Towards Lightweight Serverless Computing via Unikernel as a Function

Bo Tan, Haikun Liu, Jia Rao†, Xiaofei Liao, Hai Jin, Yu Zhang
National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab/Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
Email: {btan, hkliu, xfliao, hjin, zhyu}@hust.edu.cn
†Department of Computer Science and Engineering, The University of Texas at Arlington
Email: jia.rao@uta.edu

***Abstract*—Serverless computing, also known as "*Function as a Service* (*FaaS*)", is emerging as an event-driven paradigm of cloud computing. In the FaaS model, applications are programmed in the form of functions that are executed and managed separately. Functions are triggered by cloud users and are provisioned dynamically through containers or *virtual machines* (VMs). The startup delays of containers or VMs usually lead to rather high latency of response to cloud users. Moreover, the communication between different functions generally relies on virtual net devices or shared memory, and may cause extremely high performance overhead. In this paper, we propose *Unikernel-as-a-Function* (*UaaF*), a much more lightweight approach to serverless computing. Applications are abstracted as a combination of different functions, and each function are built as an unikernel in which the function is linked with a specified minimum-sized *library operating system* (LibOS). UaaF offers extremely low startup latency to execute functions, and an efficient communication model to speed up inter-functions interactions. We exploit an new hardware technique (namely VMFUNC) to invoke functions in other unikernels seamlessly (mostly like inter-process communications), without suffering performance penalty of VM Exits. We implement our proof-of-concept prototype based on KVM and deploy UaaF in three unikernels (MirageOS, IncludeOS, and Solo5). Experimental results show that UaaF can significantly reduce the startup latency and memory usage of serverless cloud applications. Moreover, the VMFUNC-based communication model can also significantly improve the performance of function invocations between different unikernels.**

I. Introduction

Recent years have witnessed a tremendous growth in serverless cloud computing, also known as *Function-as-a-Service* (FaaS), in which programmers use serverless functions as building blocks to construct applications. FaaS allows programmers focus on the business logic of their applications, while the cloud is responsible for system administrations, including runtime system configuration, resource allocation, task scheduling, and authentication. Compared to traditional cloud computing, serverless computing releases programmers from the burden of purchasing and managing cloud resources while offering better scalability, flexibility, and much faster deployment.

Serverless applications are built by deploying a collection of functions and a set of events that are configured to invoke these functions. The resources and runtime environment required by serverless functions, a.k.a, an execution sandbox, is provisioned upon receipt of user requests and released once function invocations are completed. Such fine-grained function-level abstractions enable cloud providers to automatically scale up/down resource provisioning in response to time-varying workloads and allows users to pay only for the actual amount of resources their functions consume. Major cloud providers, such as AWS Lambda [1], IBM Cloud Functions [2], Microsoft Azure Functions [3], Google Cloud Functions [4] all support FaaS and have showcased its advantages in cost-efficiency, ease-to-program, and scalability.

A key challenge in serverless computing is the provisioning of the ephemeral execution environment for the user-defined, short-lived functions. On the one hand, the execution sandbox should include the OS service and application runtime to execute the user functions and provide strong security/fault isolation. On the other hand, the sandbox should not consume too much resource because hundreds or thousands of functions may run on a single host. The aggregate overhead of sandbox management is usually substantial. Furthermore, the startup latency of the sandbox, i.e., the time taken to initialize the runtime environment and to allocate resources to the sandbox, is crucial to user experience.

Various types of sandboxes have been adopted in serverless computing. Conventional virtual machines offer strong isolation but run full-fledged guest OSes inside them, thereby suffering from high performance overhead. In contrast, containers achieve near-native performance but do not guarantee adequate isolation since they share the same OS kernel. A few lightweight virtualization techniques, such as Google gVisor [5] and Amazon FireCracker [6], aim to achieve the best of two worlds. However, there still remain three issues. First, recent works [7], [8] have found that the startup latency, including the initialization of system states and the loading of application libraries, still dominates the overall execution time of sandboxes, indicating that the overhead to manage sandboxes outweighs the actual function execution time and resources are not efficiently utilized. Second, the existing sandbox designs, which mainly focus on small resource footprints and isolation, incur high communication costs across sandboxes, limiting the current serverless architectures mostly

978-1-7281-6887-6/20/$31.00 ©2020 IEEE

978-1-7281-6887-6/20/$31.00 ©2020 IEEE

to loosely-coupled, stateless microservices, e.g., RESTful Web services. Third, there is a high level of redundancy across serverless functions. For example, many functions may load the same application libraries or different users may launch the same function. Since individual functions are encapsulated in their own sandboxes, there are lacks of mechanisms to reduce such redundancy, further contributing to the long startup latency.

This paper explores a different approach to construct serverless applications. Unlike the existing FaaS paradigm, in which coarse-grained, stateless functions are executed in heavy sandboxes, we propose *Unikernel-as-a-Function* (UaaF). It decomposes a business logic into a series of low-level function invocations, each running in a unikernel (i.e. a fine-grained, lightweight sandbox). A unikernel is a single-address-space machine image with a minimal set of OS services compiled specifically to run only a single application. This design requires only the necessary OS and application libraries to be loaded before a function starts to execute, thereby greatly shortening the startup latency. In addition, low-level functions that are commonly called can be shared among different applications, which helps reduce redundancy. Finally, UaaF allows for the design of more sophisticated serverless applications that involve inter-function communication.

In UaaF, an application comprises two types of functions: *session* and *library* functions. A session is a proxy function that defines the skeleton (workflow) of the application. It specifies which library functions to invoke when the application starts to execute. Library functions are pre-defined routines uploaded to the cloud by application developers, which can be invoked and shared by multiple applications. Upon receiving a user request for invoking a serverless application, the cloud provider creates a session function for the application and links it with the corresponding library functions specified in a configuration file submitted by the user.

While unikernels enforce isolation between sessions and shared library functions, function invocations across unikernels incur high overhead, mainly due to frequent context switches and expensive VM-exits. To this end, UaaF leverages a recent virtualization feature on Intel CPUs, VMFUNC, to avoid the performance penalty of function invocation across unikernels. VMFUNC allows a unikernel, which runs as a guest OS in a virtual machine, to invoke a function in another VM without a VM-exit. Specifically, UaaF configures VMFUNC to switch the *Extend Page Table Pointer* (EPTP) between two unikernels, and thus allows one unikernel to access virtual memory and execute functions in another unikernel.

