Invited Paper: Towards Efficient Microservice Communication

Divyanshu Saxena
The University of Texas at Austin

William Zhang
The University of Texas at Austin

Madhav Tummala
The University of Texas at Austin

Saksham Goel
The University of Texas at Austin

Aditya Akella
The University of Texas at Austin

ABSTRACT

Distributed applications on the cloud are being developed and deployed as microservices as opposed to the monolithic architecture. Service Meshes have emerged as a way of specifying communication policies between microservices. Service Meshes have the potential to abstract the networking requirements of distributed applications from the application logic. However, current service mesh frameworks introduce significant performance and resource overheads. We study the overheads of service meshes and make a case for redesigning both the control plane and data plane for service meshes. First, we propose the notion of Application Defined Middleboxes, which makes it possible for the mesh control planes to reduce the overheads by optimizing where to implement application policies. Second, we demonstrate preliminary ideas on accelerating the data plane to further reduce the overheads.

CCS CONCEPTS

• **Networks** → **Network services**; **In-network processing**; **Network management**; **Cloud computing**; • **Computer systems organization** → **Cloud computing**.

ACM Reference Format:
Divyanshu Saxena, William Zhang, Madhav Tummala, Saksham Goel, and Aditya Akella. 2023. Invited Paper: Towards Efficient Microservice Communication. In *The 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED 2023), June 19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3584684.3597267

1 INTRODUCTION

Applications are becoming more and more distributed in nature to deal with the ever-increasing scale and to support agile development. Cloud applications are no exception - applications on the cloud are moving from a monolithic architecture to microservice deployments - where the application is developed in the form of hundreds of *loosely-coupled* services. This enables enhanced scalability, accelerated development cycles, and language and framework heterogeneity for the application [13].

The microservice architecture introduces additional interfaces that can be configured to realize complex policies for telemetry, tracing, traffic management and authorization. In addition, these policies often require fine-grained control over layer 7 requests. However, implementing these networking policies in the application code is difficult to configure, deploy, and update.

Service meshes provide a way to lessen the burden on application developers by abstracting out all microservice communication policies into a separate process that can be deployed *alongside* the application container. This proxy is typically deployed as a container itself, commonly known as the *sidecar*. A control plane accepts user policies and pushes them to the dataplane sidecars. Several open-source service meshes [3, 4, 7] use this architecture.

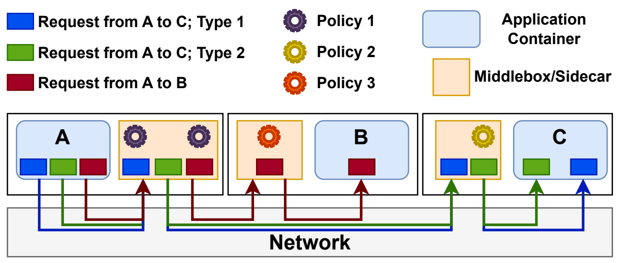
Today, the *de-facto* approach is to "inject" a sidecar into the microservice graph for each service container. The rationale is that it limits the blast radius in case of a sidecar failing, isolates the traffic of different services and scales sidecar resources proportionally with the traffic load to the application. Unfortunately, this architecture also imposes serious performance bottlenecks on the applications. In our experiments, we found that using sidecars increases CPU usage by 37-63%, memory usage by 37-41% and leads to 1.54-2.84× higher end-to-end application latencies.

We observe that the idea of extracting communication policies outside of the application and into the network has been explored before for lower layer policies with network middleboxes (e.g. NATs, Firewalls, IDS, Proxy Caches). Traditional middleboxes operate on L3/L4 policies while service meshes usually cater to L7 policies. Besides the difference in target network layers, they also differ in how they couple policies with the applications. We identify that traditional middleboxes operate on the principle of *Decoupled Enforcement and Placement* - communication policies are run in middleboxes that are decoupled from the compute servers, as they are deployed in separate servers or dedicated switches and routers. The sidecars in current mesh frameworks can essentially be thought of as middleboxes with the crucial difference that current mesh frameworks *couple Enforcement and Placement* - all the policies of a service are consolidated in a single sidecar container coupled with the specific service they are serving.

