernel-level instrumentation layer implemented with eBPF [4, 27]. On top of the kernel-level instrumentation, we run a user-space monitoring agent that periodically collects the metrics and enriches them with information about threads, processes, containers, and Kubernetes pods and services. Once the monitoring sample is ready, the monitoring agent sends it to the remote backend each second. At this point, the backend computes and stores the metrics in a time-series database and builds the connection graph between pods. Finally, a user interface visualizes the metrics.

3.1 Monitoring Agent

Starting from the principles depicted in Section 3, our goal is to build a monitoring system able to observe applications without code instrumentation, without providing significant impact on performance, and with the highest accuracy possible. Unfortunately, to avoid code instrumentation, we need to resort to the OS to retrieve the data that we need. The common way to collect data about the running workloads without instrumenting them is to develop a custom kernel module [38]. However, this is usually a time-consuming and error-prone task. To overcome this limitation, we resorted to eBPF, which is a kernel-level VM that runs inside the Linux kernel. eBPF allows to do Just In Time (JIT) compilation of C code that can get access to different events inside the Linux kernel. The kernel-level VM enforces security measures to prevent system instability: For instance, to load an eBPF program, a user needs to have privileged access to the host system. Moreover, the kernel-level VM accepts only codes that do not have loops, that have at most 4,096 eBPF assembly instructions, and that do not use more than 512 bytes of stack. These measures are adopted to avoid infinite loops and to limit the footprint on performance and memory usage of the eBPF programs. Within this context, to ease the development process, we leveraged the BPF Compiler Collection (BCC) [3] tools to compile, load, and manage eBPF codes at runtime. BPF Compiler Collection (BCC) provides a wrapper to an LLVM compiler that is able to compile C code to eBPF assembly, a set of helper functions in C for the eBPF programs, and a set of helper functions for Python codes. These last helper functions allow to load and remove eBPF programs and perform R/W operations on eBPF data structures accessible both from the Python code as well as from the eBPF code.

The peculiarities of eBPF and BCC allow us to design monitoring tools with the following general approach: (1) extract metrics from data as soon as the data is generated, (2) move metrics only in aggregated form to limit the impact on the overall system, (3) correlate metrics coming from different hosts without coordination among distributed agents. Previous works that leverage custom kernel modules are not able to provide aggregation at the kernel-level due to security and performance issues, and, as such, they encounter performance and accuracy degradation when the monitored system is overloaded.

Starting from the general approach, we designed two different eBPF programs able to collect performance data and network data, respectively. These two programs react to Linux tracepoints and kprobes that are specific to the data the monitoring tool has to collect. Once the eBPF VM receives an event from the Linux kernel, the event is passed to the proper program that processes it and stores the output on a hash map that can be accessed by a user-space agent. All the processing activity on the single event is performed within the eBPF VM and the hash map stores only aggregated results. This allows avoiding to send the raw events from kernel-space to user-space, reducing the user-space agent load that can otherwise result in high overhead and data loss in case of system saturation. At this point, the only goal the user-space agent has is to enrich the aggregated data with context information from Docker and Kubernetes and then send all the data to a remote backend to achieve cluster-level visibility.

3.1.1 Power and Performance Monitoring. Within Intel processors, Hyper Thread (HTs) belonging to the same core share some resources. This means that executing two threads on different cores has a different impact on the system's power consumption than executing the same two threads on two HTs that share the same core. To perform the power attribution, we resorted to the findings of References [7] and [41]. In particular, References [7, 41] show that there is a strong correlation between *cycles* and *RAPL core power consumption* and that there is a ratio (of 1.1 for *Sandy Bridge* and *Ivy Bridge*) that can be used to weight *cycles* to perform a fair attribution of power consumption across threads. We denote this ratio as HT_r and we derive the weighted cycles in Equation (1):

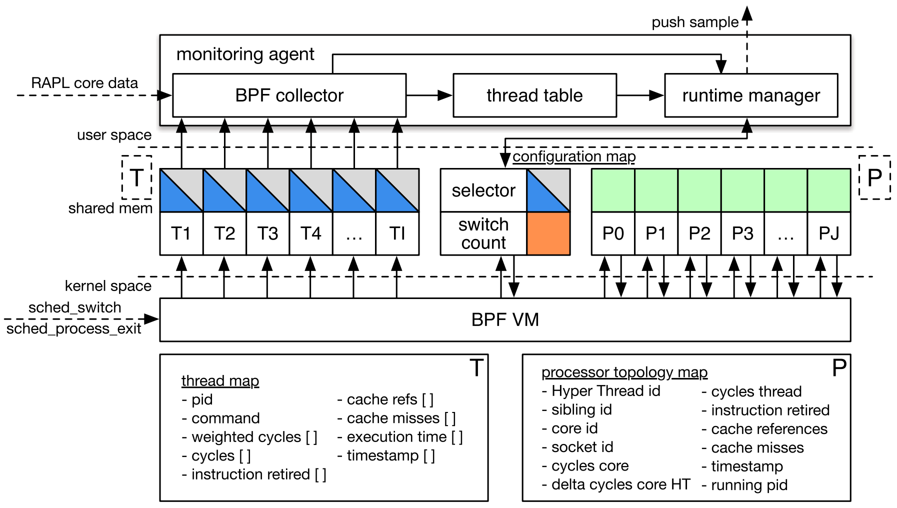


Fig. 2. Power and performance monitoring structure: The BPF code sends metrics to the agent leveraging the thread map and stores partial results within the processor topology map. The user space agent iterates over the thread map to collect the metrics and correlate cycles with RAPL power measurements. Thread metrics are stored inside the thread table ready to be sent to the backend via the runtime manager.

$$Cycles_{W_T}(t,s) = Cycle_{A_i}(s,t) + \frac{HT_r}{2} \cdot Cycle_{O_i}(s,t). \tag{1}$$

The weighted cycles for a given thread i on a given socket s during the observation time t is the sum of two contributions: (1) the cycle measurements $Cycle_{A_T}(s,i)$ when no threads are running in the sibling HT; (2) the cycle measurements $Cycle_{O_T}(s,j)$ when there is a thread running in the sibling HT. This second contribution is scaled by the HT_r ratio divided by 2 to avoid accounting twice for the same cycles measurements on the two sibling threads.

To attribute the power consumption to a thread i for a given observation interval t, it is sufficient to divide the power consumption measured with RAPL by all the weighted cycles of all the threads K that run in a given time interval t. This result can then be multiplied by the weighted cycles of thread i. This operation should be performed for each socket, as shown in Equation (2):

$$P_i(t) = \sum_{s \in S} \left(RAPL_{core}(t,s) \cdot \frac{Cycles_{W_i}(t,s)}{\sum_{k \in K} Cycles_{W_k}(t,s)} \right). \tag{2}$$

To perform power and performance monitoring, we need to track *context switches* to monitor how threads are scheduled and executed on the host processor. Context switch events are exposed with a *tracepoint* by the Linux kernel and with eBPF, we can attach to it to know when a thread starts its execution and when it ends. In this way, we can measure the execution time and performance counters and we can collect data to attribute power consumption. Figure 2 shows the main components of the kernel-level data acquisition for low level and application-related performance metrics. The eBPF virtual machine communicates with the user-space agent with two hash maps: *thread map* and *processor topology map*. The former stores the aggregated metrics for each thread observed in the system, while the latter works as a scratchpad for each HT to keep partial results during the observation. The *configuration map*, instead, stores the amount of context switch observed and the selector field used to switch between two memory areas: One is used by the eBPF code to write metrics, while the other is used by the user-space agent to retrieve the metrics.

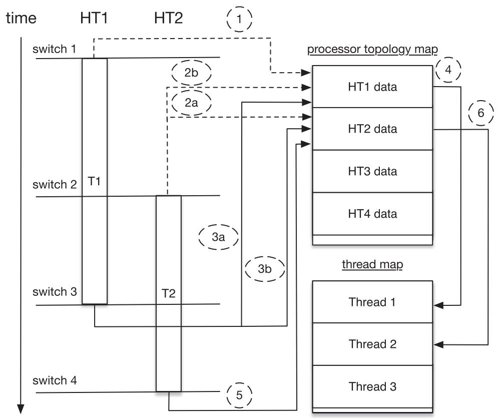


Fig. 3. Hyper Thread aware event flow for power and performance monitoring with events order highlighted. Dotted arrows are for threads that start execution, full arrows for threads that are stopping.