We have implemented a prototype of UaaF based on *Kernel-based Virtual Machine* (KVM). The kernel maintains a list of unikernels' EPTPs. UaaF offers fine-grained function abstractions and guarantees performance isolation among functions. It is also with high scalability and flexibility to deploy new shared functions in the cloud platforms. UaaF can be deployed by using typical unikernels such as MirageOS [9], IncludeOS [10], OSv [11], and Solo5 [12]. Experimental results demonstrate that the time cost of function invocation between unikernels is only 138 cycles, which is even faster than the cost of an inter-process call on a physical host. Moreover, UaaF platforms also can significantly reduce the memory footprint and startup latency of serverless functions.

In summary, our major contributions are as follows:

- We revisit the traditional FaaS paradigm and propose a new serverless model (namely UaaF) to improve the resource utilization of cloud platforms, and to decrease the startup latency of serverless functions.
- We exploit the VMFUNC instruction provided by Intel CPUs to support seamlessly runtime code sharing among different unikernels, without suffering the performance penalty of VM-exits.
- We evaluate our UaaF model on KVM-based virtual servers. Experimental results show that UaaF can significantly reduce the startup latency of serverless functions by about 300 times compared with traditional unikernels. Moreover, our inter-function communication policy can also reduce the latency of function invocation by 3 orders of magnitude compared to the traditional inter-domain communication mechanisms.

The remainder of this paper is organized as follows. Section II introduces the background of this paper. Section III presents the observations of FaaS which motivate our new design. Section IV introduces the basic design and workflow of UaaF. In Section V, we describe implementation details about UaaF. Section VI presents experimental results. We describe the related work in Section VII, and discuss the advantage and limitation of UaaF in Section VIII.

II. Background

In this section, we first describe the FaaS paradigm and how sandboxes are provisioned to offer serverless cloud service. Then, we introduce the background of unikernels and the Intel EPTP switching techniques.

A. Function as a Service

In FaaS, an application is usually programmed as a collection of microservices, each of which is executed and managed as separate functions. Functions are event-driven, triggered by a variety of requests. The cloud provider is responsible for provisioning a sandbox for the function code and allocating the resources to the sandbox. As shown in Fig. 1, when the cloud platform receives a request, a task scheduler selects a worker to execute the task. The worker runs the triggered function in an independent sandbox, which is generally a container or a VM. To ensure isolation, most commercial platforms, such as AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions, only allow one function to execute in a sandbox, and then destroy the sandbox after the task is completed.

To support stateful applications, FaaS platforms often use a separate cloud service for the storage of information exchanged between micro-services (e.g., Google Cloud Storage, AWS S3, or Azure Blob Storage), and users pay for it separately. FaaS charges users for the actual execution time of the function. Since serverless functions are typically short-lived,

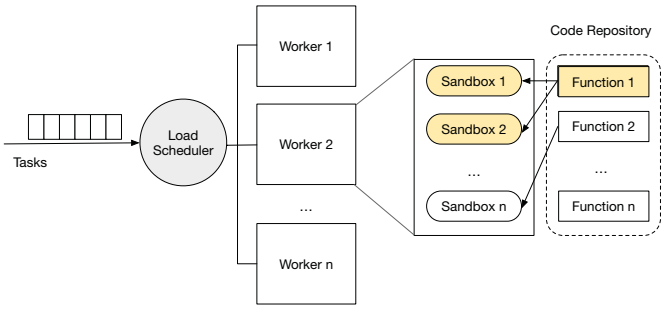


Fig. 1: A typical FaaS architecture. Functions are stored in a code repository. When a task arrives, a worker launches a sandbox to run the required function.

FaaS can greatly reduce the cost compared to traditional cloud services that charge on an hourly basis. As the sandbox is an ephemeral execution environment, the overhead of starting and managing such sandboxes should be kept low. Ideally, the time to start a sandbox should be no larger than the actual function execution time.

B. Unikernels

Unikernels [9] are constructed by using library operating systems in which the OS kernel is tightly coupled with user applications as individual software appliances. Unikernels are able to significantly reduce the memory footprint and attack surface. They work together with traditional hypervisors (e.g., Xen, KVM) and more lightweight virtualization schemes like Firecracker [13]. Other projects such as Solo5 further extends minimalism to the underlying monitor and the exposed interfaces for high performance. Unikernels are a natural fit for the FaaS model because of the lightweight and strong security isolation. Unikernels are as efficient as processes [14] while can be as secure as virtual machines.

Fig. 2 shows the startup latency of a simple function written in C using unikernels and containers. Unikernels (i.e., MirageOS, Solo5, OSv, and IncludeOS) are not necessarily faster than containers. While MirageOS and Solo5 can achieve a startup latency as short as a few milliseconds, the latency of other unikernels, such as IncludOS and OSv, are comparable to that of containers. The figure also shows that caching or reusing an already booted container (i.e., warm startup) can greatly reduce the startup latency. However, the overhead of container virtualization, including the cost of maintaining namespaces, is still significant and incurs more than 100 milliseconds startup latency.

C. EPTP Switching

VMFUNC is an Intel hardware extension for virtualization. It allows a program in VMX non-root mode to invoke a VM function in the hypervisor. This allows programs inside a VM to call pre-registered, privileged functions in the hypervisor without trapping into the hypervisor or triggering a VM-exit. Current Intel processors only support one VM function, i.e., EPTP switching, which switches the virtual address space of the caller program to one specified address in an EPTP list.