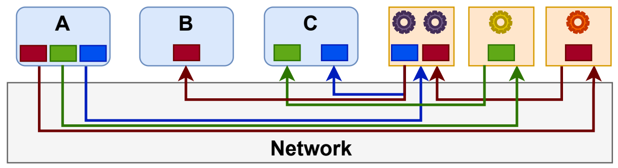
We argue that mesh policies can be thought of being run in *application-defined middleboxes* (ADMs) that form the service mesh layer underlying the application containers. Unlike the current sidecar model and traditional middleboxes, ADMs operate on the principle of *Decoupled Enforcement - Coupled Placement*. Our key insight is that while it is important for the ADMs to be coupled with the services ; the policies apply on the traffic and can hence, be enforced on either the sender service or the receiver service. Decoupling enforcement allows flexbility in choosing whether to enforce a given policy on the sender or the receiver. We show that this observation can be used to reduce the total number of ADMs needed, which in turn, can help in reducing overheads (Section 4).

Figure 1 shows the distinction between the three approaches (sidecars, traditional middleboxes, and ADMs) on a toy example using

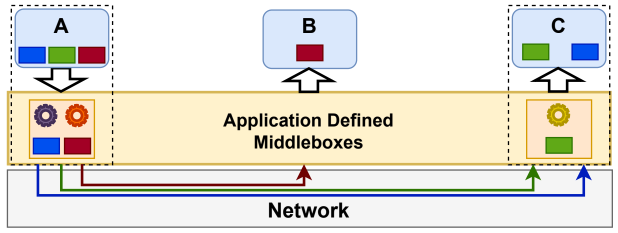
ApPLIED 2023, June 19, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0128-3/23/06.
https://doi.org/10.1145/3584684.3597267



(a) Sidecars (*Coupled Enforcement and Placement*): All policies of a service consolidated into a single sidecar per service container causing extraneous containers, network hops and processing.



(b) Middleboxes (*Decoupled Enforcement and Placement*): Dedicated middlebox for each policy, deployed separately from application compute servers, leading to common middleboxes for all applications.



(c) Application Defined Middleboxes (*Decoupled Enforcement - Coupled Placement*): Policy enforcement decoupled from the services, while middleboxes are coupled with services. Allows for per-service policy enforcement at either the sender or receiver, resulting in fewer containers and reduced processing overheads.

Figure 1: Comparison of ADMs against today's sidecars and traditional middleboxes.

three services and three policies. Service A sends requests to services B and C. Additionally, the traffic from A to C can be categorized into two types of requests with the first type requiring Policy 1 and the second type requiring Policy 2. Meanwhile, traffic to service B requires Policies 1 and 3. In Figure 1a, each sidecar runs one of the policies and the sidecars are coupled with the application containers. Note that the sidecar reconstructs payloads for all requests, irrespective of whether a policy is to be run on them or not, e.g. sidecar at A reconstructs requests meant for policy 2. Figure 1b shows middleboxes separated from the application containers, each running one of the policies (we assume the policies are very different in nature, requiring three different middleboxes to implement them). Further, the request might need to traverse multiple middleboxes, e.g. request from A to B goes to middlebox enforcing policy 3 then to the middlebox enforcing policy 1. Finally, with ADMs in Figure 1c, policies for B (1 and 3) are decoupled from B and instead run at the ADM coupled with A - leading to an overall fewer number of ADMs.

Our proposal is to tackle the performance overheads of service meshes from both aspects – the control plane and the data plane. For the control plane, we highlight the usecase for a mesh framework that operates on the principle of *Decoupled Enforcement - Coupled Placement*. We outline the design of a mesh control plane that can use the ADM abstraction to cleverly assign policies to ADMs so as to minimize the number of ADMs needed, which can lead to reduced overheads. While we envision that an ADM-based control plane can reduce overheads by minimizing the number of extra ADMs needed, the overheads of a single ADM can only be reduced by accelerating the dataplane. We call for a novel eBPF-accelerated dataplane proxy to function as ADM for this new control plane. While the use of eBPF for accelerated networking has been explored before [10, 11, 14, 15], mesh proxies are co-located on the same host as the service container and hence, the huge amount of intra-host network traffic serves as a perfect usecase to explore eBPF-based intra-host acceleration.