Figure 3 shows the events flow during metric collection for power and performance metrics with two threads. When a thread has to be executed on a given HT (HT 1 in the case of Figure 3), the scheduler performs a context switch that is captured by our monitoring tool. Each context switch provides information about the new thread (e.g., thread ID, command, priority) as well as the old thread. Once the context switch arrives, we measure *cycles*, *IRs*, *cache references*, and *cache misses* on the HT that is going to host the thread using the eBPF *perf maps*. We then store these data along with the timestamp to the *processor topology map* (step 1). Eventually, a new thread is scheduled on HT 2. In the case of Figure 3, we measure the performance counters for HT 2 and we store them in the *processor topology map* (step 2a). To account for the shared execution on the core, we count also the *cycles* of HT 1 and we update this information in the *processor topology map* (step 2b). When the first thread stops the execution, a new context switch happens. We measure again the performance counters of HT 1, we account to the thread the difference between the first measurements and the second ones (step 3a) and we increment the weighted cycles for thread 1 following Equation (1). As we did in step 2b, we account for the shared *cycles* in step 3b. Thread metrics are then stored in the *thread map* (step 4). We repeat the same operations for the second thread when it stops execution (step 5), updating its metrics in the thread map (Step 6). When a thread exits, a *process exit* tracepoint triggers the thread removal from the *thread map* and the cleaning of the *processor topology map* for the HT that was running.

Unfortunately, all these measurements are subject to variability, since context switches depend on how the application is interacting with the system. Applications that heavily exploit network communication like user-facing cloud workloads will be subject to more context switch activity than applications that focus on batch computations such as *High-Performance Computing (HPC)* or *BigData* workloads. To reduce this uncertainty, each second we generate a software event that is captured by the eBPF code. This event allows computing the metrics for each running thread without waiting for a context switch, thus having an updated view of the thread metrics before collecting them in user-space.

3.1.2 Network Monitoring. From an OS perspective, networking activity can be captured and analyzed at different layers of the stack. In particular, the OS handles communication until layer 4, while layer 7 is handled directly by the applications. To achieve *transparency*, we decided to monitor network traffic within the OS at layer 4 (e.g., Transfer Control Protocol TCP) trying also to detect plain text data that can reconstruct layer seven protocols (e.g., HyperText Transfer Protocol

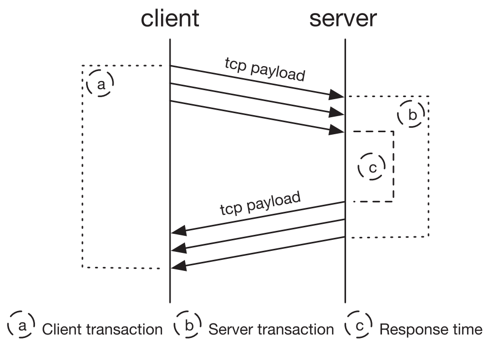


Fig. 4. Network transaction over a TCP connection as seen by a client process and a server process.

(HTTP)). To this aim, we decided to leverage Transfer Control Protocol (TCP) *kprobes* and *tracepoints* to capture the entire TCP payload and to avoid reconstructing the TCP flow. Within this context, we are mainly interested in network performance metrics such as bandwidth and latency. Once we have the network connection and the TCP payload, the effort to measure bandwidth is quite low. On the contrary, measuring latency requires to introduce the concept of network transaction.

As shown in Figure 4, a *network transaction* is a data exchange in an established TCP connection between a client process and a server process where the client initiates the transaction by sending one or more TCP payloads and the server replies by sending one or more TCP payloads. A new network transaction begins (and the previous one ends) when the client process starts again to send one or more TCP payloads after receiving the TCP payloads sent by the server during the previous transaction. Starting from a network transaction, we can define two measurements: the server response time and the transaction Round Trip Time (RTT). The server response time measures the time required by the server to build a response to a given request, while the transaction Round Trip Time (RTT) measures the time from the first TCP payload sent by the client to the last TCP payload received by the client. On the one hand, if we are monitoring a client process, we measure the transaction RTT; on the other hand, if we are monitoring a server process, we measure the server response time.

Figure 5 shows the main components behind the kernel-level data acquisition for network-related metrics. The eBPF code reacts to TCP kprobes such as *tcp_set_state*, *tcp_send_msg*, *tcp_recv_msg* and *tcp_cleanup_rbuf* and supports both IPv4 and IPv6 communications. The kernel calls the *tcp_set_state* function when there is a state change in the TCP socket (e.g., listen, syn sent, established, closing) and we capture it to track the state of each connection in the system. *Tcp_send_msg* is called when the kernel wants to send data over the TCP connection, while *tcp_recv_msg* is called when one or more TCP payloads need to be received. Kprobes and tracepoints can be attached at function invocation as well as at function return, providing, respectively, the input parameters and the return value of the given function. Unfortunately, while *tcp_send_msg* provides socket data, message content, and message size at function invocation, *tcp_recv_msg* provides the socket data and an empty message pointer at function invocation and the amount of data read at function return. Even if we instrument both invocation and return of *tcp_recv_msg*, we will not be able to assign the message size to the connection, as we do not have the socket data at function return. For this reason, we decided to attach a kprobe also at the invocation of *tcp_cleanup_rbuf*, which is a function called mainly by *tcp_recv_msg* at the end of the message retrieval. *Tcp_cleanup_rbuf* provides at function invocation both the socket and the size of the

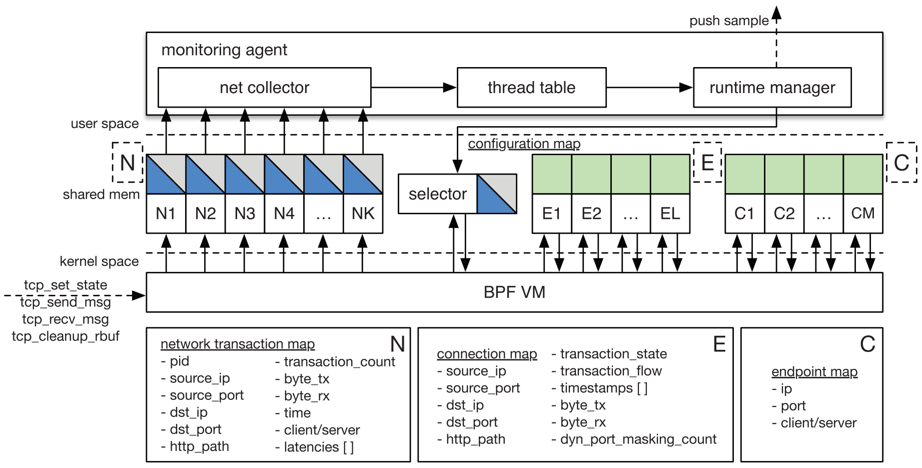


Fig. 5. Network monitoring structure: The BPF code sends metrics to the agent leveraging the network transaction map and stores partial network data in the endpoint map and in the connection map. The user space agent iterates over the network transaction map to collect the metrics and attribute them to each container. Network metrics are stored in the thread table to be sent to the backend via the runtime manager.

data (but not the message pointer), allowing us to reconstruct for a given socket the message read (through the message pointer saved from the *tcp_recv_msg* kprobe) and its size.

We record all the information we collect from the kprobes according to the mechanism of Figure 4 in the *network transaction map* and we leverage some helper maps to update the state of all the connections in the system. In particular, Figure 5 shows two helper maps: the *endpoint map* and the *connection map*. Each time a connection is established, we store the endpoints in the *endpoint map* with information on the endpoint role (i.e., client or server) to attribute the correct timing when capturing data exchange between clients and servers. The *connection map* is used instead as a scratchpad to keep track of the open connections and to keep track of the transaction state and flow for each connection. Within this context, a connection is identified by source IP and port, destination IP and port, and the HyperText Transfer Protocol (HTTP) path that we extract from the message payload in case of plain-text transmission. The *network transaction map*, instead, stores for each connection the data about all the transactions recorded in the last observation interval. In particular, we store the Process ID (PID) of the last thread that interacted with the connection, the number of transactions, the amount of bytes received and transmitted, the average latency, and a sample of the latency measures (i.e., 240 items, configurable). Once a transaction exceeds the size of the latency measures array, a new item is inserted in a uniformly random position using a reservoir sampling technique. Data in the *network transaction map* use the same selector mechanism described in Section 3.1.1.

Given that we use source IP, source port, destination IP, destination port, and HTTP path when available to identify a given connection in our system, we may have a lot of connections between the same two endpoints where the only difference is in the client port. This is particularly true for HTTP, where each request can require a new connection with new parameters unless the client decides to recycle the already open connections. This behavior severely affects our data collection system, as we need to generate a new item in the *network transaction map* for each connection we track for just a few network transactions. To avoid *network transaction map* saturation, we decided to mask the client port in case we track the HTTP protocol and in case the two endpoints exchange few transactions (currently less than 10) during the connection lifetime. This tradeoff still allows to have a detailed view of the network activity in terms of granularity of connections, allows to better use the latency sampling technique, and reduces the pressure on the *network transaction map*.

