UKFaaS: Lightweight, High-Performance and Secure FaaS Communication With Unikernel

Zhenqian Chen, Yuchun Zhan, Peng Hu, Xinkui Zhao, Muyu Yang, Siwei Tan, Lufei Zhang, Liqiang Lu, Jianwei Yin, *Senior Member, IEEE*, and Zuoning Chen

***Abstract*—Unikernel is a promising runtime for serverless computing with its lightweight and isolated architecture. It offers a secure and efficient environment for applications. However, famous serverless frameworks like Knative have introduced heavyweight component sidecars to assist function instance deployment in a non-intrusive manner. But the sidecar not only hinders the throughput of unikernel function services but also consumes excessive memory resources. Moreover, the intricate network communication pathways among various services pose significant challenges for deploying unikernels in production serverless environments. Although shared-memory based communication on the same server can solve the communication bottleneck of unikernel-based function instances. The situation where malicious programs on the server make the shared memory untrustworthy limits the deployment of such technologies. We propose UKFaaS, a lightweight and high-performance serverless framework. UKFaaS leverages the advantages of customized operating systems through unikernel and it non-intrusively integrates sidecar functionality into the unikernel, avoiding the overhead of sidecar request forwarding. Additionally, UKFaaS innovatively implements data communication between unikernels in the same server to eliminate VM-Exit bottlenecks in RPC (remote process call) based on VMFUNC without relying on memory sharing. The preliminary experimental results indicate that UKFaaS can realize 1.8×-3.5× request throughput per second (RPS) compared with the advanced serverless system FaasFlow, UaaF and Nightcore in the Google online boutique microservice benchmark.**

***Index Terms*—Unikernel, serverless computing, library operation system, virtualization.**

Received 23 April 2024; revised 8 May 2025; accepted 26 June 2025. Date of publication 4 July 2025; date of current version 10 September 2025. This work was supported in part by the National Science Foundation of China under Grant 62472375, in part by the Major Program of National Natural Science Foundation of Zhejiang under Grant LD24F020014 and Grant LD25F020002, in part by Zhejiang Pioneer (Jianbing) Project under Grant 2024C01032, and in part by Ningbo Yongjiang Talent Programme under Grant 2023A-198-G. Recommended for acceptance by R. Canal. *(Zhenqian Chen and Yuchun Zhan contributed equally to thiswork.) (Corresponding author: Xinkui Zhao.)*

Zhenqian Chen, Liqiang Lu, and Jianwei Yin are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China.

Yuchun Zhan, Peng Hu, Xinkui Zhao, Muyu Yang, and Siwei Tan are with the School of Software Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China (e-mail: zhaoxinkui@zju.edu.cn).

Lufei Zhang and Zuoning Chen are with the State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China.

Digital Object Identifier 10.1109/TC.2025.3586031

TABLE I
RUNTIME IN SERVERLESS. L: LOW. M: MIDDLE. H: HIGH. RPS: REQUEST THROUGHOUT PER SECOND WITH WRK [13]

	VMs	Container	Wasm	Unikernel
Isolation level (↑) [14]	H	L	M	H
Startup latency (↓) [14]	H	M	L	L
Memory footprint (↓) [2]	H	M	L	L
RPS of Nginx ($\times 10^4$) (↑)	1.8	6.1	3.1	12.2

I. INTRODUCTION

SERVERLESS computing enables users to concentrate exclusively on their application logic while entrusting the management of underlying operating systems and hardware infrastructure to cloud service providers. By adopting Function-as-a-Service (FaaS), users incur costs solely for request processing and function container design, eliminating the need for continuous resource rental if low startup latency and high concurrency of service can be satisfied.

Unfortunately, although container technology guarantees lightweight, its cgroup isolation cannot meet the needs of security scenarios. As shown in Table I is the comparison between different runtimes. The basis of low (L), middle (M), and high (H) comes from the reference of the corresponding item. VM here is a complete virtual machine compared with the lightweight and specified runtime unikernel. The existence of nested virtualization supports the deployment of unikernels on VMs. Secure container technology, as a compromise solution that balances flexibility and security, such as Firecracker [1] run containers in their sandbox VM. However, due to the intervention of VMs, the startup speed of containers has decreased compared to traditional containers. WebAssembly [2] is a promising runtime due to its small size and ease of deployment. Nevertheless, poor system call support and transmission performance limit its application in I/O frequency scenarios. In addition, modern data centers have the requirements for the security of deployment [3], [4] for shared-memory-based communication. This means that the work that relies on shared memory security to achieve accelerated communication is not applicable.

As a new-generation VM technology, unikernel[1] retains only the necessary kernel modules for applications to avoid unnecessary instruction cycles and state-switching overhead, avoiding the drawback of traditional VM. In addition, it is suitable for lightweight, high-performance, and secure applications

[1]In this work, we use Unikraft [5] as the unikernel by default due to its community influence and ecosystem.

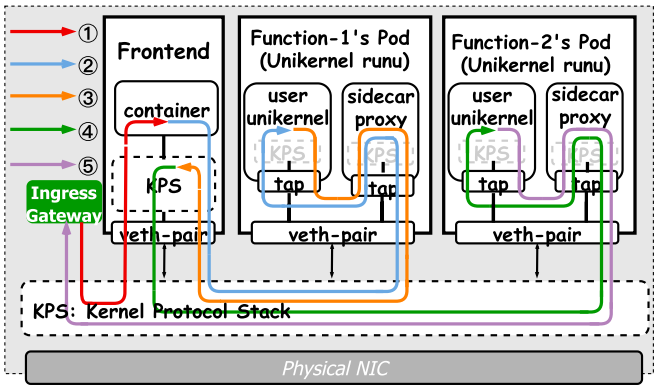


Fig. 1. Networking processing in a typical serverless function chain setup.

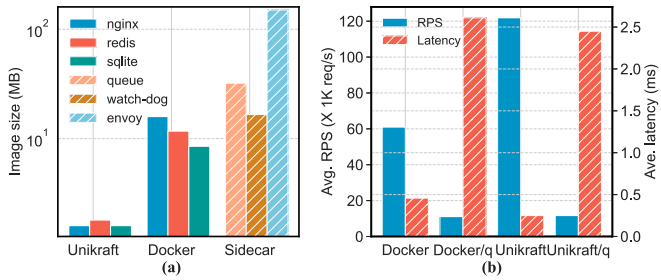


Fig. 2. Performance and overhead caused by sidecar proxy. (a) Image size of representative unikernel and docker application and sidecar image. (b) Average request per second (RRS) and latency of nginx unikernel and docker from wrk [13] (14 threads, 30 connections, 60s). q/ means application with queue-proxy sidecar.

in the serverless scenario. In Table I, we show the advanced RPS performance of unikernel Nginx compared with different runtimes (Wasm uses the server that comes with Javascript). Related work has been carried out to optimize unikernel's communication [6], [7] and reduce startup latency [8] in the serverless scenario, as well as to explore memory deduplication [9] and snapshot startup optimization [10] to improve concurrency. Unikernel has the advantages of being lightweight, having good performance and strong security. Its performance and lightweight nature determine that it is suitable for promotion to FaaS scenarios [11], while the feature of strong isolation makes it suitable for promotion to multi-tenant scenarios where the shared memory is untrusted [3], [12]. However, the deployment of unikernel to the FaaS situation presents the following challenges.

Incompatible at runtime for the sidecar. Current serverless framework optimization work in research still has a large gap to the industrial scene [15]. In Kubernetes (K8s), a service instance is deployed by Pod to manage containers/VMs. Each Pod instance in a serverless application consists of a function container and a sidecar container as shown in Fig. 1. For example, in Knative [16][2], the queue-proxy sidecar container enhances the integration of the function container with the service mesh. It facilitates non-invasive request forwarding, metric collection, and lifecycle management. In K8s, each service instance is maintained by a Pod that consists of a function container and a sidecar container. This unique setup poses compatibility issues on unikernel since the same runtime must be implemented using the CRI (Container Runtime Interface). However, this approach is incompatible with existing container runtimes like runc. Implementing a dedicated sidecar based on unikernel would create significant engineering pressure and sidecarless design would be a potential solution for unikernels.

Heavy-weight serverless components. The existence of sidecar leads to 10.5× RPS overhead in unikernel due to the extra data replication, context switching [17] and VM-Exit, as shown in Fig. 2(b). Furthermore, the Knative sidecar queue-proxy has 20.1× higher image size consumption compared to the nginx unikernel demonstrated in Fig. 2(a). This limitation hampers the application of cold start optimization techniques for unikernel/VM runtime in practical deployments.

The current sidecarless solutions, SPRIGHT [17] utilizes eBPF [18] to replace the functionality of sidecar components to avoid the context switching overhead during kernel function invocation. mRPC [19] and ServiceRouter [20] provides the public sidecar for the applications in the same host instead of the one-to-one design for application and sidecar. However, there are still doubts about the compatibility of these works applied to unikernel. SURE [7] applies the lib-sidecar design in unikernel. Unikernel cannot leverage the benefits of eBPF due to its single-address OS feature and the extra engineering preparation for eBPF. Meanwhile, the shared sidecar or the lib-sidecar design in SURE relies on the shared memory for communication. However, virtual machine monitors (VMM) like Firecracker [1] applied in FaaS scenarios do not support the ivshmem device. In addition, when the shared memory [3], [4] is untrusted, these communication methods between service and sidecar are not available.