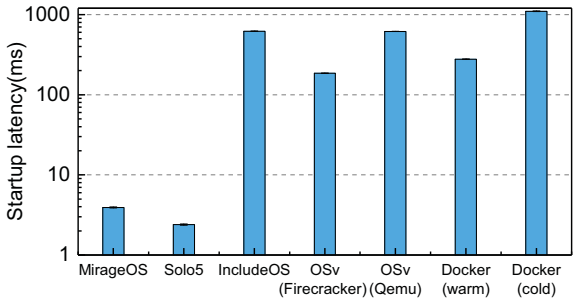


Fig. 2: Function startup latency in various sandboxes

Extend Page Table (EPT) is a *Second Level Address Translation* (SLAT) technology. The hypervisor maintains one EPT per guest OS, which maps the *guest physical address* (GPA) to the *host physical address* (HPA). EPTP switching allows one VM, after a VMFUNC instruction is successfully called, to switch the pointer of the current EPT to a different EPT address in the EPTP list, thereby allowing the VM to access the data located in another VM's memory address. Since the EPTP list is maintained by the hypervisor and the guest OS has no access to it, security isolation is guaranteed. Most importantly, the invocation of the VMFUNC instruction and the switch of the virtual addresses do not necessarily require to flush the TLB cache if the *Virtual Processor ID* (VPID) feature is enabled. It takes approximately 150 cycles to access another VM's virtual address space, making EPTP switching an attractive solution to compose serverless functions that are isolated in different unikernels.

III. Motivation

The recent advances of FaaS have made it an important service model in cloud computing. However, there are still impediments to fully unlock the potential of this new computing paradigm. In this section, we show that the performance of FaaS is significantly limited by the architecture of the existing FaaS architecture, the startup latency of serverless functions, and the coarse-grained function management.

A. Inefficient Communications in Stateful Applications

The existing FaaS architectures are most suitable for stateless applications, in which functions and their invocations are independent from each other. Currently, FaaS frameworks employ a decoupled computation and storage architecture to implement stateful applications. Information exchanged between functions are stored as messages in a separate storage service, such as Amazon S3. However, such a communication scheme is not efficient for applications with fine-grained, frequent communications. The key issue is that current FaaS platforms use the same mechanism for handling external events (user calling a function) and internal events (function calling a function) [7], which lead to high latency. Hierarchical message bus [7] and message rendezvous server [15] have been proposed to address this issue. Since these approaches are still based on message passing, they do not scale well in serverless applications with fine-grained communications.

B. High Startup Latency of Functions

Low startup latency is crucial to serverless function provisioning since function execution time is quite short, typically in the range of tens to hundreds of milliseconds. As discussed above, functions are executed in separate sandboxes. Therefore, the end-to-end latency of serverless functions, i.e., the time taken before a function returns to users, comprises three parts: 1) sandbox startup time; (2) the initialization time of the execution environment inside the sandbox; (3) the execution time of the serverless function.

The first part is mainly determined by the implementation of the sandbox. There is usually a trade-off between isolation and performance. In general, VMs offer stronger isolation than containers while suffering much higher performance overhead. Google gVisor is a virtualized container environment with a emphasis on isolation. It has been reported that the improved security in gVisor leads to at least 2x higher performance overhead than other containers [16]. Amazon firecracker [6] runs a container in a microVM to achieve the best of the two worlds. However, as shown in Figure 2, neither of these two approaches achieve the satisfactory, low latency for functions that only last for tens of milliseconds.

The initialization time for application runtimes and libraries can be even longer than the sandbox startup time. It takes non-trivial time to load modules or libraries during a cold startup of containers or VMs [17], [18]. The existing FaaS frameworks install all libraries required by application at startup in every sandbox, even though only a small portion of the runtime is needed during execution. This results in a high level of redundancy in the memory footprint and high latency in loading libraries. It takes about 2ms to 800ms to initialize the runtime library when invoking a lambda function for the first time [19], and the Google container platform reports that 80% of the startup latency are attributed to library installation [20].

A possible workaround to avoid a cold start is keeping recently used sandboxes "warm" for some time in case they can be reused, at the expense of a waste of resources. This approach has two limitations. First, it is difficult to find applications that have identical execution environments, though much of their sandboxes can be shared. There are lacks of mechanisms for reducing the redundancy between similar but not identical sandboxes. Second, even if a sandbox can be reused, it cannot be simultaneously used by multiple functions. As such, duplicate sandboxes need to be provisioned if there are concurrent requests.

C. Complex Correlations between Functions

Many cloud applications are complex, with each instance consisting of a combination of multiple functions. As individual functions are isolated in separate sandboxes, inter-function invocations are expensive. As discussed above, the current practice to use third-party storage as a means for information exchange is inefficient, especially when applications scale. There is a growing need for developing a FaaS framework capable to handle complex inter-function invocations. Inspired by the high efficiency of unikernels (pre-compiled machine images for specialized purposes) and the benefit of Amazon AWS step Functions [21] (pre-defined function workflow), we envision that high efficiency and small resource footprint can be simultaneously achieved through composing serverless applications with pre-built unikernel-based functions. This calls for a fine-grained abstraction of serverless computing at the level of individual library functions.

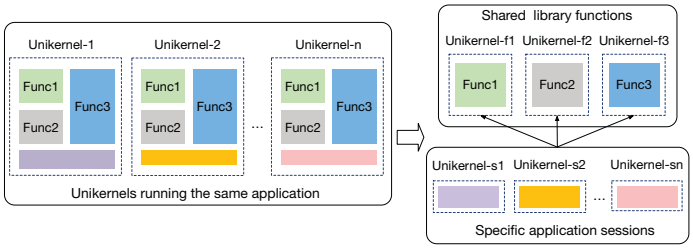


Fig. 3: In UaaF, functions are abstracted form applications and shared by multiple tasks.

IV. Unikernel as a Function

To reduce the startup latency and memory footprint of serverless functions, we propose *Unikernel-as-a-Function* (UaaF), a new method for constructing serverless applications. The design goals of UaaF are as follows.

- *Efficiency.* UaaF should reduce the startup latency of serverless functions, provide efficient inter-function communications, and reduce the memory footprint.
- *Isolation.* UaaF should provide VM-level isolation while preserving the lightweight and efficiency of unikernels.
- *Ease-to-use.* UaaF should provide an easy-to-use programming interface to integrate with popular unikernels and be compatible with different hypervisors.