3.1.3 User-space Data Collection. The user-space agent is the first data aggregation layer in our monitoring infrastructure. Its main role is to manage the kernel-level instrumentation and to collect the metrics coming from the maps located in shared memory. Each second, the user-space agent switches the selector in the configuration maps of the network monitor and of the power and performance monitor and collects the RAPL measurements for power attribution. Then, it starts iterating over the *thread map*. For each entry of the *thread map*, the user-space agent collects all the metrics (i.e., power consumption, CPU usage, execution time, cycles, IR, cache references, and cache misses), applies Equation (2) for power attribution, and creates an entry in the *thread table*. It is worth noticing that the contents of the *thread map* are not deleted after data collection, as the entries are evicted in the eBPF code when the *process exit* tracepoint is called. After the user-space agent finished collecting the power and performance metrics, it scans the */proc* folder to find the *cgroup* of each recorded thread. If the cgroup was generated by docker, the user-space agent creates a container entry in a *container table* and aggregates the metrics of all the threads belonging to that *cgroup*. The threads that do not have a cgroup are grouped in the *container table* as *other* to keep track of their activity.

When the user-space agent finished building the *container table*, we have the list of its Process IDs (PIDs) for each container; thanks to this information, we can attribute network transactions to each container. To this aim, the user-space agent starts to iterate over the *network transaction map*. For each entry in the *network transaction map*, the user-space agent creates a transaction group object that contains the aggregate network metrics (i.e., transaction count, byte transmitted, byte received, and average latency) and the raw latency samples. From the raw latency samples of each transaction group, we compute the latency percentiles using *DDSketch* [25], which is a library able to compute quantiles and percentiles over large datasets. Aggregate network metrics and raw latency samples are grouped by container looking at the PID list, then we apply again *DDSketch* to compute the percentiles for each monitored container. After this step, we clear the *network transaction map* to prepare it for the next data collection, whereas *endpoint map* and *connection map* entries are evicted by the eBPF code when the connection is closed by one of the two endpoints. At this point, the agent connects to the *kube-api-server* to retrieve Kubernetes state data (i.e., pods, containers associated to the pods, services, namespaces, and hosts), packs all these data along with the metrics of the *container table*, and submits the sample to the monitoring backend.

3.2 Cluster View and Analysis

The backend is the second data aggregation layer in our infrastructure and its goal is to provide a cluster-level view and an analysis of the performance of the monitored applications. Figure 6 shows the steps the backend takes to manipulate the metrics to achieve this goal. Metrics arrive at the *REST collector* and are propagated to other components that transform them. The final output can be consumed from three components: the *metric frontend*, the *REST endpoint*, and the *graph endpoint*. Section 3.2.1 will describe how we ingest, aggregate, and visualize data, while Section 3.2.2 will detail how we build and analyze the graph of microservices.

3.2.1 Data Ingestion, Aggregation, and Visualization. We designed the backend to be able to process metrics from different clusters of different users to provide a general service. The *REST collector* is the entry point for the metrics that we collect with the monitoring agents and receives both container metrics and Kubernetes state metrics. Container metrics are sent directly to the *metric workers*, while Kubernetes state metrics are sent to the *k8s workers*. Communication with

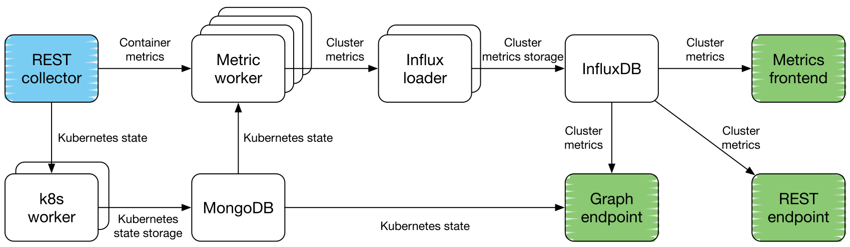


Fig. 6. Backend pipeline structure: The REST collector (highlighted in blue) is the entry point that collects all the data coming from the agents, while metrics frontend, REST endpoint, and Graph endpoint (highlighted in green) are the endpoints used to visualize and query data. Components are connected with a queuing system enabling distribution, replication, and asynchronous communication. Arrows show the data flow.

the components of the backend happens through a queuing system to work asynchronously and to keep samples in the queues storage to avoid data loss in case of system saturation.

The *k8s worker* takes the new Kubernetes state information and updates our internal view of the cluster that is stored inside the backend *MongoDB*[2] instance. At the same time, the *metric worker* takes the container metrics and aggregates them depending on the cluster data that we processed with the *k8s worker*. In particular, container metrics are aggregated by pod and by host. Then, we add information about the namespace to segment them appropriately. Network transaction group metrics follows also the *metric worker* step. Once the *metric worker* finished processing the sample, it sends everything to the *influx loader*, which prepares the data and stores them inside the backend *InfluxDB*[3] instance. At this point, the metrics are ready to be consumed by the backend endpoints: *metric frontend*, *REST endpoint*, and *graph endpoint*. For the *metric frontend*, we leveraged *Grafana*[4] to directly query the time-series database to show the data, although the system is general enough to support other dashboard frontends (e.g., Reference [26]). The *REST endpoint* provides metrics that can be used to control the performance of the observed workloads. Finally, the *graph endpoint* computes and analyzes the interactions between microservices.

3.2.2 Graph Analysis. Once we processed and grouped the metrics depending on the different layers of the infrastructure, we can build a graph that allows analyzing the behavior of the microservices in a distributed scenario. In particular, we are interested in:

(1) the distributed interactions between microservices,
(2) the ability of microservices to promptly respond to the user requests,
(3) how the performance of the distributed system components constrain the overall performance of the system.

The first step to obtain this information is to build the graph, where we consider pods as microservices. For each transaction group (TCP or HTTP), we have the connection information that allows to link microservices and we have the transaction group performance metrics. We query the *MongoDB* instance to get the list of pods and the list of hosts in the cluster with their IP addresses and we query *InfluxDB* to get the transaction groups data. We then use *JGraphT* [30] to build the graph, where pods and hosts are the vertices, while the connections between them are the edges.

[2]https://www.mongodb.com.
[3]https://www.influxdata.com/products/influxdb-overview/.
[4]https://grafana.com.

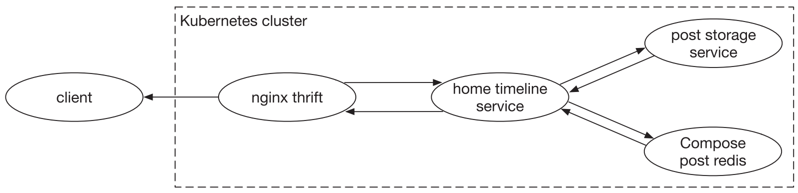


Fig. 7. View of the social network application (DeathStarBench) while reading a home timeline: The graph shows the client, a Nginx reverse proxy, two pods that manage the home timeline, and a redis database. Between client and reverse proxy, we have only one edge, as the client is outside the monitored machines.

One of the issues we found when building the graph is represented by the network manipulation performed by the Kubernetes Container Network Interface (CNI) plugin. In particular, Kubernetes Container Network Interface (CNI) plugins such as *calico*[5] and *flannel*[6] route packets leveraging *IPTables* and Destination Network Address Translation (DNAT). This means that the IP addresses of the pods and the destination IPs may be different. This happens because Kubernetes implements the *service* concept within the cluster. A pod usually cannot connect directly to another pod, because it does not know its destination address. Instead, it uses the address of a known *service* endpoint. The *service* implements a round-robin layer 4 load balancer that internally translates the IP address of the service with the IP address of the pod selected for that connection. The translation is done at the *IPTables* layer without sending out network packets, meaning that we do not have visibility of the server endpoint when we look at the client-side connection. To solve this issue, we store in the backend the service IPs to translate back the IP addresses of the pods. Of course, in the case of multiple server pods behind the service, we connect the client pod to all the server pods, but only when there is a server connection that links back the server pod to the client pod. An example of the resulting graph showing the distributed interactions between pods can be seen in Figure 7. The graph shows a *DeathStarBench* [19] *social network* benchmark stressed with a *read home timeline* workload. Connections between pods are represented by multiple pairs of edges, where each pair has an edge going from the client to the server and an edge going from the server to the client (both with the related metrics).