Data transmission overhead in function chaining. In modern cloud-native systems, component design is decoupled. Therefore, HTTP/REST is adopted to ensure communication flexibility in the service chain. However, this type of network communication incurs significant overhead. For example, in Knative, function fn-1 invokes fn-2 requires additional round trips through the sidecars and frontend as shown in Fig. 1. To address the issue of data plane invocation, several works dynamically adapt communication mode [21], [22], [23], design the runtime framework to accelerate functions coordination [2], [24] and optimize the communication path [7], [17], [23] with the shared memory to reduce the overhead of data transmission.

However, using shared memory should take the VMM into consideration for VM and whether the shared memory is available. Unfortunately, the previous work for communication optimization for unikernel are based on shared memory [7] or even do not take data transmission into consideration [6]. We hope to design a host insensitive communication protocol without relying on memory sharing and VM-Exit, where VMFUNC has potential to apply to satisfy this need.

We design the communication framework UKFaaS. It addresses the incompatibility issue of unikernel deployment under

[2]Knative is a representative serverless framework compatible with K8s. The scenario discussed in this paper adopts Knative sidecar architecture by default.

K8s by replacing the design of co-deployment with sidecar containers with lib-sidecar, and avoids the overhead caused by the complex network stack of sidecar. Our communication framework supports operation in situations where shared memory is untrusted. By introducing VMFUNC-based transmission, we avoid the high overhead of FaaS instance data transfer caused by VM-Exit. Meanwhile, its reliability in concurrent scenarios is guaranteed through the design of concurrent management and transparent data transmission protocols. Our contributions can be summarized as follows.

- We implement lib-sidecar, which supports the basic functions of sidecar and offers transparent selection of network and VMFUNC transport protocols. It eliminates the overhead of sidecar network forwarding.
- We implement lifecycle management and concurrent transmission management based on the VMFUNC transmission protocol. It does not rely on shared memory and avoids VM-Exit. It has significant advantages in the transmission of small data scales and CPU cycle savings ranging from $3.3\times$-$18.9\times$ have been achieved.
- We evaluate our system in the Google online boutique scenario [25] benchmark, where UKFaaS gets $1.8\times$-$3.5\times$ RPS improvement compared with the advanced serverless framework Nightcore [24], UaaF [6] and FaaSFlow [21].

II. Background and Motivation

A. EPTP Switching and VMFUNC

VMFUNC [26] is a hardware extension used by Intel for virtualization, which allows programs in VMX non-root mode to call VM functions in the hypervisor. Extended page table pointer (EPTP) switching enables VMs to switch the address space of calling programs to a specified address in the EPTP list while avoiding high system overhead caused by VM-Exit. Specifically, the page table of the guest VM is mapped from the guest virtual address (GVA) to the guest's physical address (GPA), and then the extended page table (EPT) converts GPA into the host physical address (HPA), achieving direct mapping from GVA to HPA. The EPTP is the head address of this EPT and $eptp = eptp\_list[eptp\_index]$, where the $eptp\_index \in [0, 511]$. VM can access data located at another VM memory address by VMFUNC with a specified EPTP index but cannot access its own (caller's) memory or data during VMFUNC.

Previous work [27] did not consider data transfer issues under VMFUNC mainly because of the simplicity of implementing memory sharing for these works due to its rootkernel for the EPT management [27]. Besides, this kind of work needs to externally consider the security and concurrency control security and concurrency control of shared memory. However, when the shared memory is not available or untrusted, this method cannot be applied.

B. Unikernels

Unikernels [5], [28], [29], [30] are built using a library operating system, where the OS kernel and user applications are coupled with the necessary system libraries. This design significantly reduces memory usage and attack surface, ensures system security, and reduces application startup time. As a representative unikernel, Unikraft [5] implements most major Linux system calls based on the POSIX standard and supports other high-level programming languages. Besides, it allows system functional modules to be added to Unikraft in a configured manner through an external library, which makes it possible to decouple modules required by serverless frameworks, such as sidecars with third-party libraries to the unikernel.

Other kinds of unikernel like UKL [30] and Lupine [29] are more like a process instead of a lightweight VM and they do not perform better than traditional unikernel due to their macrokernel-unikernel combined architecture. Besides, the distributed library OS like EbbRT [28] is used for elastic scaling in the cloud environment instead of a runtime. It also performs worse than single-core unikernel due to the IPC cost between libraries.

C. Unikernel for Serverless

The current optimization work of unikernels in the serverless scenario [11] mainly revolves around cold start [9], [10] memory saving and communication acceleration. UaaF [6] considers both cold start acceleration and inter-instance communication acceleration. It proposes a session-function (S-F) architecture, where the function unikernel is specifically designed to execute functions when the session remotely calls the function unikernel using VMFUNC. Users can combine multiple function unikernels through the session and VMFUNC to achieve the deployment of the target function chain. However, UaaF does not take data transmission into consideration since the HPA of the caller is not available after EPT switching to the callee. SURE [7] also applies the lib-sidecar design in unikernel and utilizes the shared memory for communication between unikernels in the same host. But in the case where the shared memory is untrusted or VMM does not support ivshmem [1], it is unavailable.

D. Bottleneck by the Sidecar

The issues arising from the sidecar are primarily related to the high image size it requires and the additional overhead of its gateway [15].

Image size bottleneck. As shown in Fig. 2(a), taking Nginx, Redis, and SQLite images as examples. Under the default configuration, the size of Unikraft's images is less than 2MB while Docker's image sizes (minimized from the docker hub) are around 10MB level. The image size of the unikernel is at around 1MB for some representative applications and is much smaller than container image sizes. Although containers share the host kernel, the image still needs to contain user-space tools and libraries. It cannot completely eliminate kernel-related dependencies. Image size of sidecar containers are much larger than unikernel, limiting startup speed and concurrency in serverless scenarios.

Communication bottleneck. As shown in Fig. 1, the mainstream serverless communication path requires multiple sidecar request forwarding to return requests to the requesting front end. At the network stack level, the situation of unikernel is similar

to that of containers. The existence of sidecar causes about 10 additional data copies, context switches, interrupts, and network stack processing tasks [17]. We take Knative's queue-proxy as an example to compare the stress test on Unikraft-implemented nginx with or without queue-proxy. Docker/q represents requests forwarded through queue-proxy reaching nginx, and similarly for Unikraft. We use wrk for stress testing and default allocate 1 CPU and 64MB memory per instance. The results are shown in Fig. 2(b).

Without the queue-proxy, Unikraft achieves a 2× higher Nginx throughput compared to Docker. However, with the introduction of the queue-proxy, all requests necessitate additional data copies, context switches, and interrupts. Moreover, entering and exiting the VM incurs an extra cost in virtio-io [31], resulting in thousands of additional system call cycles overhead. Consequently, this leads to a 6× and 10× decrease in throughput for Docker/q and Unikraft/q. Despite the potential for VMFUNC to provide high-performance data transmission, sidecars significantly diminish the benefits offered by serverless computing. Therefore, a more lightweight solution like a customized unikernel with a sidecar is imperative.

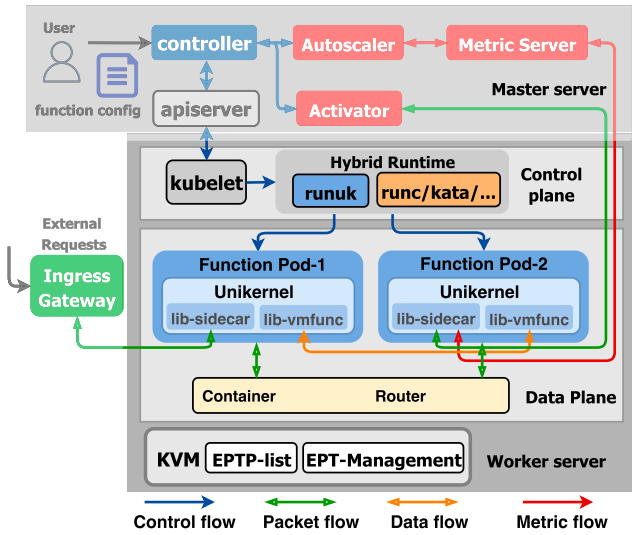


Fig. 3. Overall architecture of UKFaaS. The part of the color with a partially transparent white overlay is the part of the knative and K8s itself.

III. System Overview

Due to the lower communication efficiency of cross-server services compared to same-server services, the most advanced serverless frameworks tend to deploy services from the same invocation chain on the same server [17], [21], [32], [33]. The scheduling algorithm is independent of the runtime and is compatible with UKFaaS. Therefore, UKFaaS mainly focuses on optimizing the communication performance between unikernels on a single server.

The UKFaaS system revolves around the following targets:

- Replace sidecar with lib-sidecar. This design avoids the complex network stack of sidecar, simultaneously solves the sidecar compatibility problem when unikernel is deployed in the actual production environment, and also takes advantage of the customized feature of unikernel. Under the premise of meeting the basic functions of sidecar, lib-sidecar can transparently switch the transfer protocol according to different data scales and service placements, achieving efficient data transmission without relying on shared memory.
- Accelerate the communication between unikernels. We use the transport protocol based on VMFUNC instead of the network to avoid VM-Exit. It also supports service lifecycle management, service discovery, data transmission based on VMFUNC.
- Reliable and secure deployment. We further considered the reliability of VMFUNC transmission in concurrent scenarios. And the service security of the transmission protocol and the lib-sidecar cases are further considered.