Section 2 provides a comprehensive background on microservices and service meshes. In Section 3, we demonstrate performance bottlenecks in current service meshes. Section 4 describes the ADM principle and outlines the design of an ADM-powered control plane. Section 5 describes the motivation to use eBPF as a tool to design an accelerated data plane.

2 BACKGROUND

2.1 Microservices

Conventionally, applications have been built and deployed as a single large *monolithic* executable. However, recently, applications are increasingly being built and deployed in the form of a *large number of small, loosely-coupled* services [13] - popularly referred to as *microservices* and typically deployed in separate isolated containers. In contrast to the earlier monolithic applications, the *microservice* architecture allows for increased development and deployment flexibility Each of the small microservices can be independently scaled and deployed as per their individual requirements. Further, separate teams can work on these microservices asynchronously leading to accelerated development cycles. This flexibility is crucial to deal with the ever-growing scale and demand of modern applications.

2.2 Application Layer Policies

With this growing trend of applications adopting the microservice architecture and becoming distributed in nature, application communication is becoming an intricate part of the application functionality [9]. Application communication involves enforcing policies related to various tasks including but not limited to service discovery, traffic load balancing, rate limiting, access control and authorization. For example, an access control policy could enforce that no request from service 'A' be allowed at service 'B'.

Many of these policies are enforced at the granularity of application requests. Even traffic between the same pair of microservices may experience different policies depending on the request. For example, a Key-Value Store (KVS) server may want to enforce different load balancing policies for RPC 'GetKey(key)' and RPC 'DeleteKey(key)' invoked by the same KVS client. Therefore, these policies can only be enforced at the application layer and existing methods of specifying and enforcing policies at switches or routers are unsuitable. Implementing these policies within the application code complicates application logic and imposes heavy burden on application developers to implement each desired policy.

2.3 Service Meshes

Service Meshes have emerged as a way of extracting communication policies outside of the application logic, running in a separate process (called the ‘sidecar’) alongside the application containers. Since microservice architectures are deployed as containers, naturally sidecar processes are also deployed in containers, with each service container getting a dedicated sidecar. All traffic to and from the service is then diverted via the sidecar container. Notably, the sidecars are transparent to the application. These sidecar containers can be thought to form a ‘mesh’ of service proxies for application communication, and hence the term ‘service mesh’.

These sidecars form the data plane of the service mesh, while a separate control plane is responsible for taking in application policies and configuring the sidecars accordingly. Current mesh control planes require the policies to be provided in the form of YAML configuration files. Several commercial service mesh frameworks follow this design [3, 6, 7]. The control plane accepts this YAML file as the application policy and appropriately configures the different sidecars in the deployed application.

3 SERVICE MESH OVERHEADS

3.1 Setup

Methodology: We run the microservice application under three settings - with Istio as the service mesh, with NGINX as the service mesh, and without any service mesh at all. We choose Istio as it is one of the most commonly used production mesh, based on Envoy [2], and as a comparison, we choose NGINX service mesh, based on the NGINX [5] proxy as a non-Envoy alternative. We run these experiments on a Cloudlab [12] machine, with four core Intel Xeon CPU at 2.0 GHz and 64GB RAM.
Workloads and Metrics: We use the Hotel-Reservation benchmark from the DeathStar suite [13]. We use the wrk2 [8] load generator provided with the benchmark to generate a mixed workload of different type of requests and evaluate three metrics: (i) CPU usage, (ii) Memory usage, and (iii) End-to-end application latencies.

3.2 End-to-end Latencies

Figure 2 shows the latency plots for the three settings. We observe that deployments using service mesh exhibit significantly higher end-to-end application latencies. In our experiments, we found that using Istio results in 2.84× and NGINX results in 1.54× higher median latency compared to the no mesh case.

We claim that the higher latencies are a direct consequence of the additional processing imposed by sidecars. Each request from a sender service to the receiving service requires (i) additional traversals of the kernel stack, (ii) additional memory copies from userspace to kernel space and vice-versa, and (iii) queueing can happen at the sidecars because of the processing times. The effect of the queueing can be seen on the tail latencies (99.9 and 99.99 %-ile latencies) where both Istio and NGINX result in more than 2× higher end-to-end latencies compared to the no mesh case. Our experiments are consistent with other studies on service meshes overheads [17].