Starting from the graph, we analyze the performance of the application looking both at the vertex level and at the edge level. At the vertex level, we can analyze the performance metrics we collect to understand which pod represents a bottleneck. If we consider the graph as an open model, we can analyze the asymptotic bounds on performance, thanks to operational analysis [22]. Open models with infinite customers that can arrive at any given time do not have a pessimistic bound on response time. However, we can look at the throughput bound, trying to estimate the load of each pod and deriving the maximum throughput that the system is able to sustain before saturation and as such before response time starts to increase abruptly. All the quantities are evaluated as average values and we consider as arrival rate of the system $\lambda(t)$ the sum of the arrival rates generated by elements of the graph that are outside the distributed application. To correctly perform the operational analysis, we modified the equations provided in Reference [22] to properly model multithreaded workloads co-located on the same multicore processor, as the original equations consider only single core processors serving one workload each.

Equation (3) shows how we compute the utilization $U_i(t)$ of each pod i for a given time t. In Equation (3), we scale the $CPU\_usage_i(t)$ of pod i by the minimum between the number of threads of the pod and the number of cores of the host machine to obtain a value that falls in the range

[5]https://www.projectcalico.org.
[6]https://github.com/coreos/flannel.

[0..1] bounds included:

$$U_i(t) = \frac{CPU\_usage_i(t)}{\min(\#cores_j, \#threads_i)}. \tag{3}$$

Then, we compute the service demand $D_i(t)$ for each pod i for a given time t. The service demand indicates how much time a job stays inside a given pod. In our case, $D_i(t)$ is the ratio between the pod utilization and the arrival rate $\lambda(t)$:

$$D_i(t) = \frac{U_i(t)}{\lambda(t)}. \tag{4}$$

Finally, we compute the estimated arrival rate of saturation $\lambda_{sat}(t)$ of the overall distributed application at a given time t by taking the inverse of the maximum service demand $D_{max}(t)$. The arrival rate of saturation $\lambda_{sat}(t)$ is a theoretical bound that indicates which microservice is close to throughput saturation in a given time and can be used to decide whether it is necessary to scale that microservice to sustain the load of all the incoming customers:

$$\lambda_{sat}(t) = \frac{1}{D_{max}(t)}. \tag{5}$$

If, instead, we look at the edge level, we can find the critical path between the entry points of the distributed application and its pods. To do so, we resorted again to *JGraphT*. We modified the graph structure, pruning the edges by selecting only the server edges with maximum latency percentiles (we can choose the weight among 50th, 75th, 90th, and 99th percentile latency). From this graph, we take the entry points, which are all the vertices without outgoing edges, and we compute the widest path from each entry point to each vertex in the graph. From this set of widest paths, we select the one with the highest score as the critical path.

4 EVALUATION

In this section, we evaluate the proposed monitoring approach. First, we assess the accuracy of the monitoring approach for all the metrics we collect w.r.t. a set of tools that can represent a golden standard for the measurements (Section 4.2). Then, we evaluate the cost of data collection. In particular, we compare our work with state-of-the-art tools that are able to collect a similar set of metrics (Section 4.3). Finally, we leverage our tool to analyze some applications of the *DeathStarBench* benchmark suite [19].

4.1 Experimental Setup

4.1.1 Hardware Platform. We evaluated the proposed approach on a small cluster composed of two Dell PowerEdge r720xd servers, each one equipped with 2x Intel Xeon E5-2680 *Ivy Bridge* with 10 cores each (20 HT) clocked at 2.80 GHz and with 380 GB of RAM. The hardware platform represents a recent mid-range setup. The host OS is a Ubuntu Linux 16.04 with kernel 4.15 with eBPF enabled. On each machine, we installed Docker Community Edition with version 18.06.2 and Kubernetes with version 1.17.3. All the experiments were carried out with HT enabled.

4.1.2 Benchmarks and Goals of the Experimentation. We leverage a different set of workloads to evaluate different aspects of the proposed approach. We centered our evaluation on the *DeathStarBench* benchmark suite [19]. At the time of writing, DeathStarBench provides implementations for Kubernetes environments for the social-network benchmark and the media-microsvc benchmark. The *social-network* benchmark is composed of an *Nginx* web server that works as an entry point for many other microservices. Each microservice covers a particular functionality of a social network deployment like user management, post management, social graph management, URL shorten and advertising, read and write timeline, contents management, search. The benchmark deploys many

Table 1. Experimental Setup for the Social-network Benchmark and the Media-microsvc Benchmark

benchmark	**workload**	**# pods**	**low $\lambda(t)$**	**mid $\lambda(t)$**	**high $\lambda(t)$**	**duration**
social-network	compose post	21	300 req/s	400 req/s	500 req/s	120 s
social-network	read home timeline	5	5,000 req/s	6,000 req/s	8,000 req/s	120 s
media-microsvc	compose review	27	250 req/s	300 req/s	350 req/s	120 s

For each workload, we show the number of pods involved and the request rates for low, medium, and high configurations.

database instances (e.g., MongoDB, Redis, and Memcached) to support each microservice functionality, where there is no shared database instance between microservices. The *social-network* benchmark provides two workloads, one that simulates users reading their home timeline and one that simulates users composing posts. The *media-microsvc* benchmark, instead, implements a media reviewing, renting, and streaming platform. This application is composed of identification services (users and movies), a review composition service, a page composition service, services for users and reviews consultation, and a video streaming service. As in the social-network application, the microservices store data within databases and in-memory stores such as MongoDB, Redis, and Memcached. The *media-microsvc* benchmark provides one workload that simulates users composing reviews. These two applications are able to effectively resemble the interactions between microservices in a complex deployment, and, as such, they can provide interesting insights on how to monitor them. Table 1 shows how we configured the benchmarks: For each workload, we test three different levels of intensity named *low*, *mid*, and *high*. We used these configurations to evaluate the overhead and the network metrics accuracy of the proposed solution.

To evaluate the accuracy of the low-level measurements, the execution time, the CPU usage, and the power consumption metrics, we leveraged three micro-benchmarks from the NAS Parallel Benchmark (NPB) suite [2]. Although they are not commonly referred to as representative of microservices, they provide stable benchmarks to evaluate the measurement mechanisms (particularly for low-level metrics). Here, we provide details about the benchmarks selected from the NPB suite:

- **Embarrassingly Parallel (EP)** is a kernel that generates pairs of Gaussian random deviates and is a benchmark highly CPU-intensive and CPU-bound;
- **Multi Grid (MG)** is a memory-intensive benchmark that performs a simplified multi-grid calculation, it requires highly structured long-distance communication, and it tests both short and long-distance data communication;
- **Conjugate Gradient (CG)** is a mixed CPU and memory-intensive workload that performs an approximation to the smallest eigenvalue of a large, sparse, symmetric-positive definite matrix through a conjugate gradient method.

Before resorting to the NPB benchmarks, we tested the accuracy of the low-level metrics with the two aforementioned benchmarks of the *DeathStarBench* benchmark suite. To do so, we selected the *perf* [13] tool to provide the golden standard for the performance metrics. Although perf can attribute metrics to each running *cGroup*, the overhead generated heavily affected the execution of the benchmarks (+12.93%, +14.6%, and +9.16% perf overhead w.r.t. our solution for read home timeline, compose post, and compose review workloads, respectively). This prevented us to compare the results obtained with perf w.r.t. our solution when executing the tools separately. Unfortunately, it is not possible to run the tools concurrently, as perf resets the counters after reading them.

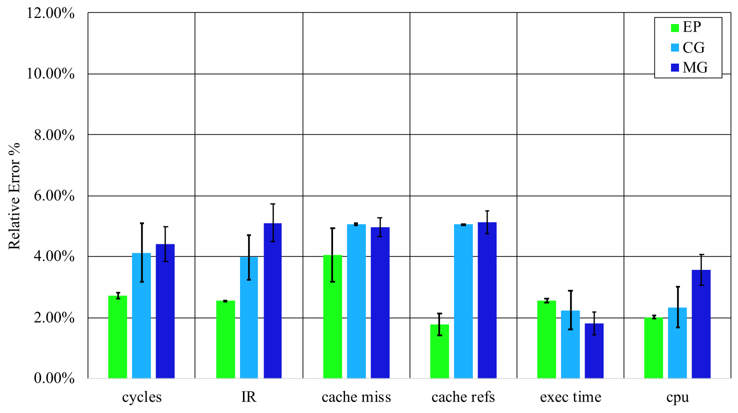


Fig. 8. Relative Error of cycles, IR, cache references, cache misses, execution time, and CPU usage of our tool w.r.t. the golden standard measured on 30 runs of EP, MG, and CG. Bars show 95% confidence interval.