In Fig. 3, the overall architecture of UKFaaS is depicted. The master server serves as the core component of Knative [16] and is responsible for interacting with the data plane on the worker for request forwarding (green lines) and metric collecting for scaling (red lines). Runuk on the control plane manages the lifecycle of pod instances on the data plane with EPTP management in the KVM module and cooperates with the router to achieve service discovery based on VMFUNC communication. The implementation details are presented in §V-A.

On the data plane, We list the running unikernel function instances. The green line is mainly responsible for handling the requests of network packets by the lib-sidecar. These requests include those from gateway and registered with the router after startup. Besides, the big data requests that are not suitable to transmit through VMFUNC and the cached data coming from activator (§IV). The orange line lib-vmfunc implements efficient short message passing bypassing local memory, where the packet sending is about 3.3×-18.9× faster compared with unikernel network (§VI-C). We achieve efficiency improvement through point-to-point transmission of functions to functions, bypassing redundant components [17], [23]. The communication process and concurrency control are demonstrated in §V-B and V-C. Finally, we discuss the security concerns in the context of accessing the system based on the VMFUNC architecture (§V-D).

Threat Model. We guarantee the trustworthiness of the host OS and hypervisor, as well as the correct deployment of unikernel and the execution of VMFUNC instructions. We conduct binary checks and control flow integrity checks on the unikernels deployed on the server [27], [34]. We assume that the shared memory on the host and the unikernel service registered by the user is untrusted. Malicious services can cause incorrect booting of components in the Knative system by hijacking sensitive data in the sidecar, such as statistical indicators. Furthermore, other programs deployed on the server make the shared memory untrusted because they can access it. SURE [7] only restricts the division of shared memory when accessing between unikernels and does not discuss the behavior of other programs on the host.

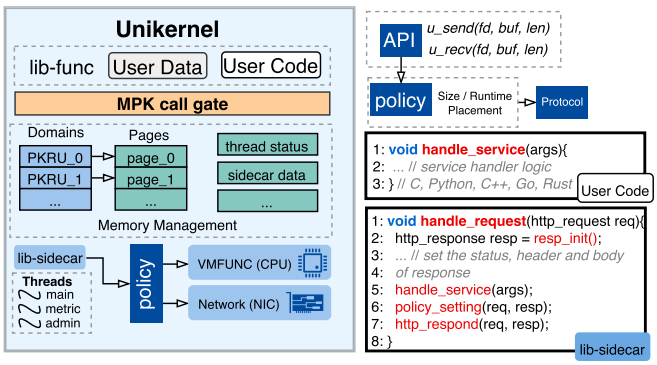


Fig. 4. Architecture and communication process of lib-sidecar.

IV. Design of Lib-Sidecar

The sidecar is an HTTP server. The queue-proxy component of Knative's sidecar implemented by Go, mainly starts three coroutines: main server, metric server, and admin server to manage request forwarding, metric monitoring, and lifecycle management respectively, as well as a logging output coroutine. We use Unikraft's lib-musl and lib-lwip to implement the lib-server, and lib-sidecar is the extension of lib-server. We used three threads to implement the same processing logic of three HTTP servers in the queue-proxy of Knative. Among them, only corresponding ports need to be exposed for metric monitoring and lifecycle management. The main, metric and admin threads in Fig. 4 are all implemented in lib-server.

Sidecar function. Pod in Knative receives requests from the gateway through port 8012 exposed by queue-proxy and forwards them to the port exposed by the user-container. In lib-sidecar, we directly register the handler function. As shown in Fig. 4, `handle_request` is responsible for processing HTTP requests and encapsulating the user-defined function `handle_service`. Before being processed in the main server, `handle_request` is subject to various checks such as health status check, processing rate statistics, and message header file inspection like queue-proxy.

Users need to implement `handle_request` to process requests from the gateway when the `handle_service` defines the data processing functions. After the function definition in Fig. 4, lib-sidecar's main function starts these three servers. The startup process is completed during the boot phase.

Function expansion. In the sidecar, there are additional requirements for observability. Regarding observability at the state level, users can extend the metrics to be observed, such as CPU, memory, disk usage, etc., into the metric server. For functionality modules that are not part of the three mentioned threads, custom functionalities can be introduced into lib-sidecar to further expand its capabilities. The function of lib-sidecar is scalable. However, compared with the decoupled design of sidecars like mRPC [19] and ServiceRouter [20], our lib-sidecar is tightly coupled with the function runtime. Although there are already dynamic compilation technologies for OS [35], it is uncertain whether they can be widely applied to unikernel because of its single address design. Therefore, we need to restart to update.

Transparent protocol. Our VMFUNC transmission is based on registers and has significant advantages over networks in the transmission of short messages. However, when the data scale increases, the efficiency is lower than the network (discussed in §VI-C), and cross-server transmission is not supported by VMFUNC. Therefore, we transparently implement register-based VMFUNC transmission and network transmission in lib-sidecar with policy to achieve adaptive adjustment of different protocols.

For users, only the `u_send` and `u_recv` APIs need to be uniformly used to achieve the sending and receiving of data. As shown in Fig. 4, we implement the reception logic before the invocation `handle_request`, where the `u_recv` is applied for receiving the request outside. `u_send` is encapsulated in `http_respond`. We allow users to write policies in lib-sidecar. The policy allows us to determine the policy based on the size of the data, the runtime objects of communication, and whether the services are placed on the same server. For example, we default to using VMFUNC-based transmission for data smaller than 64KB. For larger data, non-unikernel instances and cross-server transmissions, we use network transmission. The design of this policy also supports us in expanding more transmission protocols.

Isolation for lib-sidecar. Since unikernel is an operating system with a single address space, this means that in order to protect the memory safety of calls between system libraries, we need to design an isolation mechanism to prevent untrusted third-party services from hijacking the data of lib-sidecar, resulting in malicious scaling, etc. The mechanism of security isolation has been discussed in FlexOS [36] and SURE [7], and it needs to be simplified in our scenario.

As shown in Fig. 4, the part defined by user functions is regarded as an untrusted third-party component and placed in a separate lib-func. This library is placed in a separate stack. Cross-domain function calls must go through a MPK gate mechanism similar to ERIM [34] to achieve stack switching. Cross-domain function calls only allow access to shared variables in the specified heap address space. These shared variables require specially allocated via specified API `share_alloc` to allow access to all isolated domains. Other variables are stored by default in the heap where the library is located. It effectively prevents the functions of the user located in lib-func from tampering with the metrics or private data from lib-sidecar.

Multi-language support. Currently, Unikraft supports Python, C++, Go, and Rust. With the lib-sidecar, users only need to use the corresponding language to define the processing logic in the `handle_service`. We implement a benchmark for the UKFaaS framework based on Python in §VI-F and further analyze the overhead of using Python.

V. VMFUNC Communication Protocal

In this section, we will introduce how to manage the lifecycle and data transmission of unikernels in the scenario of VMFUNC communication. The communication is essentially based

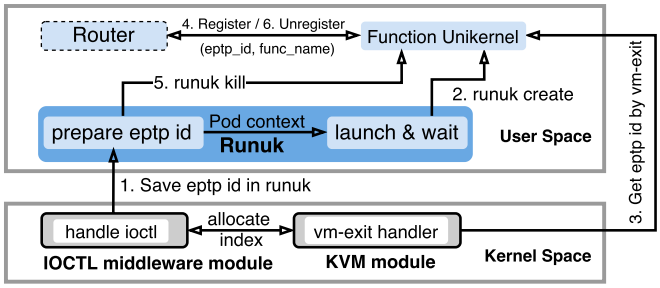


Fig. 5. Lifecycle of function unikernel registration managed by runuk.

on EPTP index management of the unikernel. First, we explain how runuk and registration centers synchronize and maintain EPTP index creation to destruction to achieve automated service discovery (§V-A). Second, we discuss how we achieve data transfer with VMFUNC (§V-B) and its concurrent request handling (§V-C). Finally, we discuss some security details on the process of communication (§V-D).

A. Lifecycle Management by Runuk

In Fig. 5, it is the lifecycle of a unikernel managed by runuk with four main components. Among them, runuk and router are running in user space, while KVM and IOCTL (input and output control) middleware modules are components running in kernel mode.

Runuk. It is a CRI (Container Runtime Interface) of Kubelet implemented by Go to manage the lifecycle of a unikernel. Unlike Runu [37], since we need to implement communication based on VMFUNC, additional interaction with the kernel is also required to obtain communication information. In the kernel mode, it mainly contains the kernel-mode IOCTL middleware module and the KVM module as shown in Fig. 5. The IOCTL middleware module maintains a character device, through which the runuk in user space can obtain the EPTP index of the currently created VM. IOCTL is a Linux system call to manage the I/O pipeline. For the KVM module, *vmcs* (VM control structure) is used to save the state and configuration of a VM. *vcpu_vmx* (virtual CPU VM eXtensions) is an extension structure managed by KVM software to supplement the deficiencies of *vmcs*. During the creation process of the unikernel, the data structure *vmcs* encapsulated by *vcpu_vmx* is created for the unikernel. Unikraft is a single-core VM that corresponds to each *vcpu_vmx* one-to-one. We add two additional fields, *eptp_list* and *eptp_index* to *vcpu_vmx* so the unikernel itself can get their EPTP information in *vmcs* through VM-Exit.

Router. Each server will deploy a router to maintain the routing information of unikernels. It is used to maintain the EPTP information transmitted by the service name and IP as well as VMFUNC. The maintenance includes the service registration and destruction.