A. Overview

An application in UaaF is composed of two parts: a program skeleton (workflow) and a set of library functions. Fig. 3 shows the difference between deploying an application with UaaF and the traditional FaaS approach. The traditional FaaS installs all libraries required by the application in one function (sandbox) and runs each task individually. In contrast, we abstract fine-grained functions from application codes and deploy these shareable libraries in different unikernels. A special unikernel called "session" acts as a proxy of a serverless application. It includes the workflow of the application, i.e., the sequence of library function invocations, and represents the serverless application in the task scheduler. Compared to traditional FaaS sandboxes, a session can be launched more quickly. In this way, application programmers can focus on application logic and link the shared serverless functions in a session.

Multiple sessions can invoke the same library function concurrently. UaaF installs separate stacks for different sessions within the virtual address space of called library functions. Therefore, only one copy of a library function needs to be kept in memory, reducing memory redundancy across different serverless applications. In addition, if library functions are

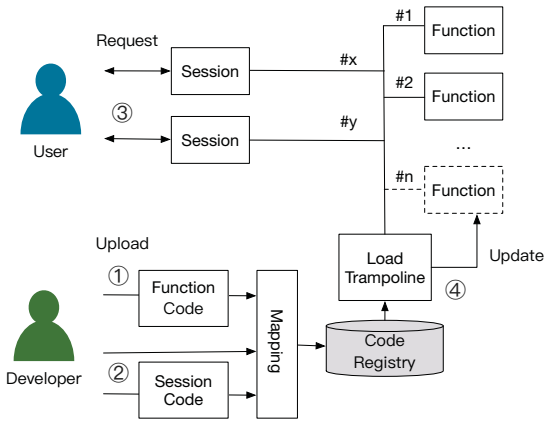


Fig. 4: The workflow of UaaF

already available in memory, the startup time of applications that invoke these functions can be greatly reduced.

UaaF assigns each library unikernel a specific *ID*. A remote call from session unikernels must provide the *ID* of the library function that it is going to call. The session unikernel executes a VMFUNC instruction along with the corresponding caller IDs. To enable cross-unikernel function call, UaaF leverages EPTP-switching to switch the virtual address spaces between the session and library unikernels. UaaF employs a similar mechanism proposed in [22] to install a trampoline in the caller and callee unikernels.

A trampoline is a special memory page in the memory address space of unikernels. It cooperates the EPTP list with EPTP-switching related instructions to handle function invocations in the VMX non-root mode. Since a VMFUNC instruction is correlated with a ID of the EPTP, UaaF should manage the caller-callee mappings between different unikernels. UaaF allocates a page to the EPTP list for each client, so that the session of a client can not invocate other functions (unikernels) if the EPTP list does not contain the corresponding EPTPs.

B. UaaF Workflow

In this paper, we define a special unikernel which represents the workflow (skeleton) of serverless applications as a *session*. In contrast, we define the library unikernels called by client sessions as *functions*. We describe the workflow of UaaF applications in the following, as shown in Figure 4.

1) *Application Deployment.* Programmers upload the unikernel image of a function to the code repository (①), and provide an access interface by exposing the function's entry address in a trampoline, which is also loaded in the memory space of the function image once it starts up. Programmers also deploys the unikernel images of sessions together with a list recording the mappings between the session and called functions (②).
2) *User Requests.* When the cloud platform receives a user request, UaaF launches a session (③) and the corresponding functions needed by the session on demand. These functions are mapped to the session in a session-function (S-F) mapping table when the functions are initialized. As shown in Figure 4, a function is mapped to different sessions using the same label. Actually, the label is not necessary to be the same for different sessions. The session terminates soon when the client closes the connection. Because a function might serve multiple sessions, UaaF should check whether there is still reference to the function when the session terminates. The functions can be terminated or kept warm for a while if they are not used by any session.
3) *Function Updating.* Programmers uploads the updated function image to the code repository. UaaF starts the new function and updates the mappings between sessions and the refereed function (④). In this way, other sessions can use the latest version of the function later. For sessions, they can be updated by simply replacing the old images of sessions.

V. Implementation

We implement our system based on KVM and Solo5. KVM is a virtualization module in the Linux kernel which provides hardware-assisted virtualization for guest OSes. For high performance, UaaF also leverages Solo5 which is a lightweight and customized unikernel execution environment. Solo5 is responsible for setting up unikernels and emulation of hardware such as tap and block devices. It only contains some necessary interfaces needed by unikernels. UaaF can also run other unikernels such as IncludeOS and OSv that are not supported by Solo5.

Figure 5 shows the architecture of UaaF. (1) The EPTP list in the KVM kernel stores EPT entries that would be used by the VMFUNC instructions. (2) A trampoline is integrated into each unikernel, and is used to handle remote calls between unikernels. (3) There are two modules in the user mode. The session-function (S-F) mapping module manages the mappings between the session ID and the EPTP ID of the called function in a table, and the reference counting module records the number of references to each function. The functions that are not referenced by any session would be destroyed by the UaaF framework.

A. EPTP Lists

A EPTP list is a special memory page in the KVM kernel space, containing at most 512 EPTP entries that point to the EPT of other unikernels. For each session or function, if it has to call other unikernels via VMFUNC, UaaF would assign it an EPTP list. It determines how many other unikernels can be called by the session or function via the VMFUNC instruction. The EPTP list is loaded into the kernel when a session or function starts up. For each unikernel, we use a 64-bit field in the *VM Control* Structures (VMCS) to record the starting address of its EPTP list. Therefore, a unikernel can access the EPT of other unikernels via the VMFUNC instruction. When the unikernel terminates, UaaF should free the page of its EPTP list.