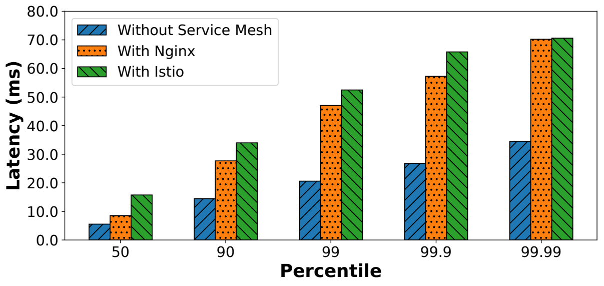
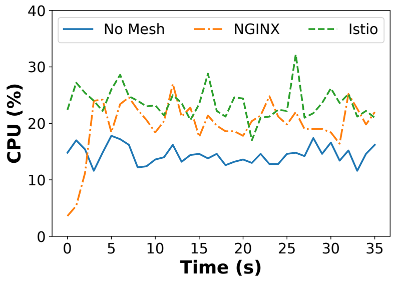


Figure 2: Overheads imposed on median and tail latencies because of current mesh frameworks



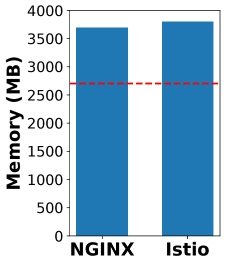


Figure 3: Comparison of CPU Utilization. **Figure 4: Memory Usage.**

The better performance of NGINX compared to Istio can be attributed to the fact that Istio uses Envoy, which provides much more features, compared to NGINX, making it bulkier.

3.3 CPU Usage

For each configuration (No Mesh, NGINX and Istio), we run the workload for 40 seconds and measure the CPU utilization every 1 second. Figure 3 shows the CPU usage for the three settings. We observe that Istio consumes 63% more CPU resources on average and NGINX service mesh consumes 37% more CPU resources on average, compared to the No Mesh case. This additional CPU usage is again because of the additional processing at the sidecars. These CPU processing overheads are likely to increase as the microservice application graph gets bigger (more services leading to more sidecar containers) and denser (more services invoked for each request).

3.4 Memory Consumption

Figure 4 shows the additional memory needed for deployments using service mesh, over the memory needed just for the application deployment without sidecar containers. The red dashed line shows the memory usage of the server when the application was deployed in the No Mesh case. We observe that deploying the application with Istio mesh leads to 41% more memory usage compared to the No Mesh case, while deploying with NGINX leads to 37% more memory usage.

4 APPLICATION DEFINED MIDDLEBOXES

An ADM is a container in the service mesh, that is responsible for executing one or more policies on the traffic to/from a particular service. An ADM is different from a sidecar in that not every service is required to have an ADM and not all traffic to/from a service is required to be passed through the ADM.

4.1 Coupled Placement

One of the primary reasons current service meshes followed the sidecar model was to make the operational cost of service meshes low. The traditional middlebox approach was to have a common shared middlebox to perform a specific functionality. However, using a common middlebox for multiple applications poses two challenges. Firstly, policies from different services may impact each other, complicating policy management. Secondly, shared middleboxes can lead to queueing and head-of-line blocking. Additionally, a common middlebox becomes a single point of failure. Hence, the alternative approach to couple a sidecar with each service container and letting the sidecar handle all the traffic to the service is preferred. We call this mode of deployment, where a sidecar is coupled with the service that it serves, *Coupled Placement*.

ADMs also follow the same principle. Therefore, each ADMs is coupled with a specific service, and handles only the traffic sent to/from the service. However, not all services may have an ADM, as we show next.