Registration. As shown in Fig. 5, the startup process is implemented by runuk create **(step 1 and 2)**, where a unikernel Pod initializes its network, namespace and other context information. When a new unikernel instance starts up, it will send information such as the EPTP index got from VM-Exit **(step 3)** and service name to the router **(step 4)**. The router will broadcast this routing information to instances on the same server and routers across servers, thereby ensuring the correctness of the global routing information.

Destruction. Serverless instances have a default save time and will be destroyed after being inaccessible for a period of time. Before the unikernel instance enters the destruction stage **(step 5)**, lib-sidecar will send a notification to the router to clean up this route **(step 6)**. The router broadcasts to other members like a registration to update their routing cache.

Both the creation and destruction phases require maintaining the information of the EPTP index in KVM. There are the following two problems to be solved. Firstly, the EPTP list in all unikernels must be synchronized to ensure the same $eptp = eptp\_list[eptp\_index]$. We maintain a data structure *geptp_list* for all unikernels in KVM, which consists of two member variables *global_eptp_list* and *spinlock*. *global_eptp_list*[512] is the unified EPTP list for all unikernels when the *spinlock* ensures the consistency if multiple functions are registered simultaneously.

Secondly, the client runuk needs to access the EPTP index associated with each unikernel. During creation, a unikernel can retrieve its own EPTP index through a VM-Exit and send the EPTP index and function name to the router. However, the challenge arises to update the route mapping when a unikernel is destroyed since the unikernel does not support this signal notification mechanism like containers when killed.

To handle the second problem, runuk sends an unregister request to the router before killing as shown in Fig. 5. Specifically, we need to send the EPTP index of the unikernel that is about to be destroyed to the router. However, since runuk runs in user space, it cannot directly access the EPTP index from KVM in kernel mode. Therefore, we introduce an IOCTL middleware module in the kernel module to bridge the access between runuk and KVM with a character device. The IOCTL calls in KVM invoke the *allocate_index* to get the EPTP of the created unikernel and send it to runuk.

B. Data Transmission With VMFUNC

EPTP Switching (VMFUNC) can achieve switching without VM-Exit, but the context information (data in memory) retained in the VM caller cannot be transferred to the callee directly. After EPTP switching, although EPT is different, data can be saved in registers. Therefore, we propose a register-based VMFUNC transmission protocol to accelerate data transfer. This process happens when callee exists in the routing cache of lib-sidecar and VMFUNC data transmission is used according to the policy setting. Otherwise, the data will be sent to the activator and the subsequent steps are the same as those in Knative.

In Fig. 6, the transmission roles are caller and callee. During transmission, the buffer relies on the `handle_msg`(*struct msg *msg_t, void *msg_r, int src, int dst*) function written in assembly code including four parameters. The first parameter *msg_t* is the data we want to transmit. The second parameter *msg_r* is the data fixed in the same address space by trampoline code during the startup phase of VMFUNC. The third and fourth parameters

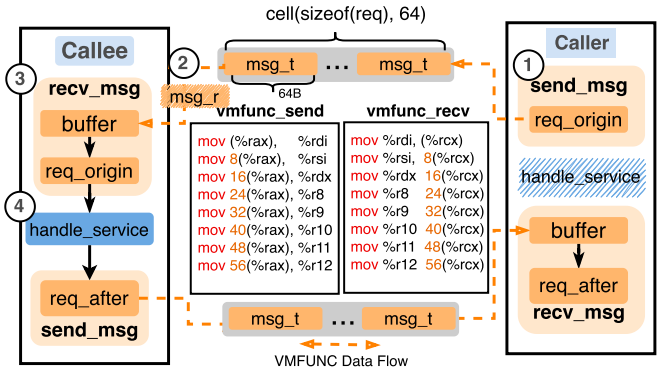


Fig. 6. Data transmission with VMFUNC.

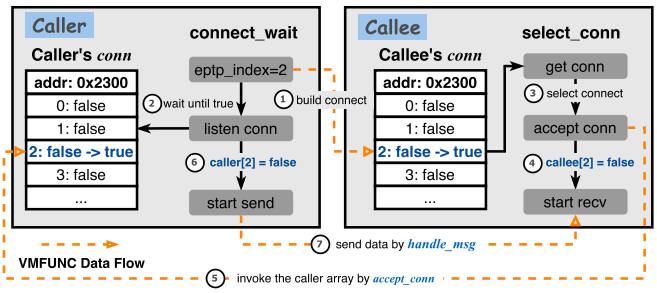


Fig. 7. Concurrency management of VMFUNC data transmission.

are respectively *eptp_index* of caller and callee. The steps are as follows in Fig. 6.

① **`vmfunc_send`.** When executing VMFUNC, it is necessary to save the context of the registers, and four parameters of `handle_msg` in registers (stored in rax, r15-r13). The remaining 8 idle registers as shown in Fig. 6 are used to temporarily store the *msg_t*. For a 64-bit unikernel, one register can transmit 8B and all 8 registers can transmit data up to the size of a *msg_t* (64B) at a time. We save the address of the first parameter *msg_t* passed in `handle_msg` to register rax. In `vmfunc_send`, we sequentially put the 64B data from the corresponding address in register rax into 8 registers.

② **`vmfunc_recv`.** After completing the storage in the register, we execute VMFUNC to jump to the callee. First, we assign the address of *msg_r* stored in register r15 to register rcx. Subsequent assembly code in `handle_msg` executes `vmfunc_recv` and moves 64B data stored in 8 registers one by one, assigning them to the address of *msg_r* sequentially.

③ **Loop until transmission ends.** Finally, callee executes VMFUNC and jumps back to the caller's VM, completing a data transfer operation `handle_msg`. The actual transmission size of the data is generally greater than 64B. Therefore, after each completion of `handle_msg`, we will offset the first address of the passed *msg_t* by 64. The number of executions is `cell`(*sizeof(req), 64*). The *msg_t* maintains a signal that informs the receiver that the data transmission has finished.

④ **Callee executes and returns the result.** The receiving callee converts the buffer consisting of *msg_r* into the corresponding data structure and executes the processing logic of `handle_service` (§IV) after getting the data from message queue (§V-C). After completing the processing, callee performs a `send_msg` to the caller again. Similarly, the caller performs a `recv_msg` operation to receive and process the function that has been handled. For users, it is like experiencing a remote call and obtaining the processed result.

Multi-tenant deployment. In a multi-tenant scenario, CPU switching occurs when the number of unikernel deployments exceeds the number of server cores. We do not set a dedicated CPU binding for unikernel. Because this will limit the number of unikernels deployed. Therefore, there exists a case where the vCPU switching of unikernel occurs in the data transfer based on registers. The data transmission was still correct when the switch occurred. Because when the vCPU is switched out, the CPU automatically saves all register states to the *vmcs*. When switching back, the CPU directly restores the state from the *vmcs* and continues to execute. During this process, there will be a pause in data transmission. However, the entire transmission process is carried out by the caller, and callee does not participate in the synchronization. Therefore, the restoration of the state can still ensure the correctness of the transmission.

Limitations. The transfer mode based on registers can circumvent VM-Exit. There are significant benefits for short messages such as message notifications between microservices. However, when the data scale increases, the efficiency will also decrease due to the increase in the number of migrations, and the network transmission performance is better (§VI-C). Meanwhile, this mechanism can only be applied between VMs of the same server, because VMFUNC can only be applied in non-root mode and cannot be applied across servers. Based on the dynamic adjustment mechanism of the policy mechanism of lib-sidecar, we use this scheme for short message passing in the unikernel of the same server.

C. Concurrency Management

In this section, we first explain how to ensure concurrent management of concurrent VMFUNC data transfers initiated by multiple caller to callee. Next, we will analyze the management of message queues under the combination of multiple transmission protocols and other situations that need to be considered in data transmission.

1) VMFUNC Connection Management: When using `handle_msg` for transmission (§II-A), it is necessary to consider the issue of concurrency control. When a callee is called by multiple callers simultaneously without concurrency control, the callee's buffer will be written by multiple callers with data confusion. We implement a similar epoll mechanism to manage multiple calls to callee as shown in Fig. 7.

Caller. Before establishing a connection, the caller executes `connect_wait`*(caller_eptp_index, callee_eptp_index)* to join this connection. We maintain a *conn* array with a length of 512 in both the caller and callee, where the address of *conn* is fixed through the trampoline code. When the caller initiates a connection establishment, it sends a VMFUNC call

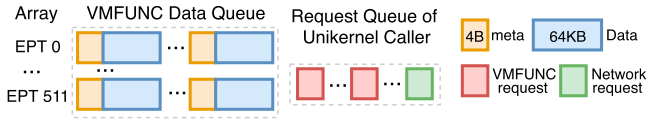


Fig. 8. Message queue for VMFUNC and network connection establishment.

to the callee (①). This VMFUNC operation will cause callee's $conn[caller\_eptp\_index] = 1$. When multiple callers access simultaneously, values corresponding to multiple indices of the callee's $conn$ are set to 1. The caller will be blocked (②) until the connection is established, where the value of the caller's $conn[caller\_eptp\_index]$ would be set to 1.