Table 2. Average Execution Times (with 95% Confidence Interval) and Overhead of the Monitoring Tool for the Three NPB Applications with Problem Size C and 40 Threads Each

	monitor	**no monitor**	**overhead**
EP	6.947s ± 0.004	6.950s ± 0.003	−0.05%
CG	21.116s ± 0.140	20.207s ± 0.192	4.49%
MG	8.435s ± 0.051	8.303s ± 0.038	1.59%

4.2 Metrics Accuracy

We leveraged both the NPB and the three DeathStarBench workloads to assess the accuracy of each metric we collect. In particular, we used the NPB with problem size *C* and 40 threads to measure the accuracy of cycles, IR, cache references, cache misses, execution time, and CPU usage on a multi-socket server. We leveraged *perf* as a golden standard for cycles, IR, cache references, and cache misses, while the Linux tool *time* was used as the golden standard for execution time and CPU usage. We then used the NPB with problem size *C* also to assess whether the power attribution mechanism described in Section 3.1.1 provides good results. Finally, to measure the accuracy of the latency metrics (i.e., average latency, request count, byte transmitted, 50th, 75th, 90th, and 99th percentile), we monitored the workloads described in Table 1 and we compared our results with what we obtained from the load generator. We run each experiment 20 times.

4.2.1 Performance Metrics. Figure 8 shows the Relative Error (RE) between our measurements and the golden standard results for the cycles, IR, cache references, cache misses, execution time, and CPU usage metrics. EP metrics gave us a result that is always below 4% in Relative Error (RE). For CG and MG, the maximum RE is near 5% for the cache references and cache misses metrics (and IR for MG), while the cycles metric has an RE below 4.5%. Finally, execution time has an RE below 3% for all experiments and CPU usage has an RE near 2% for EP and CG, while MG has an RE of 3.56%. The results we obtained allow us to say that the monitoring tool is accurate on those metrics considering the goal of the proposed approach, that is, monitoring and not profiling. We can find the reasons for the RE values on the different overheads posed by the golden standard tool and our implementation. As we can see from Table 2, our tool has a negligible impact on EP, while MG and CG have overheads that are comparable with the RE values shown in Figure 8. Moreover,

Table 3. Execution Time, Energy, and Power Consumption Values (with 95% Confidence Interval) of EP and MG (10 Threads Each) When Executed in Isolation or in Co-location with a Fully Disjoint Set of Cores

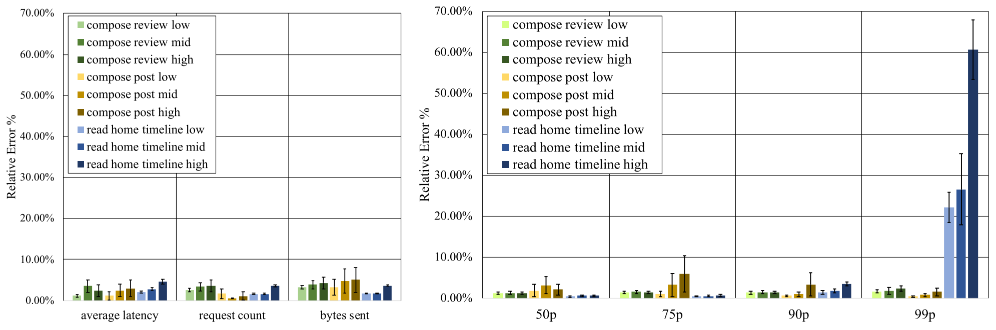
	configuration	exec time (s)	energy (J)	power (W)	time delta	energy delta	power delta	power scaled (W)	power scaled delta
EP	**co-location**	25.06s ± 0.13	1,168.23 ± 11.45	46.62 ± 0.43	7.09%	−9.30%	−15.30%	49.93	−2.87%
	isolation	23.40 ± 0.12	1,288.08 ± 15.86	55.04 ± 0.55				51.40	
MG	**co-location**	16.79 ± 0.09	736.33 ± 9.33	43.84 ± 0.47	21.23%	−15.78%	−30.54%	53.16	2.10%
	isolation	13.85 ± 0.10	874.33 ± 6.20	63.12 ± 0.23				52.07	

our tool does not compute metrics on process exit, as these metrics are not actionable anymore. In particular, if the monitoring tool is used in a control loop to act on the monitored system, then when a process exits, the control loop cannot act anymore on the process, as its effects are no longer visible.

4.2.2 Power Consumption Metrics. Table 3 shows the results of EP and MG for what concerns the power attribution metrics. Unfortunately, a golden standard for power attribution in case of workloads co-location does not exist. Within this context, we can try to evaluate if power attribution provides reasonable results using the NPB micro-benchmarks, although formal correctness cannot be verified. We configured EP and MG to run with 10 threads and we pinned them to a disjoint set of HT of the first socket of the test server: 10 EP threads on 5 cores and 10 MG threads on the other 5 cores. Then, we run the two benchmarks in co-location as well as in isolation. We decided to not use CG, because the benchmark has a higher execution time w.r.t. EP and MG, and most of its execution would have happened in isolation.

As we can see from Table 3, both benchmarks show an increase in execution time and a decrease in total energy consumed when running in co-location. It is clear that co-location increases contention and slows down the access to shared resources, however, if we compute the *scaled power consumption* (i.e., we compute the power consumption of a benchmark run in a given configuration using its energy and the execution time of the other configuration), we can see that the difference between the power consumption of the two configurations is of ≃2% both for EP and MG. This happens as both EP and MG are compute-bound micro-benchmarks and their operations are regular. This means that changing their pace can change almost linearly the execution time and the energy.

4.2.3 Network Metrics. Figure 9 shows the results we obtained running the workloads of Table 1 and comparing the network metrics collected by our solution with the ones gathered by our golden standard: *wrk2.* Looking at the results related to average latency, requests count, and bytes sent by the workloads (shown in Figure 9(a)), we can see that the proposed tool is able to measure them with an error that is less than 5% on average. The 95% confidence interval shows that the worst case for this set of metrics is represented by the compose post workload with the bytes sent metric, where the error falls in the range 5% ± 3.08%. Part of these errors are due to the different measurement mechanism: wrk2 measures the metrics on the client-side within the application, while our monitoring tool measures them within the kernel on the server-side. These errors are limited, as the proposed tool is able to compute those metrics considering all the requests and not just a subset. Figure 9(b) shows instead the measurement error for the latency percentiles, which are computed from a reservoir of 240 items collected each second for each open connection. Before looking at Figure 9(b), it is fundamental to keep in mind that the results are computed for a single HTTP endpoint and with a single client sending all the requests to stress test the data collection system. In a more general setting, the requests sent to the workloads of Table 1 would have been



(a) Average RE between our tool and wrk2 for avg latency, requests and bytes transmitted.

(b) Average RE between our tool and wrk2 for 50th, 75th, 90th, 99th percentile latency.

Fig. 9. Average RE between our tool and wrk2 for network performance metrics for the three workloads with different amounts of requests. Bars show 95% confidence interval.

generated by multiple clients. In this case, our tool would have been able to create multiple reservoirs, one for each TCP connection and one for each HTTP endpoint requested by each client, increasing the accuracy. In general, it is always possible to increase the reservoir size, although it is necessary to evaluate the tradeoff between accuracy and measurement overhead. Looking at Figure 9(b), for what concerns the compose review workload, the worst-case error is 2.35% on average for the 99th percentile metric for the high configuration. The compose post workload, instead, has the errors on all the metrics below 3.5% on average except for the 75th percentile of the high configuration, where the error is 5.97% on average. For what concerns these two workloads, the reservoir sampling is behaving in a good way, providing samples that are representative of the requests performed by wrk2. The read home timeline workload, instead, shows a different behavior. While for the 50th, 75th, and 90th percentile latency metrics the errors shown in Figure 9(b) stays below 3.5% on average, the 99th percentile latency shows errors of 22.17% for the low configuration, 26.61% for the mid configuration, and 60.70% for the high configuration. This error is due to the sampling activity performed by our tool. In particular, the read home timeline workload generates for each second requests for the high, mid, and low configurations that are, respectively, 33, 25, and 20 times higher w.r.t. the number of samples collected each second by our tool. The 99th percentile metric becomes unreliable with this load level.

To investigate to which extent our tool can be effectively used to measure the 99th percentile latency, we run the read home timeline workload with an increasing number of requests (up to 16 times the size of the reservoir) and we show the measurement errors in Figure 10. We report also the 50th, 75th, and 90th percentile latency for completeness. As we can see from Figure 10, the proposed tool is able to measure the 99th percentile latency up to 16 times the reservoir size with an error that is always below 3% on average. When we increase the load further, as shown in Figure 9(b), such measure becomes unreliable.