Figure 6 shows how a VM can switch its EPT to another VM's EPT through VMFUNC and the EPTP list. The GPA of

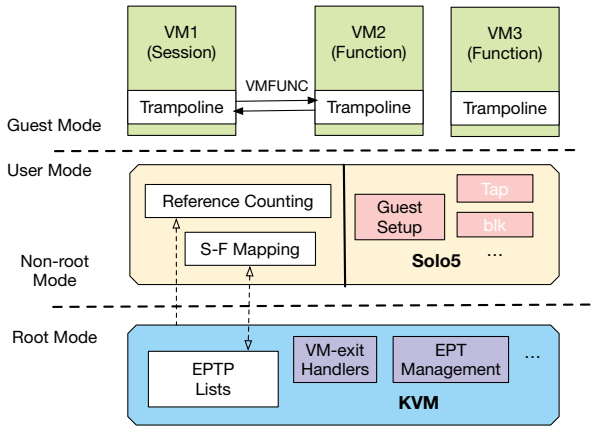


Fig. 5: The architecture of UaaF (The modules in the white rectangles are newly developed in UaaF, and Other modules exist in original KVM and Solo5.)

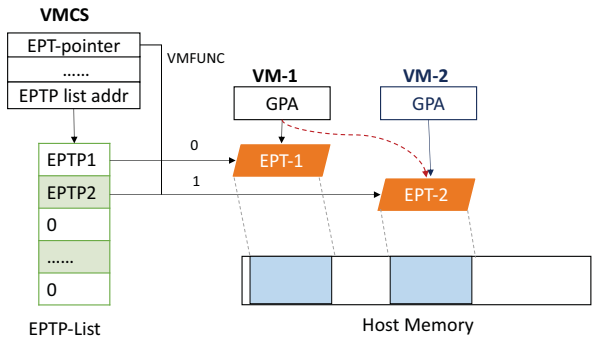


Fig. 6: EPT switching through VMFUNC

guest VM is translated to HPA by the EPT whose entry address is configured in the VMCS. As shown in Figure 6, EPTP1 and EPTP2 correspond to the EPTPs of VM-1 and VM-2, respectively. When VM-1 executes the VMFUNC instruction, its EPT table (EPT-1) is switched to EPT-2 which belongs to VM-2. In this way, VM-1 can call a function in VM-2 seamlessly, without suffering the cost of VM-exit.

The session-function (S-F) mapping module manages the caller-callee mappings between different unikernels in a table. When a unikernel registers in the UaaF, it submits a list of functions it would call later. The S-F mapping module records the caller and callees ID in the S-F mapping table. Actually, the callees ID is the index of callee's EPTP in the EPTP list. UaaF recursively analyses the calling relationship between different unikernels and fills all caller-callee entries in the mapping table.

The S-F mapping module provides an interface to configure and update each unikernel's EPTP lists in the kernel. When UaaF receives the first request of the user session, the S-F mapping module retrieves the ID of all functions that the session would call in the future. The called functions should start up ahead of the startup of the session. If the called functions are already launched, their EPTPs are written into the EPTP list of the newly launched session. At this time, when the session would call a function, it can get the callee's EPTP by retrieving the callee's ID in the mapping table. For the following request of the same user session, UaaF can simply clone another instance of the session, without changing the EPTP list.

The reference counting module is responsible for the startup and termination of unikernels. It maintains a reference count for each active function. When the reference count of a function becomes zero, this function would keep warm for a given period of time and then terminate.

B. Trampoline

To seamlessly invoke a function in another unikernel, UaaF should install the same trampoline code in both caller and callee unikernels to support EPTP-switching. A trampoline is a binary code that is loaded in the kernel space together with the unikernel image at startup.

First, we should guarantee that the trampoline code at the caller side can seamlessly jump to the callee side. Because the value of CPU instruction pointer does not change after executing the VMFUNC instruction, there is not a context switch between the caller and callee. More specifically, we should guarantee that the trampoline code is deployed in the same address space at both the caller and callee sides. Because unikernels are single-address-space images, we map the trampoline code to the same guest physical address space for the both caller and callee.

Second, we should carefully manage the program stack in the trampoline code. When a unikernel executes code in another unikernel, the trampoline maintains the program stack during the function invocation. The trampoline code should save the state of the caller stack, and then assign a valid *rsp* to the called unikernel. Finally, it restores the state of the caller after the called function executes. For each function, its code, global variables, and heap are accessible to other unikernels whose EPT has switched to it. However, its local variables are stored in a private stack for each caller. To limit the memory space consumed by the program stacks, we allow one unikernel to serve at most 250 callers empirically.

Code 1 shows the key function of the trampoline pseudocode written in assemble language. Line 2 clears the *eax* register, the value of zero indicates that the desired functionality of VMFUNC is EPTP-switching. Line 3 sets the *ecx* register to the target function ID, which is actual the index of the EPTP list in the kernel. In line 4, the VMFUNC instruction performs the EPTP switching to change the EPT whose index is the value of *ecx*. If the item in the EPTP list is valid, the trampoline in the caller transfers control to the trampoline in the callee. Line 6 and 7 check up on the key provided by the caller in the *rdi* register, and execute the next statement (line 9) if the caller has an access to the callee. Otherwise, the trampoline jumps to the label *RT*, which returns the source unikernel to run. Moreover, if the *target_id* is invalid or the called unikernel is not active, the VMFUNC fails to execute, and the trampoline also jumps to the label *RT*. Line 9 stores the stack pointer of the session, which would be restored when the

Code 1 An example of trampoline

```
1    ; switch EPTP
2    xor %eax, %eax
3    mov target_id, %ecx
4    VMFUNC
5    ; check the key (zero in sessions)
6    cmp %rdi, key
7    jnz RT
8    ; save rsp of the session
9    movq %rsp, session_stack
10   ; determine the target stack
11   callq STK_SWITCH
12   movq target_stack, %rsp
13   ; call target function
14   callq $TARGET_ENTRY
15   ; switch to session's stack
16   movq session_stack, %rsp
17   ; switch back
18   RT:
19   xor %eax, %eax
20   mov session_id, %ecx
21   VMFUNC
22   ret
```

called function returns. Line 11 calls *STK_SWITCH*, and then jumps to the stack of the target function according to its ID. Line 12 loads the target stack pointer into the *rsp* register. Line 14 calls the target function specified in the target trampoline code. Once it returns, the trampoline restores the stack pointer of the caller from the callers status page (line 16). In the end, the trampoline resets the EPT to the caller (lines 19-21) and leave the trampoline at the callee side (line 22).