Callee. Callee traverses its $conn$ to find the first index with a value of 1 (③) and set *conn*[index] to 0 (④). Callee executes `accept_conn`(*caller_eptp_index, callee_eptp_index*) to set $conn[caller\_eptp\_index]$ of the caller to 1 with VMFUNC(⑤). Since the caller's $conn[caller\_eptp\_index] = 1$, the blockage is released and the connection is established (⑥). We set the $conn[caller\_eptp\_index] = 0$ and execute the process of data transmission (⑦). Through this mechanism, we can ensure that only one set of data transmission occurs between the caller and the callee at the same time. The process of establishment is short. The concurrency management ensures the correctness of the caller's multi-threaded connection. The order of processing actual requests is managed by the message queue (§V-C2).

2) Message Queue Management: For each callee, both the data transfer queue and the request queue need to be maintained simultaneously in lib-sidecar. As shown in Fig. 8. Since policy can dynamically and transparently implement data transmission of different protocols (§IV), the request queue will contain requests from VMFUNC transmission (§II-A) and requests from the network. When the data corresponding to the transmission path is transmitted, the request identifier will be added to this queue. lib-sidecar pops the identifier from the request queue and retrieves the transmitted data from the corresponding data queue for processing.

Data queue. The data buffer queue $\mathcal{Q}_d$ is responsible for maintaining that callee accepts data transfers from multiple callers. $\mathcal{Q}_d$ is a queue array with a length of EPTP upper limit 512. $\mathcal{Q}_d$ is used to manage the buffer sent from each caller. The servers of the queue contain a 4B-length meta for maintaining the data length and a 64KB buffer since we transfer the data less than 64KB through VMFUNC. For each time caller-callee establishes a data sending connection request, a server needs to be allocated on the queue $conn[caller\_eptp\_index]$ of callee first, and data transfer starts. Dividing the data buffer queue separately can avoid the conflict of VMFUNC-based transmission from different callers.

Request queue. After the data transmission is completed, $caller\_eptp\_index$ will be added to the request queue $\mathcal{Q}_r$. Each time, callee pops a $caller\_eptp\_index$ from $\mathcal{Q}_r$ and then pops the data from $\mathcal{Q}_d[caller\_eptp\_index]$. Then the callee processes it with `handle_service` and returns the result to the corresponding caller. The above design can achieve simultaneous multi-to-one connection and ensure the correctness of program execution in order.

D. Security Enforcement

We further discuss the security issues of VMFUNC-based communication in addition to the isolation in lib-sidecar (§IV).

VMFUNC trusted access. Since VMFUNC can jump to the memory address of another VM through trampoline code, it is necessary to inspect malicious code submitted by tenants for any illegal VMFUNC instructions. Here, we can refer to binary rewriting strategies similar to ERIM [34] and SkyBridge [27]. When a tenant submits a unikernel, it undergoes binary inspection, and any illegal VMFUNC instructions outside of the trampoline code page and the PKRU instructions are replaced with functionally equivalent instructions. So the malicious users' functions of callers or callees cannot execute these instructions even with return-oriented programming (ROP). This ensures that the VMFUNC instruction call only appears in the trampoline code of lib-sidecar for communication. The MPK instruction only invokes in the cross domain call for the isolation between user functions and the other compartments in the unikernels.

Communication control. All VMFUNC transmissions are based on the unified `send_msg` API, and the trampoline code is ultimately used for data transmission and stored in a buffer, ensuring controlled access to VMFUNC code. Furthermore, VMFUNC transmission occurs solely during the data transmission phase. This approach circumvents complex security access and stack management issues [6], [27] that arises from memory access to the callee by an untrusted caller's CPU.

Trusted services access. To ensure trusted access between services, we allow users to define a whitelist within their services in lib-vmfunc to prevent illegal service access sequences. When a caller accesses a service, the service name is first checked. If the function is allowed access according to the whitelist, the request proceeds while an exception is raised and sent to the gateway for illegal service.

VI. Evaluation

A. Experiment Setup

Implementation. Our entire experiment is conducted on a physical machine with 48 Intel(R) Xeon(R) Silver 4214 CPUs @2.20GHz, and a server with 256GB of memory and a network bandwidth of 10GB. The disk is a 2TB Intel S4520 SATA SSD. The operating system version is Ubuntu 18.04, with a kernel version of 4.15.18, where the version of the Linux kernel does not influence the implementation and performance in this work. In this experiment, all VMs are compiled with the KVM-guest option to support virtualization. When we start a unikernel or docker image, we set the number of CPUs as 1 (Sing-core unikernel takes one CPU by default) and memory as 64MB. The implementation of lib-sidecar and lib-vmfunc requires 5k LoCs and 1k LoCs of C code respectively. At the kernel level, modifications to KVM are 400. Runuk requires 2k LoCs of Go

code. The version of unikraft, lib-musl and lib-lwip are 0.11.0. The hypervisor is qemu with the version of 6.2.0.

Methodology. The comparative methods mainly include three categories: UKFaaS-based ablation, runtime evaluation, and state-of-the-art (SOTA) frameworks. The ablation experiments test the communication performance of UKFaaS under different conditions. The runtime evaluation tests the communication performance in Docker and Wasm with unikernel. For the SOTA evaluation, two representative serverless frameworks are selected for comparison. Nightcore [24] is a typical serverless framework based on the optimization of inter-thread communication. Since Nightcore also supports C language implementation, we choose it as the baseline framework. FaasFlow [21] is a typical framework that optimizes scheduling. They aim to deploy services with call chains on the same server and use shared memory for IPC (inter-process communication). We use FaaSFlow deployed with Python and ensure that all services are in an ideal state with IPC on the same server. Although the we assume shared memory is untrusted in our work, we implement the iveshem driver in unikraft to achieve the shared-memory-based communication as the simplified the baseline of SURE [7], where the security mechanism like MPK is not applied. Although the shared-memory-based method SURE is more efficient compared with UKFaaS, this method is not available in our scenario.

Benchmark. The experiment primarily revolved around analyzing the overhead of startup latency (§VI-B) end-to-end transmission (§VI-C) and forwarding (§VI-D), as well as testing the concurrent performance (§VI-E). We conducted performance tests on the Google online boutique benchmark [25] to evaluate the performance in real-world microservice scenarios (§VI-F). We use wrk [13] to send requests to the front service. There are default call sequences between these services with six different function calls in total. For more details of the benchmark please refer to the appendix section of SPRIGHT setup.

B. Startup Latency

As shown in Table III, we present the cold startup latency of the unikernel with different conditions. The application of the unikernel (UK) a http-server. (+s) means the application with lib-sidecar. (+v) means the application with lib-vmfunc, where the initialization process in Fig. 5 is needed. The additional overhead of startup delay caused by introducing these third-party libraries can be ignored. (w/f) means the VMM is Firecracker [1], where the Firecracker eliminates the unnecessary boot process in VMM and BIOs as discussed in SURE [7]. Overall, in the case of cold start, the runtime based on unikernel starts faster than Docker since the unikernel is more lightweight compared with Docker shown Fig. 2(a).

C. CPU Cycle of End to End Transmission

We analyze the CPU cycle in Table II. When concurrency requests from one caller come, the function caller and callee take `connect_wait` to ensure synchronous access. Since the above operations are based on VMFUNC without VM-Exit,

TABLE II
CPU CYCLE COST IN VMFUNC, COMMUNICATION, UKFAAS AND SERVICE RELATED OPERATIONS

VM ops	CPU cycle	**Comm. ops**	CPU cycle
VMFUNC	147	socket connect	137207
VM-Exit	8231	file I/O	891119
UKFaaS ops	/	**Service ops**	/
`handle_msg`	294	serialization	653848
		unserialization	55714
`connect_wait`	687	services	106-1708

TABLE III
BOOT TIME UNDER DIFFERENT CONDITIONS

	UK	UK(+s)	UK(+s,+v)	UK(w/f)	Docker
Startup latency (ms)	72	74	75	30	187

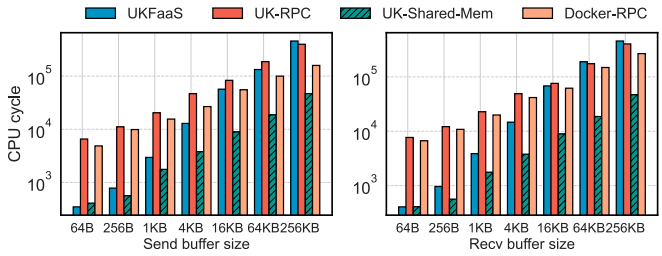


Fig. 9. CPU Cycle (↓) of UKFaaS (VMFUNC) and RPC data transmission with the change of buffer size. Improvement is the CPU cycle ratio of UK-RPC / UKFaaS (a) CPU cycle of buffer sending. (b) CPU cycle of buffer receiving.

the efficiency is much faster than network (socket connect) and shared file system (File I/O) in unikernels for data transfer.

After connection establishment, we evaluate VMFUNC-based transmission (UKFaaS), unikernel-based network transmission (UK-RPC), shared-memory-based transmission (UK-Shared-Mem) implemented on iveshem and network-based transmission (Docker-RPC) on Docker under different buffer sizes. To make a fair test, we omit the sidecar for unikernel-to-unikernel RPC communication with language C. We deploy two unikernels u1 and u2, where u1 sends a buffer with a specified size to u2 via socket, VMFUNC transmission or iveshem, and u2 returns a data packet to u1 after receiving it. Our experiment is conducted under a single thread, with data sizes ranging from 64B to 256KB as shown in Fig. 9.