4.2 Decoupled Enforcement

The primary objective of service mesh is to enforce communication policies for microservice applications. We make the observation that most policies can be enforced *either* on the ingress of the receiver or the egress of the sender. For example, access control policies can be implemented either as admission control components at the ingress of the target service, or at the egress of *all* services that could potentially send traffic to the given service. Similarly, load balancing policies can be enforced either at the ingress of a specific service to balance load among its replicas or at the egress of *all* services that send traffic to this service. The only exceptions to the above observation are policies pertaining to external traffic or protocol-specific policies, e.g. setting the max timeout duration for a TCP connection can only be done at the connecting TCP endpoint. For external traffic, policies must be enforced at the ingress (egress) of the specific microservice that receives (sends) external traffic.

This observation drives the principle of *Decoupled Enforcement*. In our proposal, each policy is run in an ADM. For any given application policy, the control plane can *dynamically* choose to implement it at either at the ADM of the receiver, or at the ADM of the sender service. In contrast, existing mesh frameworks can only support static policy enforcement at the receiver or the sender, depending on the policy. For example, Figure 1c shows that decoupling the policies of `B` can allow us to implement Policy 3 at `A` instead. However, existing frameworks can only implement Policy 3 at the receiver and are hence, forced to have an additional sidecar at `B`.

4.3 A Framework to Reduce Overheads

We propose a clean-slate microservice communication framework that makes use of ADMs. Each policy must be implemented in one or more ADMs and each ADM must be associated with exactly one service. Therefore, a single ADM can only handle policies pertaining to the ingress or egress traffic of a particular service.

We can now formulate the placement of a policy in an ADM and the association of an ADM as an optimization problem. The objective of the optimization problem is to minimize the total number of ADMs, with the constraints being as follows:

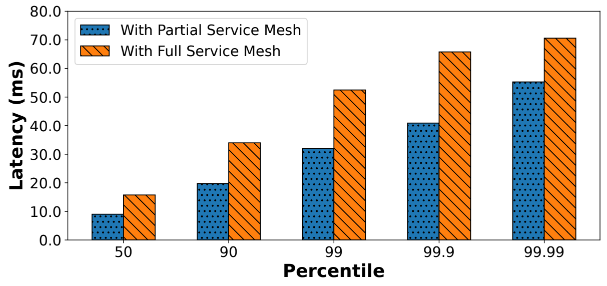


Figure 5: Impact of reducing ADMs on end-to-end latencies.

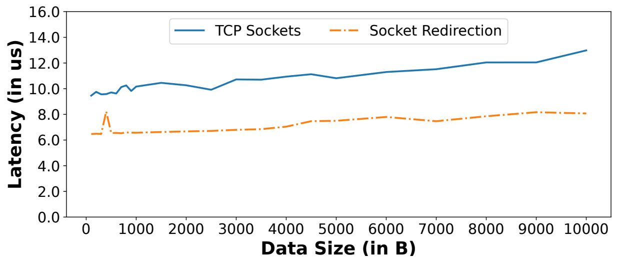


Figure 6: eBPF accelerated communication between ADM and service container

- An ADM must only handle policies pertaining to either the outgoing traffic or incoming traffic for exactly one service.
- A policy must either be placed on the ingress of the receiver ADM, or the egress of all sender ADMs.
- Policies pertaining to external traffic must be enforced on the ADM of the service receiving/sending the traffic.
- Policies that can only be implemented on the ingress or only on the egress, should be mentioned as correctness constraints.

By solving this optimization problem, the control plane can reduce the number of ADMs needed to satisfy a given set of application policies - thereby reducing the processing and resource overheads of the additional containers in existing frameworks.

We conduct a preliminary study to demonstrate the benefits of reducing the number of ADMs. We use the same testbed and setup as mentioned in Section 3. As a case study, consider a policy that aims to rate limit requests at database services. In our deployment, out of 18 total services, 9 were database services (using either memcached or mongodb instances). Figure 5 shows a comparison of end-to-end latencies for two configurations:

- Partial Mesh: Sidecars only at compute (non-database) services.
- Full Mesh (Default): Sidecars at all services.

We observe that consolidating policies at fewer sidecars and removing the redundant sidecars can lead to immense reduction in overheads. However, how to perform this reduction *automatically and correctly* given the user policies is an open challenge and the ADM principle is one step in this direction.