C. Security Issues

Malicious VMFUNC. The VMFUNC instruction would introduce security hazards because it provides an efficient mechanism to share codes between unikernels. If a malicious unikernel has the access to another unikernel's memory, it could read and modify the heap and stack of the target unikernel. Because VMFUNC instruction operates in the VMX non-root mode, it is impossible to perform code verification in the VMX root mode. A recent work limits the VMFUNC operation in a trusted domain [22]. To mitigate security hazards of malicious VMFUNC, we can simply scan the code of unikernel images when it registers to UaaF, and forbid the images containing any VMFUNC instructions. Note that the normal VMFUNC instructions are only included in the trampoline code, which is dynamically loaded into the unikernels by UaaF at runtime. We deem that the trampoline is trustable because the trampolines are provided by UaaF and invisible to developers and users.

Function never returns. The called function may face some internal errors or do not return to the caller deliberately, allow the caller to suspend forever. To address this problem, UaaF terminates those hanging unikernels upon a timeout.

Untrusted function invocation. There are also security hazards of stack switching if the caller and callee distrust each other. Because UaaF stores all EPTPs that a unikernel depends on in a EPTP list, so that a unikernel can switch EPT to the registered functions and also other unikernels needed by those registered functions. Thus, a unikernel can provide an arbitrary function ID that it is going to call in its trampoline. To forbid these invalidated function invocations, UaaF needs to verify the key of each callee, and only permits the function invocation if the caller provides the correct key only assigned by UaaF, as described by the lines 6-7 in Code 1.

VI. Evaluation

In this section, we evaluate UaaF in terms of application startup latency, memory footprints, communication cost between unikernels, and performance overhead.

A. Experimental Setup

We evaluated UaaF on servers equipped with Intel Skylake Core i5-8700 CPU processors, which have 6 cores and 16 GB memory. The servers ran KVM Linux–Ubuntu 16.04 with kernel 4.15. We implemented UaaF based on Solo5 0.4.1 and OSv 0.54. In our evaluation, we measured the performance of UaaF on the following unikernels: MirageOS 3.7.1, Solo5-based unikernels, IncludeOS 0.15, OSv on Qemu 2.5, and OSv on Firecracker 0.20.

B. Startup Latency

First, we study how much UaaF can reduce the startup latency of serverless applications compared to ordinary unikernels. We used Solo5 as unikernels used in user sessions due to its lightweight design. The library *functions* were built by commonly-used unikernels such as IncludeOS and OSv. We first launched library *functions* and their dependent functions whose IDs have been registered in the Session-Function mapping table. We kept these library *functions* always in memory and then started the session functions. As such, a session can call functions through VMFUNC. Fig. 7 shows the startup latency of sessions and library functions in UaaF. Although the startup latency of library functions was approximately 1 second for most scenarios, UaaF preloaded these functions in memory so that front-end sessions can be promptly launched. As shown in Figure 7, UaaF can effectively limit application startup latency below 3 ms, 300 times faster than traditional unikernels. In addition, each library function can be concurrently invoked by multiple sessions.

Next, we study how the startup latency scales with the number of application instances. We gradually increased the number of instances of the same application and measured the startup latency of each instance. Fig. 8 shows that application startup latency increased with the number of instances in IncludeOS, OSv, and Docker because each instance launch required an new execution environment and no sharing was possible. Furthermore, the number of IncludeOS and OSv instances that can be launched was limited to 135 and 234, respectively, due to resource constraints. In contrast, in UaaF, each instance consisted of one session function and multiple library functions, which were shared among all instances. UaaF was able to scale to a large number of instances and its performance was similar to that in Solo5 because the session functions in UaaF use Solo5-based unikernels.

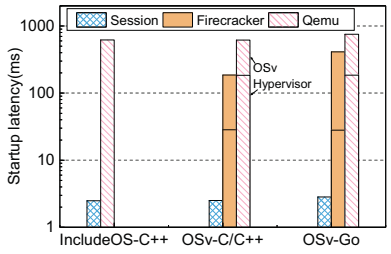


Fig. 7: The startup latency of sessions and functions in UaaF

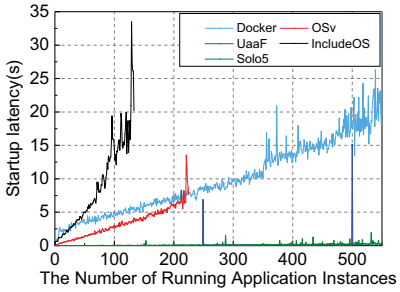


Fig. 8: The startup latency varies with the number of application instances

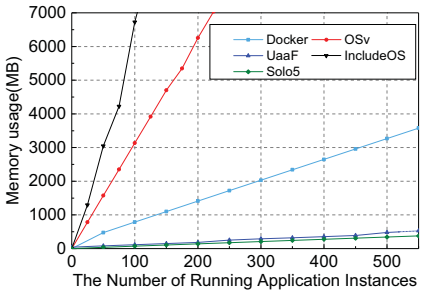


Fig. 9: The memory usage of different functions and containers

TABLE I: Inter-function communication latency using network

Linux Process	Unikernel	Container
3.94 ms	4.77 ms	5.08 ms

C. Memory Footprint

Fig. 9 shows the aggregate memory footprint of all application instances. As previously discussed, IncludeOS, OSv, and Docker do not allow instances to share execution environments. Therefore, their memory footprints increased linearly with the number of application instances. Since each instance required a full execution stack, the memory footprints grew very large. The memory usage of IncludeOS and OSv was even higher than that of Docker because all containers share the same OS kernel and only application stacks were duplicated. In UaaF, we used OSv intentionally, which is more heavyweight than Solo5, in session and library functions. As shown in the figure, UaaF achieved similar memory usage to Solo5 since all library functions were shared.

D. Communication Cost

The existing inter-domain communication between unikernels and containers mainly relies on the (virtual) network device. Communication latency is high since messages need to go through container's runtime security checks and the guest network stack. Table I shows inter-function communication via the network interface using native Linux processes, unikernels, and Docker containers. The performance of Linux *Inter-Process Communication* (IPC) serves as the baseline. While unikernels and containers incurred additional overhead relative to native Linux processes, their inter-function communication costs were comparable at the millisecond level. In comparison, the cost of function invocation mechanism in UaaF is as low as 0.5μs, three orders of magnitude lower than the Linux IPC.