By default, function communication based on unikernel uses the network. Since this communication method will trigger VM-Exit, when the network communication is less than 64KB, the transmission cost is greater than that based on VMFUNC. When the data scale further increases, network communication-based transmission is superior to UKFaaS transmission until 256KB. This is because the transmission cost of UKFaaS depends on the number of executions of `handle_msg` (§V-B). When the data transmission is 64B, only two round trips of VMFUNC are required, and the cost is much lower than VM-Exit. However, when the scale of data transmission further increases, the increase in the number of executions of `handle_msg` is

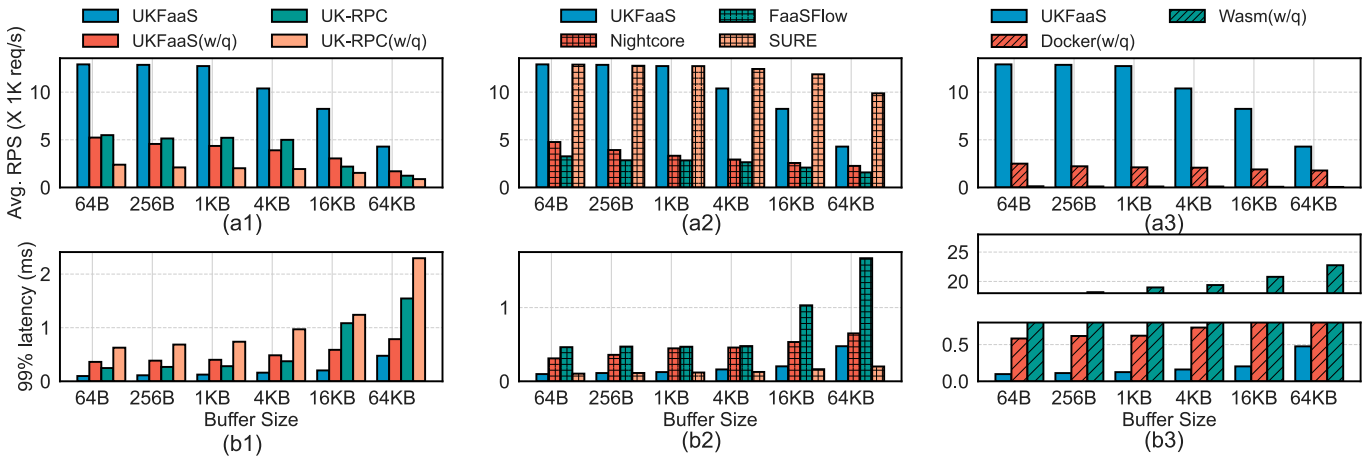


Fig. 10. Average RPS (↑) and latency (↓) with the change of buffer size (threads = connection = 1 in wrk). w/q means runtime with queue-proxy. RPS improvement is the ratio of UKFaaS / other methods and latency reduction is (other methods - UKFaaS) / other methods. (a) Average request per second (RPS). (b) 99%ile latency. Col 1: Ablation study on UKFaaS. Col 2: UKFaaS compares with other SOTA serverless frameworks. Col 3: UKFaaS compares with other runtimes.

less than that of VM-Exit. UK-Shared-Mem is equivalent to transplanting the communication between SPRIGHT [17] and SURE [7] to the unikernel. During the communication process, callee only needs to constantly poll the shared memory. Callee listens for whether the caller has data transmission and then executes subsequent processing without VM-Exit. Meanwhile, the cost of memory copying is much lower than that of network transmission or VMFUNC-based register transmission, so the transmission overhead is the lowest. However, such methods must ensure the credibility of shared memory. SURE can ensure the security of using shared memory among unikernels within the transmission system, but it cannot guarantee the operations of other processes on the shared memory on the host, and thus cannot be used in our scenario. The situation of Docker-RPC is similar to that of UK-RPC. The difference lies in that it does not need to go through VM-Exit, so the transmission cost is always lower than that of UK-RPC. Docker-RPC outperforms UKFaaS when the data scale is larger than 16KB. Overall, at 64B-4KB, the transmission efficiency of UKFaaS in `send` and `recv` has been improved by 3.3×-18.9× and 2.1×-16.4× compared to UK-RPC and Docker-RPC respectively.

D. Forwarding Performance Evaluation

In this section, we further verify the forward performance throughout and latency under different buffer sizes and transmission conditions. We take two unikernels u1 and u2 as HTTP servers, where we apply wrk to u1 and u1 and forward the packet with specified size to u2. Our experimental comparison is mainly evaluated from three dimensions. As shown in Fig. 10, they are respectively the communication mode of unikernel (the first column), the FaaS communication framework (the second column), and the runtime (the third column). Our wrk conditions on concurrent and thread number are both 1, and the amount of data transmitted is 64B-64KB.

Unikernel-based communication comparison. As shown in the first column of Fig. 10(a1) and (b1), we demonstrate the forwarding throughput and tail latency of UKFaaS and UK-RPC at different data scales with and without sidecar. According to the experimental conclusion in §VI-C, in the case of a small data scale (≤16KB), the CPU cycle overhead of sending and receiving data packets in UKFaaS is much smaller than the network forwarding overhead of UK-RPC. Further, we discuss the situation where lib-sidecar is introduced. w/q represents a queue proxy. In w/q, we introduce a unikernel server as a sidecar. The request first arrives at the sidecar and then is forwarded to u1. u1 forwards it to the sidecar and then to u2. UKFaaS conducts data transmission through registers based on VMFUNC, and UK-RPC implements network transmission. The existence of sidecar prolongs the data transmission path, especially greatly increasing the transmission cost when shared memory is unavailable. Overall, compared with UKFaaS (w/q) and UK-RPC at 16B-64KB, UKFaaS achieves a RPS improvement of 2.5×-2.9× and 2.1×-3.8× respectively. As for tail latency, UKFaaS performs 39%-71% and 55%-81% reduction compared with UKFaaS(w/q) and UK-RPC respectively.

SOTA FaaS communication system comparison. As shown in Fig. 10(a2) and (b2). We present a comparison between UKFaaS and some SOTA FaaS communication frameworks. We no longer compare SPRIGHT because it is also based on shared memory transmission, and the throughput of the unikernel itself is greater than that of the containers used by SPRIGHT [17]. Although shared memory is unavailable in the scenario of this article, we still introduce a performance evaluation of it.

Although inter-thread communication in Nightcore [24] can be efficient, the upper limit RPS of a unikernel HTTP server is larger than other threads that benefit from its single-address architecture. Besides, data transmission relies on post-based protocol, leading to extra overhead during transmission. As for the DAG schedule optimization-based methods FaaSFlow [21], the SOTA schedule case is IPC communication between instances with shared memory designed for Python. Therefore, we use a Python-based HTTPs server in FaaSFlow to test the forwarding RPS and tailed latency with IPC. Due to the poor

performance of Python, the result is even worse than Nightcore. Compared with Nightcore and FaaSFlow, UKFaaS achieves 1.9×-3.8× and 2.7×-4.5× improvement in RPS at 64B-64KB respectively. According to the conclusion of §VI-C, since SURE adopts the UK-Shared-Mem communication mechanism, its throughput and tail latency perform optimistically in the case where shared memory is available.

Runtime comparison. We also compare with other runtimes such as docker and WebAssembly (wasm) with sidecar since it is necessary for these runtimes to be deployed in a modern serverless system as shown in Fig. 10(a3) and (b3). Wasm cannot directly utilize sockets for communication, so it must rely on protocol standards implemented in JavaScript, where I/O operations still depend on the kernel API, resulting in poor performance of Wasm(w/q) with over 95% reduction in 99% tail latency. Due to the main bottleneck of overhead by request forwarding by sidecar in Knative, Docker (w/q) performs 50% to 83% in 64B-64KB compared with UKFaaS.

E. Concurrency Performance

In this section, we further extend the situation of §VI-D to different concurrency and the number of thread connections. wrk [13] is still used to increase the load on u1, and unikernel u1 forwards the request to u2. In wrk, we set different numbers of connections and threads to adjust the concurrency conditions. Meanwhile, in order to evaluate the influence of the introduction of message queue (§V-C), we also test the RPS under different numbers of callers to evaluate the effectiveness of message queue. The sidecar is not considered in this evaluation.

One-to-one concurrency. In this case, one caller u1 forwards the data to the callee u2 with different wrk conditions. As shown in Fig. 11(a), when the data transmission scale is 1KB, the number of connected transmissions gradually increases from 1 to 8. This is because in the case of low concurrency, the system throughput does not reach the saturation. When the number of concurrent connections is greater than 8, the throughput tends to be saturated. It is basically consistent with the conclusion in §VI-D in terms of RPS performance. SURE can always maintain the optimal performance because the transmission performance is based on shared memory when the shared memory is trusted. Furthermore, the conclusion in Fig. 11(a2) indicates that when the throughput reaches saturation, the increase in the number of threads does not affect its throughput performance. As shown in Fig. 11(b), when the amount of transmitted data further increases, the performance advantage of SURE becomes more prominent. Except for SURE, UKFaaS performs best in the case where the shared memory is not trusted.