5 eBPF-ACCELERATED DATA PLANE

While the control plane mentioned earlier in Section 4.3 reduces overheads to some extent by reducing the number of mesh containers needed, there might still be several ADMs. Hence, we also explore and present preliminary ideas for an accelerated ADM that can further reduce the latency and CPU overheads.

As shown in Section 3.2, the additional network traversals and request processing can lead to higher application latencies. Since each ADM is coupled with its specific service, and both deployed on the same node, we can further optimize the communication between each service-ADM pair.

The `sockops` hook point provided by eBPF [1] opens up an interesting usecase here. It allows for a zero-copy redirection between two sockets inside the kernel without traversing the whole network stack. We maintain an eBPF map of active sockets, that gets updated at connection establishment and is later used for forwarding packets to the destination socket directly.

We implement a eBPF socket redirection module and compare the round-trip time of an echo server for messages up to 10KB in Figure 6. The takeaway is that using eBPF socket redirection provides good speedups. We observe a consistent 1.4-1.6× speedup against raw TCP sockets. This speedup is especially useful for service meshes, where all traffic between a service container and its corresponding ADM (if present) passes through the host network stack. Further, each user request can invoke several of these services and therefore, any speedup in intra-host networking will have cascading speedups on end-to-end latencies. However, full integration of the eBPF-accelerated data path warrants more research on multiple fronts. First, fault tolerance schemes are needed to decide what happens when either the service container or the ADM fails. Second, isolation mechanisms are needed when multiple policies are consolidated into a single ADM.

6 FUTURE WORK

In this paper, we highlighted the problem of performance bottlenecks and resource overheads in current mesh frameworks. We propose Application Defined Middleboxes as a novel logical framework to think of service meshes, and highlight how the principle of Decoupled Enforcement, Coupled Placement can enable control planes to optimally place policies in ADMs so as to reduce the number of userspace containers.

The future work is to develop and implement such a control plane. Crucially, the control plane needs semantic information about various policies (whether a policy is a protocol-specific policy or not, whether a policy can be implemented at ingress and egress both, etc) to optimize placement. The current YAML file-based approach for specifying policies may not be ideal, necessitating a redesign of the control plane interface. Secondly, the above discussion is mostly focused on RPC-style communication, which has been carrying the majority (upto 95%) of traffic in modern datacenters [16]. However, for other inter-service communications - for example, event-based messaging, the constraints on policy implementation must be revisited. Depending on the specific inter-service communication protocol, some policies may be suitable for the Decoupled Enforcement paradigm while other policies will require constraints on their implementation that can be used as mentioned in Section 4.3. Lastly, our proposed control plane may open more avenues to optimize resource management - the placement of ADMs can lead to load imbalance, hence, along with the correctness constraints, and performance objectives in Section 4.3, the resource usage of ADMs must also be taken into account.

Finally, simple sidecars are not enough to be used as ADMs and to reduce the overheads of the interaction of service containers and ADMs, an accelerated dataplane must be designed. We motivate the usecase for eBPF hooks for service meshes and list challenges on isolation and fault tolerance for future work.

REFERENCES

[1] eBPF. https://ebpf.io/.
[2] Envoy Proxy. https://www.envoyproxy.io/.
[3] Istio. https://istio.io/.
[4] Linkerd. https://linkerd.io/.
[5] NGINX Load Balancer. https://www.nginx.com/.
[6] Cillium Service Mesh. https://docs.cilium.io/en/v1.13/network/servicemesh/index.html, 2023.
[7] NGINX Service Mesh. https://www.nginx.com/products/nginx-service-mesh, 2023.
[8] wrk2. https://github.com/giltene/wrk2, 2023.
[9] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. Leveraging service meshes as a new network layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, pages 229–236, New York, NY, USA, 2021. Association for Computing Machinery.
[10] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.
[11] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing quic. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 59–74, New York, NY, USA, 2019. Association for Computing Machinery.
[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 3–18, New York, NY, USA, 2019. Association for Computing Machinery.
[14] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.
[15] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 780–794, New York, NY, USA, 2022. Association for Computing Machinery.
[16] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: Admission control for performance-critical rpcs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 1–18, New York, NY, USA, 2022. Association for Computing Machinery.
[17] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting service mesh overheads, 2022.