To demonstrate that the main source of uncertainty in the quality of the measurements is the number of requests collected each second for each connection, we tested the workloads of Table 1 in a more variable setup. In particular, we added a 10 ms ± 5 ms normally distributed delay on all the network requests performed by wrk2 and the workloads using the Linux tool *tc*. The results of this experiment are shown in Figure 11. Here, we report only the high configuration, as the workloads in the three configurations behaved in the same way. As we can see from Figure 11(a), the errors for the average latency, request count, and bytes sent are limited and always below 2% on average. We obtained the same behavior for 50th, 75th, 90th, and 99th percentile latency (shown

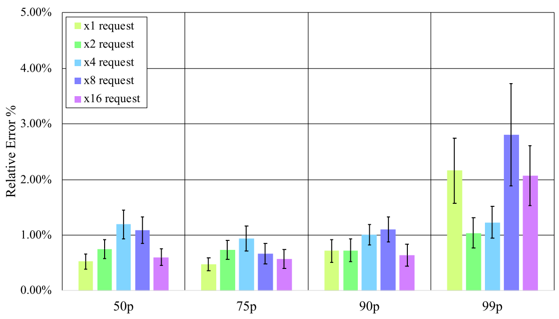
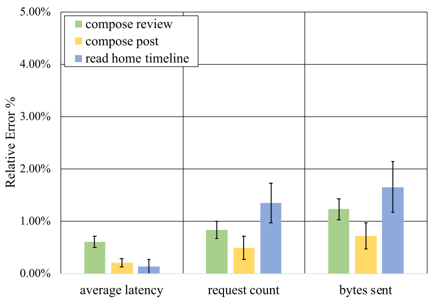
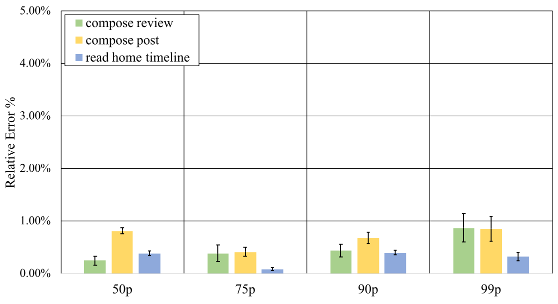


Fig. 10. Average RE between our tool and wrk2 for 50th, 75th, 90th, 99th percentile latency for the read home timeline workload with different amount of requests (from 1 to 16 times w.r.t. the capacity of the samples reservoir). Bars show 95% confidence interval.



(a) Average RE between our tool and wrk2 for avg latency, requests and bytes transmitted.



(b) Average RE between our tool and wrk2 for 50th, 75th, 90th, 99th percentile latency.

Fig. 11. Average RE between our tool and wrk2 for network performance metrics for the three workloads (high configuration) with network delay of 10ms ± 5ms normally distributed. Bars show 95% confidence interval.

in Figure 11(b)) with an error always below 1.5%, even with the read home timeline workload. The reason why we obtained this result is twofold. On the one hand, the delay reduced the number of processed requests per second, allowing the tool to build a more comprehensive view of the performance of the workloads. On the other hand, the way in which we perform the measure is able to account for all the network delays and errors of the TCP protocol, thus providing very accurate results.

4.3 Agent Overhead

Monitoring activity should be carried out without imposing too much overhead on the monitored applications. Within this section, we will evaluate the overhead of the proposed monitoring tool w.r.t. other two similar state-of-the-art solutions: *Weave scope* [40] and *Sysdig cloud* [38]. We selected *Weave scope* as a baseline because it monitors just CPU, memory usage (by scanning */proc*), and the number of open connections between pods. Its overhead should be always lower than the one we introduce in the system, as *Weave scope* does not monitor performance counters and network transactions. *Sysdig cloud*, instead, is a monitoring tool that measures resource usage, network I/O, and file I/O using *system calls* captured by a kernel module that sends to user-space all the raw system call traces. To have a fair comparison, we disabled all the metrics that *Sysdig cloud* collects and we do not collect, with the only exception of file I/O, as some of the network

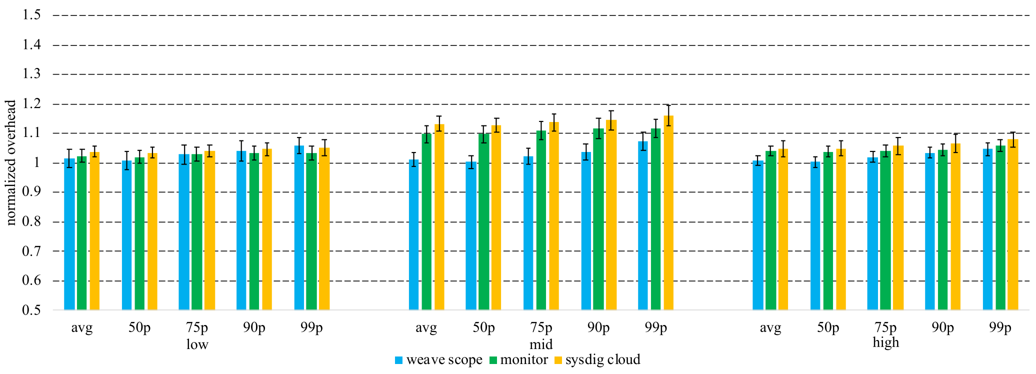


Fig. 12. Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and Sysdig cloud while running the compose post workload for three different workload levels.

operations can be performed with file-related system calls with a network File Descriptor (FD) as a parameter [6]. We measured the overhead of our approach leveraging the workloads detailed in Table 1. The overhead is computed as the increase in latency of the network requests of the benchmarks, obtained by wrk2 from the average latency measurement and the 50th, 75th, 90th, and 99th percentile latency measurements. We deployed each application on the two test machines using Kubernetes and we pinned each pod to a given machine to avoid too much noise in the resulting data. Each test was executed 20 times. Here, we show average results with 95% confidence interval.

Although the testing setup is small, it can be considered a relevant setup, as the overhead of the proposed monitoring tool depends only on the instrumentation performed through eBPF. The instrumentation adds execution time to the network operations performed by the benchmarks and to the context switches the OS performs during execution. For this reason, if we consider a microservice benchmark executed on a small cluster w.r.t. the same benchmark with the same amount of containers executed on a larger one, the smaller cluster is a far more challenging environment. This happens because the number of network requests captured by our tool remains the same across the two setups, but in the small cluster setup the pressure on our monitoring tool is higher, as the network requests and the context switches are not spread across many monitoring agents but are instead condensed into few ones.

Figure 12 shows the normalized overhead w.r.t. a no monitor execution of the compose post workload of *Weave scope*, our solution, and *Sysdig cloud* for the three levels of the benchmark we described in Table 1. The results of Figure 12 confirm that *Weave scope* provides very low overhead in all configurations. Our solution is able to outperform *Sysdig cloud* in all the configurations in all cases, providing an overhead of $\simeq$ 10% at most, obtained the mid configuration. The 95% confidence interval shows that the execution of our tool was sufficiently regular across all the runs.

Figure 13 shows the normalized overhead w.r.t. a no monitor execution of the compose review workload of *Weave scope*, our solution, and *Sysdig cloud* for the three levels of the benchmark we described in Table 1. This experiment shows an interesting behavior w.r.t. the previous one. In general, our solution always outperforms *Sysdig cloud* also in this case, but it is also able to outperform *Weave scope* in terms of impact on the average latency and on the 99th percentile latency of the three workload configurations. On the contrary, *Weave scope* behaves better in the 50th, 75th, and 90th percentile latency.

Figure 14 shows the normalized overhead w.r.t. a no monitor execution of the read home timeline workload of *Weave scope*, our solution, and *Sysdig cloud* for the three levels of the benchmark we described in Table 1. In this case, our solution behaved unexpectedly: *Sysdig cloud* was able to

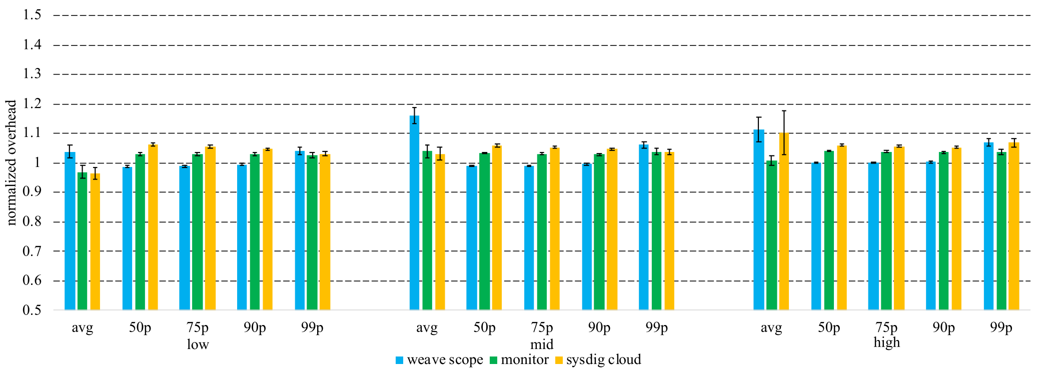


Fig. 13. Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and Sysdig cloud while running the compose review workload for three different workload levels.