Many-to-one concurrency. We further investigate the RPS performance when initiating data transfer to the same callee from multiple callers. The introduction of message queues allows multiple callers to simultaneously implement data transmission to a callee (§V-C). The callee only needs to return the data to the corresponding caller after completing each request. Here we compare the scheme (w/o msg queue) that

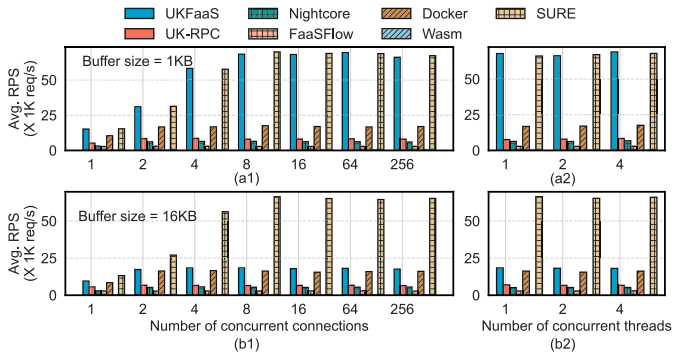


Fig. 11. RPS (↑) With the change of connections and threads in wrk. (a) RPS under different connections and threads in 1KB. (b) 16KB evaluation. RPS under threads = 1 and col 1 connections = 8 in col 2.

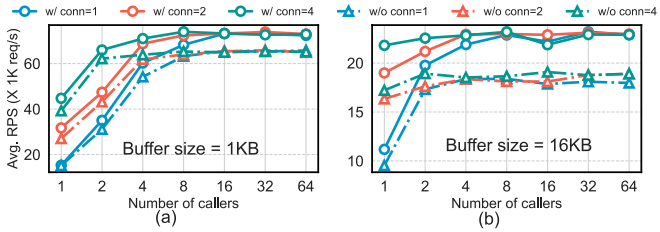


Fig. 12. The total RPS when different caller numbers simultaneously forward data packets of different lengths to a callee. "w/" indicates the introduction of a message queue, and "w/o" indicates no. "conn" represents the number of connections set by wrk. (a) Buffer size = 1KB. (b) Buffer size = 16KB.

does not introduce the message queue and only supports communication between one group of caller and callee at a time to evaluate the impact of the message queue on UKFaaS.

As shown in Fig. 12, we evaluate different numbers of caller connections (1-64), and the impact of the number of connections for adjusting the concurrency on the results under the specified number of caller connections on UKFaaS. Overall, the system throughput effect of introducing message queues is better, especially when RPS saturation is reached. The main reason is that it realizes asynchronous data transmission. The inability to achieve further improvement is due to the fact that the processed results rely on callee's CPU when being returned. After the introduction of the message queue, the RPS upper limit is increased by 1.12× and 1.28× respectively in 1KB and 16KB. In addition, when the number of caller numbers is 64, it exceeds the total number of CPUs in the host and the test executes correctly. It proves the feasibility of deploying this data transmission scheme in a multi-tenant scenario discussed in §V-B.

F. Realistic Workload Benchmark

In this section, we analyze the performance of UKFaaS and other methods in Google online boutique benchmark [25]. We provide benchmarks based on process and implement UKFaaS with Python-based service to prove the language compatibility as UKFaaS(py). Besides, we combine the data transmission process of UKFaaS into s-f architecture UaaF [6] so as to compare

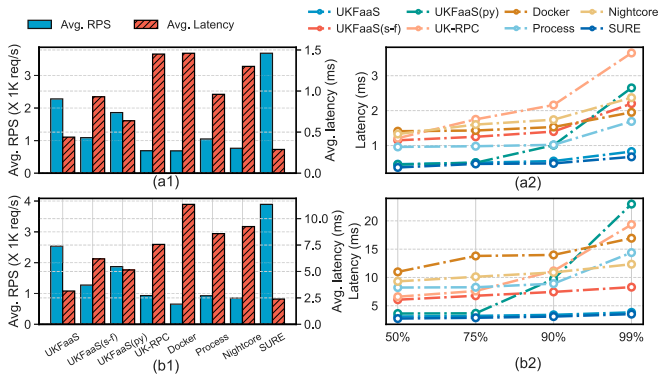


Fig. 13. Average RPS (↑) and latency (↓) distribution of online boutique benchmark. The text above the bar is the improvement and reduction of UKFaaS compared with other methods. Row 1: connections = 1, threads = 1. Row 2: connections = 8, threads = 4. (a) Average RPS and latency and the data show the UKFaaS's improvement on RPS and reduction in latency. (b) Latency distribution.

the performance in the benchmark. The structure transmitted during online boutique benchmark communication is a data structure of about 170KB that maintains information involved in mall transactions. For concise comparison results, sidecar containers are not involved in any communications within benchmarks since the previous evaluation proves the performance improvement without sidecar. We do not compare webassembly since the network performance is poor as proved in Fig. 11. Compared to the network testing in §VI-D, the microservices deployed as an HTTP server. Since network transmission is based on HTTP messages, we use cJSON [38] to support serialization and deserialization of transmitted structures with post protocol with the CPU cycle in Table II.

According to the results of Fig. 13(a1), (b1), we can see that UKFaaS performs best among other serverless architecture or runtime. Compared with UK-RPC, UKFaaS does not need an extra serialization process since transmission of UKFaaS does not include post-protocol. The additional overhead of VMFUNC compared to UKFaaS is due to the lack of optimization for communication chains, resulting in additional receiving processes. As for the docker network, it can be seen that UK-RPC performs worse than docker and process since the cost between VMs is higher due to the complexity of the virtualization. Since FaaSFlow/DataFlower only supports Python cases, we take a 170KB data transmission case to evaluate the performance. It performs worse than UKFaaS due to the performance limitation of Python. As for Nightcore, it is mainly designed for the small size ($\leq$1KB) transfer and has poor performance in the benchmark. To prove the compatibility in language, we implement UKFaaS(py) and its RPS is less than UKFaaS since Python runs slow than C. Besides, the transmission result needs to extra serialization step so as to pass the parameter from C to Python. The performance of SURE remains the best. According to the conclusion of §VI-C, it avoids the I/O overhead of data transmission. But it is not available in our case since shared memory is untrusted.

The result when the number connections = 8 and threads = 4 in Fig. 13(a2), (b2) shows a similar conclusion in RPS and latency distribution, where UKFaaS performs 1.8×, 3.5×, 3.4×, 2.3×, 3.2× and 3.5× better in RPS and keeps low latency in different distribution compared with UKFaaS(py), UK-RPC, docker, UaaF, Nightcore and FaaSFlow respectively.

VII. Related Work

A. VMFUNC-Based System

Previously, work based on VMFUNC mainly focused on secure data access between VMs [39], [40], [41] and communication acceleration for kernel modules [27]. UaaF [6] is the first work to introduce VMFUNC into unikernel. Unfortunately, its inability to achieve data transmission has limited its applicability in general serverless scenarios.

However, most of these works are applied to the protection of traditional VMs or trusted communication acceleration between microkernels and kernel modules, rather than involving data plane communication acceleration at the application layer of microservices in serverless scenarios. Lightweight unikernel has the advantage of using VMFUNC for data communication acceleration due to its single address space feature. In the future, technologies related to EPT upper limit extension and secure isolation can be applied to large-scale trusted unikernel-based microservice scenarios.

B. Optimizing Serverless Computing

Current serverless work mainly focuses on optimizing cold start at the container level [42], [43], communication acceleration [2], [7], [21], [23], [24], [32], heterogeneous hardware computing [44]. However, container-based implementations of optimization work are not directly applicable to unikernel VMs. Although some techniques have also been optimized for VMs in serverless scenarios [8], [9], [10], these efforts mainly focus on the cold start phase and do not apply to complex microservice communication scenarios.

UKFaaS introduces an external library to meet the needs of function containers in a serverless framework and saves network forwarding bottlenecks [17] at a low intrusive cost. Additionally, we apply UaaF's VMFUNC technology to eliminate the overhead caused by VM-Exit during communication without relying on shared memory in previous work [7].

VIII. Conclusion

This article proposes the UKFaaS system, which mainly includes two modules: lib-sidecar and lib-vmfunc. Based on the customizable feature of unikernel, it optimizes data plane communication. Lib-sidecar integrates the functionality of heavyweight sidecar containers responsible for serverless framework applications as external libraries to reduce the request forwarding overhead and integrates the communication function based on VMFUNC with adaptive policy. Lib-vmfunc leverages the advantages of a unikernel single-address OS to implement VMFUNC-based data transfer protocols when shared memory is not available. Our experiment shows that UKFaaS can realize 1.8×-3.5× RPS improvement compared with the advanced serverless framework in Google online boutique microservice benchmark.

REFERENCES

[1] A. Agache et al., "FireCracker: Lightweight virtualization for serverless applications," in *Proc. NSDI*, vol. 20, 2020, pp. 419–434.

[2] S. Shillaker and P. R. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX)*, 2020, pp. 419–433.

[3] Y. Ren et al., "Shared-memory optimizations for inter-virtual-machine communication," *ACM Comput. Surv*, vol. 48, no. 4, pp. 49:1–49:42, 2016.

[4] Y. Zhang et al., "SHELTER: extending arm CCA with isolation in user space," in *Proc. 32nd USENIX Secur. Symp., USENIX Secur.* Anaheim, CA, USA. USENIX Association, 2023, pp. 6257–6274.

[5] S. Kuenzer et al., "Unikraft: Fast, specialized unikernels the easy way," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 376–394.

[6] B. Tan, H. Liu, J. Rao, X. Liao, H. Jin, and Y. Zhang, "Towards lightweight serverless computing via unikernel as a function," in *Proc. IEEE/ACM 28th Int. Symp. Qual. Service (IWQoS)*, Piscataway, NJ, USA: IEEE Pres, 2020, pp. 1–10.