E. Overhead

In UaaF, the shared library functions should load all executable programs and set up program stacks for later invocation in advance. Otherwise, the trampoline may call an invalid address. Thus, when a session calls a library function at the first time, it must wait to initialize the library function. Table II shows the average cost of VMFUNC and different KVM VMX Exit handlers. A successful execution of VMFUNC only takes 138 cycles. The trampoline can seamlessly jump to the target functions in another unikernel, and thus leads to no performance overhead. The *handle_vmfunc* takes about 1123 cycles if VMFUNC switches the EPTP to an invalid entry address. Since different types of unikernels are assigned with different CR3 values, if VMFUNC switches the EPTP to a guest CR3, the *handle_vmfunc* takes 32352 cycles (less than 0.5 ms). Since the values of EPTPs are stored in the kernel, it takes 2428 cycles to update the EPTP list.

TABLE II: Execution time of VMFUNC and different KVM VMX Exit handlers

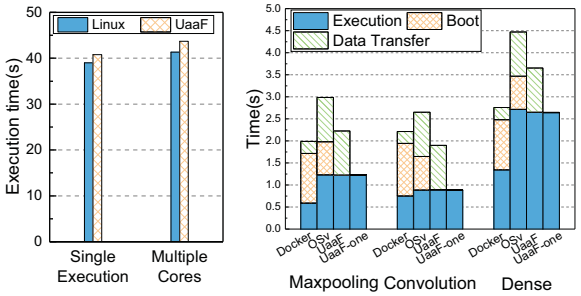
Operation	Cycles
VMFUNC	138
handle_vmfunc	32352 / 1123
handle_ept_violation	11284
handle_io (null)	5887
EPTP list updating	2428

F. Case Studies

In the following, we explore some application scenarios that can benefit from UaaF.

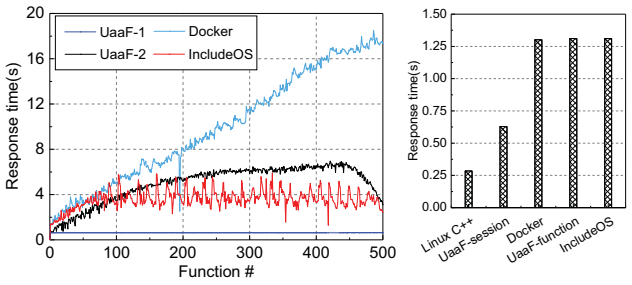
1) ***Symmetric Multi-processing****: First, UaaF is effective for parallel program deployment in multi-core machines*. Taking a machine learning application as an example, we implement a *K-Nearest Neighbour* (KNN) algorithm to recognize handwritten digits using MINST database. For KNN, the parameter "K" has a significant impact on the accuracy of classification. At first, we deploy the KNN algorithm in an unikernel which provides a *KNN_classifier* interface and also loads the image data. Second, other five unikernels are started to call the KNN function in the first unikernel with different "K" parameters. For comparison, we also run the program using six threads on the same machine in parallel .

Fig. 10(a) shows the execution time of the application in different models. The single-execute only runs *KNN_classifier* once with a single "K" parameter. In this case, the unikernel in UaaF spends a little more time than the native Linux due to the overhead of virtualization. When the number of executors increase to six, the execution time increases slightly for both Linux and UaaF. The differences are mainly attributed



(a) KNN execution time (b) CNN execution time

Fig. 10: Image processing



(a) High-frequent user requests (b) Single user request

Fig. 11: Function execution time

to the cost of synchronization between executors. However, UaaF offers strong isolation between different executors. UaaF needs about one second to deploy the unikernel containing the KNN codes, and then launch the other five executions (functions), while the performance gap between the native execution and UaaF only increases 0.57s. To this end, this experiment demonstrates that UaaF is efficient for deploying parallel applications in the cloud.

*2) **Stream Processing**: Second, UaaF is particularly useful for executing stream processing applications in the serverless computing model* [7]. Taking the *Convolutional Neural Network* (CNN) algorithm as an example, a typical image processing program usually contains three dimensions of operations–max-pooling, convolution, and dense, and are executed sequentially. We deploy the three components in three sandboxes built by OSv-based unikernels and Docker containers, respectively. In UaaF, we deploy each component in an independent OSv-based unikernel, and use a session to call the three functions sequentially. For comparison, we also deploy the three components in a single OSv-based unikernel (namely UaaF-one) to avoid data transferring between the three unikernels.

Fig. 10(b) shows the execution time of CNN components in different execution models. We find that the traditional schemes spend a large proportion of total execution time in the startup of Docker containers and OSv-based unikernels. Comparing to Docker, OSv spends more time in execution and data transferring due to the virtualization overhead. Because the functions have already booted before the sessions. UaaF only needs to boot the tiny user session in several milliseconds on-demand, and thus significantly reduce the startup latency of user applications. UaaF-one further reduces the execution time by reusing the intermediate data within a single unikernel. Overall, the best deployment of UaaF can improve the application performance by 55% compared with the traditional deployment using OSv.

*3) **Microservice Architecture**: Third, UaaF is able to support microservice efficiently.* Most microservices are lightweight and stateless. They can be processed in a very short period and then destroy the microservices immediately. UaaF is particularly suitable for serverless computing in a microservice architecture. Here, we take cryptographic hash computations as an example. We deploy the same hash function in Docker container, IncludeOS-based unikernel, and UaaF, respectively. Note that UaaF also uses IncludeOS to deploy the function. Fig. 11(b) shows the response time of a single request in different schemes. Docker, UaaF-function, and includeOS all need about 1.3s to process a request, most time is spent in booting the sandboxes. When we submit each user request in an interval of 200ms, the systems using Docker and IncludeOS become overloaded. As shown in Fig. 11(a) The response time of Docker is even as high as 18s after the 400th request. In contrast, UaaF shows very stable response time (0.6s) because the session startup latency is extremely low (several milliseconds), as presented by *UaaF-1*. However, when we shorten the request arrival time interval to 80ms, the response time also increases because the server load becomes high, as shown by *UaaF-2*.