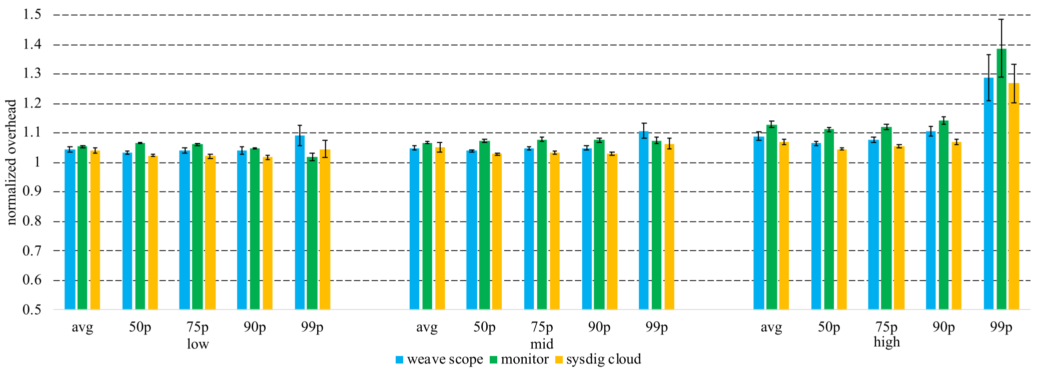


Fig. 14. Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and Sysdig cloud while running the read home timeline workload for three different workload levels.

outperform it in all cases for all configurations. It is worth noticing that the overhead introduced by our solution w.r.t. a no monitor execution is, in the worst case, of 0.2 ms on the average network latency metric and 2 ms in the 99th percentile latency metric. To further investigate this result, we increased the number of parallel connections (from 100 to 500) and we reduced the number of requests per second (2,000 for high, 1,500 for mid, 1,000 for low). The idea is to reduce the request per connection ratio to assess the overhead on the requests themselves and the connection setup. Figure 15 shows the result of this new experiment. Our solution behaves better than *Weave scope* and *Sysdig cloud* in all the cases except the 50th percentile latency metric for the low and mid configuration. This result is extremely important, as it shows that our solution behaves better in case of many parallel connections, which is far more common w.r.t. what we presented in Figure 14.

4.4 Benchmark Study

To give an overview of the insights we can provide, we decided to stress test the social network benchmark with a compose post workload with 10,000 R/s divided across 60 connections. We configured the *graph analysis* to retrieve the last connection data of each pod in a time interval of 5 seconds and we report the output data in Table 4 for what concerns the operational analysis and

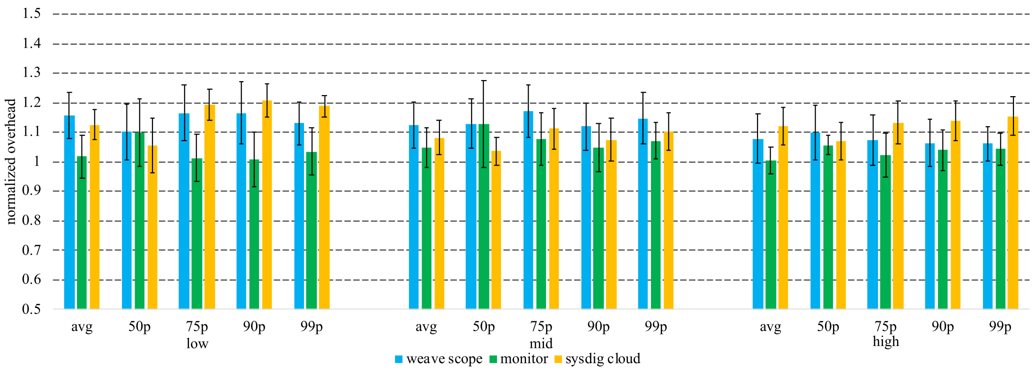


Fig. 15. Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and Sysdig cloud while running the read home timeline workload over the social network benchmark with more connections (500 instead of 100) and less requests per second (2,000 req/s for high, 1,500 req/s for mid, 1,000 req/s for low).

Table 4. Example Output of the Operational Analysis with Pod Name, Utilization, Arrival Rate of Each Pod, Maximum Arrival Rate for Each Pod, Power Consumption, Number of Threads, and HTTP and TCP Average Latency for a DeathStarBench Social Network Benchmark Loaded with 10,000 R/s with 60 Open Connections

						average latency	
pod name	$U_i(t)$ **%**	$\lambda_i(t)$	**max** $\lambda_i(t)$	**power (W)**	**# threads**	**HTTP (ms)**	**TCP (ms)**
compose-post-redis	96.90%	11138	11,493.57	7.40	1	-	1.36
user-timeline-redis	7.00%	1348	19,249.45	1.03	1	-	0.01
social-graph-service	5.56%	401	7,210.87	7.43	8	-	0.57
nginx-thrift	5.53%	1571	28,365.00	13.41	32	39.34	19,27
social-graph-redis	5.14%	596	11,574.56	0.55	1	-	0.08
user-memcached	4.06%	1288	31,663.31	3.37	8	-	0.00
write-home-timeline-rabbitmq	3.03%	741	24,401.31	3.55	15	-	0.23
text-service	2.53%	1461	57,553.38	8.48	231	-	27.39
user-mention-service	2.51%	2845	113,337.91	7.98	61	-	4.87
compose-post-service	2.36%	2899	122,493.18	17.27	296	-	13.76
user-timeline-service	1.51%	471	31,116.93	4.33	14	-	0.53
post-storage-service	1.46%	714	48,796.94	4.40	24	-	0.52
url-shorten-service	1.08%	1420	130,699.35	4.23	112	-	13.07
unique-id-service	0.83%	879	105,375.93	3.75	62	-	12.67
media-service	0.81%	777	94,905.51	3.40	62	-	12.49
user-timeline-mongodb	0.51%	962	188,029.60	2.17	30	-	0.16
post-storage-mongodb	0.28%	481	168,390.95	0.92	26	-	0.24
url-shorten-mongodb	0.25%	537	214,770.44	1.12	46	-	0.20
user-service	0.19%	480	242,366.64	1.76	60	-	16.12
user-mongodb	0.16%	537	318,997.27	1.01	49	-	0.13
social-graph-mongodb	0.14%	228	152,210.67	0.35	22	-	0.21

Arrival rate 1576 R/s, estimated saturation arrival rate 1626 R/s.

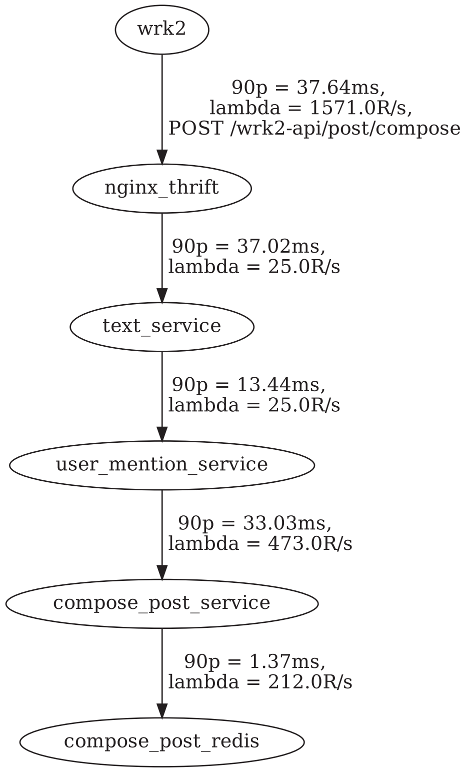


Fig. 16. Critical path example of the DeathStarBench social network loaded with 10,000 R/s with 60 open connections. Edge weights are the 90th percentile latency.

in Figure 16 for what concerns the critical path. We do not report the entire graph due to space limits.

Although we loaded the benchmark with 10,000 R/s, the system can respond only to 1,576 R/s. To investigate the source of this bottleneck, we resorted to the operational analysis formulas described in Section 3.2.2. As we can see from Table 4, the *compose-post-redis* pod is almost saturated with a $U_i(t)$ equal to 96.9%. Moreover, *compose-post-redis* is responding to 11,138 R/s coming from the other pods in the application. If we apply Equation (5) to this case, we find that $\lambda_{sat}(t)$ is 1,626 R/s, which is extremely close to the current $\lambda(t)$. If we take another look at Table 4, we can see that the application entry point represented by *nginx-thrift* and most of the other application components have very low utilization, meaning that a strategy to replicate the *compose-post-redis* database could solve the bottleneck. Another interesting insight that we can obtain from Table 4 is that the average latency of *nginx-thrift* is higher w.r.t. most of the other containers within the microservice application. This happens as *nginx-thrift* serves as an entry point and reverse proxy for the other pods of the application. For this reason, its latency is affected by the latency of the pods it has to connect to to build a full response that is sent to the client as a response to an HTTP request. Such connections may happen in parallel or sequentially, depending on the data dependencies of the application. Finally, Table 4 shows also that the most utilized pod in the application is not the one with the highest power consumption. Solving the bottleneck could change the power profile of the system: This aspect should be considered when building applications that aim to be efficient in terms of performance, resource usage, and power consumption.