[7] F. Parola, S. Qi, A. B. Narappa, K. K. Ramakrishnan, and F. Risso, "SURE: Secure unikernels make serverless computing rapid and efficient," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, Redmond, WA, USA. New York, NY, USA: ACM, 2024, pp. 668–688, doi: 10.1145/3698038.3698558.

[8] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: Skip redundant paths to make serverless fast," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–15.

[9] G. Gain, C. Soldani, F. Huici, and L. Mathy, "Want More Unikernels? Inflate Them!" in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 510–525.

[10] L. Ao, G. Porter, and G. M. Voelker, "FAASNAP: FAAS made fast using snapshot-based VMS," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 730–746.

[11] "NanoVM. deploy unikernels to any cloud in seconds with no Devops." 2025. Accessed: May 8, 2025. [Online]. Available: https://github.com/unikraft/lib-lwip.git

[12] J. Hwang, K. K. Ramakrishnan, and T. Wood, "NETVM: High performance and flexible networking using virtualization on commodity platforms," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation (NSDI),* Seattle, WA, USA. USENIX Association, 2014, pp. 445–458.

[13] "Wrk - a http benchmarking tool," 2025. Accessed: May 8, 2025. [Online]. Available: https://github.com/wg/wrk

[14] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–34, 2022.

[15] Q. Liu, D. Du, Y. Xia, P. Zhang, and H. Chen, "The gap between serverless research and real-world systems," in *Proc. ACM Symp. Cloud Comput. (SoCC),* Santa Cruz, CA, USA, 2023, pp. 475–485, doi: 10.1145/3620678.3624785.

[16] "Knative," 2025. Accessed: May 8, 2025. [Online]. Available: https://katacontainers.io

[17] S. Qi, L. Monis, Z. Zeng, I-c Wang, and K. Ramakrishnan, "Spright: Extracting the server from serverless computing! High-performance EBPF-based event-driven, shared-memory processing," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 780–794.

[18] "EBPF: Dynamically program the kernel for efficient networking, observability, tracing, and security," 2025. Accessed: May 8, 2025. [Online]. Available: https://ebpf.io

[19] J. Chen et al., "Remote procedure call as a managed system service," in *Proc. 20th USENIX Symp. Netw.ed Syst. Des. Implementation (NSDI)*, Boston, MA,USA. USENIX Association, 2023, pp. 141–159.

[20] H. Saokar et al., "ServiceRouter: Hyperscale and minimal cost service mesh at meta," in *Proc. 17th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, Boston, MA, USA. USENIX Association, 2023, pp. 969–985.

[21] Z. Li et al., "FaasFlow: Enable efficient workflow execution for function-as-a-service," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Operat. Syst.*, 2022, pp. 782–796.

[22] Z. Li, C. Xu, Q. Chen, J. Zhao, C. Chen, and M. Guo, "DataFlower: Exploiting the data-flow paradigm for serverless workflow orchestration," in *Proc. 28th ACM Int. Conf. Architect. Support Programm. Lang. Operat. Syst.*, vol. 4, Vancouver, BC, Canada. New York, NY, USA: ACM, 2023, pp. 57–72.

[23] G. Liu et al., "FUYAO: DPU-enabled direct data transfer for serverless computing," in *Proc. 29th ACM Int. Conf. Architect. Support Programm. Lang. Operat. Syst.*, vol. 3, La Jolla, CA, USA. New York, NY, USA: ACM, 2024, pp. 431–447.

[24] Z. Jia and E. Witchel, "NightCore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proc. 26th ACM Int. Conf. Architect. Support Program. Lang. Operat. Syst.*, 2021, pp. 152–166.

[25] "Online boutique," 2025. Accessed: May 8, 2025. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[26] "VMFUNC — invoke VM function," 2025. Accessed: May 8, 2025. [Online]. Available: https://www.laruence.com/x86/VMFUNC.html

[27] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "Skybridge: Fast and secure inter-process communication for microkernels," in *Proc. 14th EuroSys Conf.* 2019, pp. 1–15.

[28] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo, "EBBRT: A framework for building per-application library operating systems," in *Proc. 12th USENIX Symp. Operat. Syst. Des. Implement. (OSDI)*, 2016, pp. 671–688.

[29] H. Kuo, D. Williams, R. Koller, and S. Mohan, "A Linux in unikernel clothing," in *Proc. 15th EuroSys Conf.*, Heraklion, Greece, 2020, pp. 11:1–11:15.

[30] A. Raza et al., "Unikernel linux (UKL)," in *Proc. 18th Eur. Conf. Comput. Syst. (EuroSys),* Rome, Italy, 2023, pp. 590–605.

[31] R. Russell, "VIRTIO: Towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.

[32] C. Jin et al., "DITTO: Efficient serverless analytics with elastic parallelism," in *Proc. ACM SIGCOMM Conf., ACM SIGCOMM*, New York, NY, USA, Sep. 2023, pp. 406–419.

[33] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *Proc. 20th USENIX Symp. Netw. Syst. Des. Implementation (NSDI),* Boston, MA, USA, 2023, pp. 1489–1504.

[34] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: "Secure, efficient in-process isolation with protection keys," (MPK)," in *Proc. 28th USENIX Security Symp. (USENIX Security)*, 2019, pp. 1221–1238.

[35] T. Ma et al., "Efficient scheduler live update for Linux kernel with modularization," in *Proc. 28th ACM Int. Conf. Architect. Support Programm. Lang. Operat. Syst.* (*ASPLOS),* Vancouver, BC, Canada. New York, NY, USA: ACM, 2023, pp. 194–207.

[36] H. Lefeuvre et al., "FlexOS: Towards flexible OS isolation," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2022, pp. 467–482.

[37] "Runu. OCI runtime to runu Unikraft like containers." 2025. Accessed: May 8, 2025. [Online]. Available: https://unikraft.org/docs/getting-started/integrations/container-runtimes

[38] "CJSON," 2025. Accessed: May 8, 2025. [Online]. Available: https://github.com/DaveGamble/cJSON.git/

[39] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1607–1619.

[40] Z. Hua, D. Du, Y. Xia, H. Chen, and B. Zang, "EPTI: Efficient defence against meltdown attack for unpatched VMS," in *Proc. USENIX Annu. Techn. Conference (USENIX ATC 18)*, 2018, pp. 255–266.

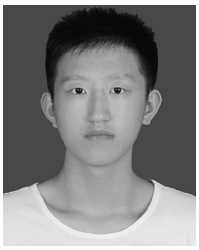
[41] K. Yasukata, H. Tazaki, and P.-L. Aublin, "Exit-less, isolated, and shared access for virtual machines," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Operat. Syst.*, vol. 3, 2023, pp. 224–237.

[42] Z. Li et al., "Help rather than recycle: Alleviating cold Startup in serverless computing through Inter-Function container sharing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2022, pp. 69–84.

[43] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM Int. Conf. Architect. Support Program. Lang. Operat. Syst.*, 2021, pp. 559–572.

[44] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless computing on heterogeneous computers," in *Proc. 27th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2022, pp. 797–813.

Zhenqian Chen received the B.E. degree from Zhejiang University of Technology, Hangzhou, China, in 2021. He is currently working toward the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include library operation systems and virtualization.



Yuchun Zhan received the B.E degree from Hohai University, China, in 2022. He is currently working toward the master's degree with the School of Software Engineering, Zhejiang University, China. His research interests include library operation systems and virtualization.

Peng Hu received the B.E. degree from Wuhan University, Wuhan, China, in 2021. He is currently working toward the master's degree with the College of Software, Zhejiang University, Hangzhou, China. His research interests include library operation systems and reinforcement learning.

Xinkui Zhao received the Ph.D. degree from the College of Computer Science and Technology, Zhejiang University, in 2016. He is a ZJU 100-Young Professor with Zhejiang University, China. His research interests include cloud-native architecture and technologies, edge computing, and service computing.

Muyu Yang received the B.E. degree from the Northeast Agricultural University, China, in 2023. He is currently working toward the master's degree with the School of Software Engineering, Zhejiang University, China. His research interests include library operation systems and virtualization.

Siwei Tan received the B.S. degree in 2019 from Zhejiang University, where he is currently working toward the Ph.D. degree, both in computer science. His research interest includes the application of quantum computing.

Lufei Zhang is currently a Researcher with SKL-MEAC, Wuxi, China. His research interests include cloud computing, big data, etc.

Liqiang Lu received the Ph.D. degree from the School of Computer Science, Peking University, Beijing, China, in 2022. He is an Associate Professor (ZJU100 Young Professor) with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. He has authored over 20 scientific publications in premier international journals and conferences in related domains, including ISCA, IEEE Micro, FCCM, TCAD, and DAC. His research interests include quantum computing, deep-learning accelerator design, and spatial architecture optimization.

Jianwei Yin (Senior Member, IEEE) received the Ph.D. degree in computer science from Zhejiang University, in 2001. He was a Visiting Scholar with Georgia Institute of Technology. He is currently a Full Professor with the College of Computer Science, Zhejiang University. He has published more than 100 papers in top international journals and conferences. His research interests include service computing and business process management. He is an Associate Editor of the IEEE TRANSACTIONS ON SERVICES COMPUTING.

Zuoning Chen is currently a Researcher with SKL-MEAC, Wuxi, China. Her research interests include operation system and system security.