VII. Related Work

Recently, unikernels have attracted increasing attentions because they offer strong isolation and even better performance than container [23]. Unikernels have demonstrated their advantages in the field of cloud/edge computing. USETL [24] exploits unikernel to deploy serverless *Extract-Transform-Load* (ETL) workloads. Unikernels are also cooperatively used with containers [25] to offer efficient software programming and deployment environments, and high isolation of software execution environments. However, there are still very few work on using unikernels for serverless computing. UaaF is a new FaaS framework exploiting a session-function decoupled programming model to provide lightweight and elastic serverless micro-services.

There have been a few studies on reducing the startup latency of unikernels. SOCK [18] sets up a cache of libraries and uses a python interpreter to only load necessary in-cache libraries into the installing functions. UaaF can share libraries in different functions for multiple user sessions at the same time, so there is no need to keep libraries “warm”. SAND [7] leverages a similar strategy to place fine-grained workers in the same sandbox and share it with multiple instances of the same application. In contrast, functions in UaaF are isolated in unikernels and shared in a more finer-grained model. Cntr [26] decreases the startup latency of containers by splitting the traditional container image into two fragments:

the "fat" fragment contains all application tools and are shared in a filesystem server, while the "slim" fragment only contains the main program. In contrast, UaaF abstracts an application into a session and multiple lightweight functions based on the application logic. Moreover, UaaF exploits a hardware feature VMFUNC to achieve high-performance communication between unikernels.

Recently, a few works have been proposed to leverage VMFUNC for cross-world calls (e.g., syscall, hypercall). CrossOver [27] is designed for general-purpose inter-domain communication in virtualization environments. Skybridge [22] leverages VMFUNC to improve the performance of inter-process calls in microkernels. They create a virtualization layer for processes to make use of VMFUNC. Hodor [28] is proposed to improve intra-process isolation, and also use VMFUNC to call shared libraries. Unlike those works, UaaF exploits VMFUNC to share application codes in lightweight unikernels for serverless computing.

VIII. Conclusions

In this paper, we rethink the architecture of FaaS and propose UaaF to reduce the startup latency of functions and the cost of function invocations. UaaF offers several remarkable advantages for serverless computing. First, unikernel is naturally suitable for serverless computing because it is extremely lightweight for low-latency startup of functions. Second, library functions can be shared by multiple user sessions concurrently. Thus, once a library function has been initialized, other callers do not have to wait for initializing the function. Third, user sessions can seamlessly invoke the library functions without suffering the performance penalty due to VM-exits and user-to-kernel mode switching. Fourth, the user session can releases resource immediately once the process terminates. Finally, UaaF can support many cloud applications that need to scale out immediately to serve a burst of user requests.

Acknowledge

This work is supported jointly by National Natural Science Foundation of China (NSFC) under grants No. 61672251, 61732010, 61825202, 61929103.

References

[1] AWS Lambda. https://aws.amazon.com/lambda.
[2] IBM Cloud Functions. https://www.ibm.com/cloud/functions.
[3] Azure Functions. https://azure.microsoft.com/en-us/services/functions.
[4] Google Cloud Functions. https://cloud.google.com/functions.
[5] gVisor. http://cloud.google.com/blog/products/gcp/open-sourcing-gvisor-a-sandboxed-container-runtime.
[6] Firecracker. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing.
[7] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," In *Proceedings of 2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 923–935.
[8] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, 2020, pp. 467–481.
[9] A. Madhavapeddy, and D. J. Scott, "Unikernels: Rise of the Virtual Library Operating System," *Communications of the ACM*, vol.57, no.1, pp. 61–69, January, 2014.
[10] A. Bratterud, A. A. Walla, H. Haugerud, P.E. Engelstad, and K. Begnum, "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services," in *Proceedings of 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 250–257.
[11] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarEl, D. Marti, and V. Zolotarov, "OSv–Optimizing the Operating System for Virtual Machines," in *Proceedings of 2014 USENIX Annual Technical Conference (ATC)*, 2014, pp. 61–72.
[12] D. Williams, and R. Koller, "Unikernel Monitors: Extending Minimalism Outside of the Box," in Proceedings of *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.
[13] Making OSv Run on Firecracker. http://blog.osv.io/blog/2019/04/19/making-OSv-run-on-firecracker.
[14] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as Processes," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 199–211.
[15] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-latency Video Processing Using Thousands of Tiny Threads," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 363–376.
[16] E. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The True Cost of Containing: A gVisor Case Study," in *Proceedings of 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.
[17] E. A. Brewer, "Kubernetes and the Path to Cloud Native," in *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, 2015, pp. 167–167.
[18] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2018, pp. 57–70.
[19] T. N. Bui. Benchmarking AWS Lambda runtimes in 2019 (part I). https://medium.com/the-theam-journey/benchmarking-aws-lambda-runtimes-in-2019-part-i-b1ee459a293d.
[20] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015, pp. 1–17.
[21] What is AWS Step Functions? http://docs.aws.amazon.com/step-functions/latest/dg/welcome.html.
[22] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "SkyBridge: Fast and Secure Inter-process Communication for Microkernels," in *Proceedings of the 14th EuroSys Conference (EuroSys)*, 2019, pp. 1–15.
[23] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) than Your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 218–233.
[24] H. Fingler, A. Akshintala, and Y. J. Rossbach, "USETL: Unikernels for Serverless Extract Transform and Load Why Should You Settle for Less?" in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2019, pp. 23–30.
[25] Z. Shen, Z. Sun, G. E. Sela, E. Bagdasaryan, C. Delimitrou, R. V. Renesse, and H. Weatherspoon, "X-Containers: Breaking Down Varriers to Improve Performance and Isolation of Cloud-native Containers," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 121–135.
[26] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS Containers," in *Proceedings of 2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 199–212.
[27] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan, "Reducing World Switches in Virtualized Environment with Flexible Cross-World Calls," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture(ISCA)*, 2015, pp. 375–387.
[28] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process Isolation for High-throughput Data Plane Libraries," in *Proceedings of 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 489–504.