To further analyze the relations between pods in the social network application, we found the critical path between the entry point and all the other pods. We configured the graph to have as edge weights the 90th percentile latency of the connection that is represented by the given edge. Of course, two pods may have multiple connections between one another; however, given that we are using a percentile latency, we decided to prune the graph by taking the slowest edge connecting each pair of pods. Figure 16 shows the critical path with the chain of connections from the entry point to the last pod in the path. As we can see, within the chain, we have also the same *compose-post-redis* pod that we found out to be the bottleneck of the application. Solving the bottleneck means that the increased amount of requests will affect all the pods in the chain and, for this reason, it is fundamental to verify whether the pods in the chain will be able to sustain the new load.

5 CONCLUSION AND FUTURE WORK

In this article, we proposed a novel black-box monitoring approach to build a unified view from the architectural level to the application performance level of a microservice-based application. The proposed approach leverages eBPF to generate the metrics at the kernel level to transparently build a view that aims to be accurate and with low overhead on the monitored applications. Within the article, we showed how we collect data about resource usage, performance counters, power consumption, and network performance without instrumenting the user code. We then used such data to analyze microservice applications w.r.t. their interactions with the external world as well as w.r.t. the internal connections and dependencies. The experimental campaign showed that the proposed approach is reasonably accurate w.r.t. its goal, which is performance monitoring. We also showed how the proposed approach can introduce less overhead in the monitored application w.r.t. a solution that is based on similar technologies but with a different data extraction strategy.

Although the proposed approach seems promising, it should be integrated with more data sources to build a better view of the monitored applications. In particular, monitoring file and disk I/O will allow detecting bottlenecks that depend on the performance of the disks. Moreover, we will introduce predictive models w.r.t. the measured latency to detect in advance bottlenecks that will saturate the capacity of the system. This will allow a timely reaction when such situation occurs.

SOURCE CODE RELEASE

The agent code of the proposed black-box monitoring approach is available as open-source at: https://github.com/necst/DEEP-mon. We welcome feedback and suggestions, hoping that the release of the tool will be helpful to improve the monitoring of cloud-native workloads and to advance the research in the field.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments, which improved the quality of this article. In addition, the authors would like to thank Mr. Andrea Damiani for his helpful insights and comments on the manuscript.

REFERENCES

[1] Amedeo Asnaghi, M. Ferroni, and M. D. Santambrogio. 2016. DockerCap: A software-level power capping orchestrator for docker containers. In *Proceedings of the Conference on Computational Science and Engineering (CSE'16) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC'16) and 15th International Symposium on Distributed Computing and Applications for Business Engineering (DCABES'16)*. IEEE, 90–97.

[2] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, et al. 1991. The NAS parallel benchmarks. *Int. J. Supercomput. Applic.* 5, 3 (1991), 63–73.

[3] BCC. 2020. BPF Compiler Collection (BCC). Retrieved from https://www.iovisor.org/technology/bcc.

[4] Andrew Begel, Steven McCanne, and Susan L. Graham. 1999. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *ACM SIGCOMM Comput. Commun. Rev.*, Vol. 29. ACM, 123–134.

[5] Frank Bellosa. 2000. The benefits of event: Driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System.* ACM, 37–42.

[6] Rolando Brondolin, Matteo Ferroni, and Marco Santambrogio. 2019. Performance-aware load shedding for monitoring events in container based environments. *ACM SIGBED Rev.* 16, 3 (2019), 27–32.

[7] Rolando Brondolin, Tommaso Sardelli, and Marco D. Santambrogio. 2018. DEEP-mon: Dynamic and energy efficient power monitoring for container-based infrastructures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18).* IEEE, 676–684.

[8] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Queue* 14, 1 (2016), 70–93.

[9] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. 2017. A Kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM'17).* IEEE, 1–6.

[10] Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. 2019. Microservices monitoring with event logs and black box execution tracing. *IEEE Trans. Serv. Comput.* (2019).

[11] Cncflandscape. 2020. CNCF Cloud Native Landscape. Retrieved from https://landscape.cncf.io.

[12] Datadog [n.d.]. Datadog. Retrieved on July 2020 from https://www.datadoghq.com.

[13] Arnaldo Carvalho De Melo. 2010. The new linux "perf" tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[14] Luca Deri, Maurizio Martinelli, and Alfredo Cardigliano. 2014. Realtime high-speed network traffic monitoring using ntopng. In *Proceedings of the 28th Large Installation System Administration Conference (LISA'14).* 78–88.

[15] Luca Deri, Samuele Sabella, and Simone Mainardi. 2019. Combining system visibility and security using eBPF. In *Proceedings of the Italian Conference on Cybersecurity.*

[16] Docker 2020. Docker. Retrieved from https://www.docker.com.

[17] Matteo Ferroni, Juan A. Colmenares, Steven A. Hofmeyr, John D. Kubiatowicz, and Marco D. Santambrogio. 2018. Enabling power-awareness for the Xen hypervisor. *ACM SIGBED Review* 15, 1 (2018), 36–42.

[18] Martin Fowler and James Lewis. 2014. Microservices—A definition of this new architectural term. Retrieved from http://martinfowler. com/articles/microservices.html.

[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems.* 3–18.

[20] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems.* 19–33.

[21] Emir Imamagic and Dobrisa Dobrenic. 2007. Grid infrastructure monitoring system based on Nagios. In *Proceedings of the Workshop on Grid Monitoring.* ACM, 23–28.

[22] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. 1984. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Inc.

[23] Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. 2020. Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems. *IEEE Trans. Parallel Distrib. Syst.* 32, 1 (2020), 184–198.

[24] Matthew L. Massie, Brent N. Chun, and David E. Culler. 2004. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Comput.* 30, 7 (2004), 817–840.

[25] Charles Masson, Jee E. Rim, and Homin K Lee. 2019. DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proc. VLDB Endow.* 12, 12 (2019), 2195–2205.

[26] Benjamin Mayer and Rainer Weinreich. 2017. A dashboard for microservice monitoring and management. In *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW'17).* IEEE, 66–69.

[27] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter Conference*, Vol. 93.

[28] Farnaz Moradi, Christofer Flinta, Andreas Johnsson, and Catalin Meirosu. 2017. Conmon: An automated container based network performance monitoring system. In *Proceedings of the IFIP/IEEE Symposium on Integrated Network and Service Management (IM'17).* IEEE, 54–62.

[29] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, Vol. 710.

[30] Barak Naveh et al. 2008. JGraphT. Retrieved from http://jgrapht. sourceforge. net.

[31] Newrelic [n.d.]. NewRelic. Retrieved on July 2020 from https://newrelic.com.

[32] Ayman Noor, Devki Nandan Jha, Karan Mitra, Prem Prakash Jayaraman, Arthur Souza, Rajiv Ranjan, and Schahram Dustdar. 2019. A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments. In *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD'19)*. IEEE, 156–163.

[33] Fábio Pina, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardroom. 2018. Nonintrusive monitoring of microservice-based systems. In *Proceedings of the IEEE 17th International Symposium on Network Computing and Applications (NCA'18)*. IEEE, 1–8.

[34] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, and Rajkumar Buyya. 2015. A framework and algorithm for energy efficient container consolidation in cloud data centers. In *Proceedings of the IEEE International Conference on Data Science and Data Intensive Systems (DSDIS'15)*. IEEE, 368–375.

[35] Prometheus [n.d.]. Prometheus. Retrieved on July 2020 from https://prometheus.io.

[36] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro* 32, 2 (Mar. 2012), 20–27. DOI: https://doi.org/10.1109/MM.2012.12.

[37] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. 2013. Power containers: An OS facility for fine-grained power and energy management on multicore servers. *ACM SIGPLAN Not.*, Vol. 48. ACM, 65–76.

[38] sysdig [n.d.]. Sysdig. Retrieved on July 2020 from https://sysdig.com.

[39] tcpdump [n.d.]. Tcpdump. Retrieved on July 2020 from http://www.tcpdump.org.

[40] weavescope [n.d.]. Weave scope. Retrieved on July 2020 from https://www.weave.works/oss/scope/.

[41] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. 2014. HaPPy: Hyperthread-aware power profiling dynamically. In *Proceedings of the USENIX Annual Technical Conference*. 211–217.

Received March 2020; revised July 2020; accepted August 2020