Serpens: A High Performance FaaS Platform for Network Functions

Heng Yu, Han Zhang, *Member, IEEE*, Junxian Shen, Yantao Geng, Jilong Wang, *Member, IEEE*, Congcong Miao, and Mingwei Xu, *Senior Member, IEEE*

***Abstract*—More and more enterprises deploy applications on Function-as-a-Service (FaaS) platforms to improve resource efficiency and save monetary costs. Network Functions (NFs) suffer from staggered peaks of traffic patterns and could benefit from fine-grained resource multiplexing in FaaS platform. However, naively exploring existing FaaS platforms to support NFs can introduce significant performance overheads in three aspects, including slow instance startup, remote state access for NFs, and costly packet delivery between NFs. To address these problems, we propose Serpens, a high performance FaaS platform for NFs. First, Serpens proposes a reusable NF runtime design to slash instance startup overhead. Second, Serpens designs a novel state management mechanism to support local state access. Third, Serpens introduces an advanced service chaining approach to avoid extra packet delivery. Besides, Serpens designs an NF scaling mechanism to minimize performance fluctuation. We have implemented a prototype of Serpens and conducted comprehensive experiments. Compared with the NFs and Service Function Chains (SFCs) that run on existing FaaS platforms, Serpens can improve the throughput by more than 10× and reduce the latency by more than 90%.**

***Index Terms*—FaaS Platform, high performance, network function.**

I. Introduction

NETWORK Function Virtualization (NFV) was introduced to address the limitations of traditional proprietary middleboxes. NFV realizes network functions (NFs) in software and runs them on cloud platforms rather than various dedicated hardware to reduce expenses and simplify management [1], [2], [3]. Currently, both enterprises and telecommunications are increasingly adopting the NFV technology to evolve their network environments [4], [5].

Nowadays, more and more enterprises run their applications on FaaS platforms to improve resource efficiency and save costs [6], [7]. FaaS platform takes the responsibility of resource management and automatically runs functions anywhere with enough resources when receiving requests [8]. Besides, users only provide their applications as a set of functions, and pay for what they use at very fine time and resource granularity [8].

Recently, there is a tendency of outsourcing NFs to FaaS platforms [9], [10]. Since NFs suffer from staggered peaks of traffic patterns, compared with VM or container-based cloud platforms, realizing NFs on FaaS platforms could benefit from fine-grained resource multiplexing and completely release the management burden of users. However, existing FaaS platforms have the following three limitations, which reveals two orders of magnitude performance loss compared with conventional NFV platforms [2], [11], [12] and prohibits large-scale deployment of NFs in practice.

The first limitation relates to slow instance startup. In reality, user packets can arrive at any time. To guarantee the performance of NFs, FaaS platforms require dynamically starting instances to handle varying packet rates. However, most FaaS platforms suffer from large cold start overheads, i.e., instances would take a long time to start (e.g., 56.98 ms [13]), and the large startup latency could hurt the performance of NFs. The second limitation relates to remote state access. FaaS functions are stateless and FaaS platforms always store the necessary states in a remote storage module (e.g., Redis [14]). However, this design can worsen the latency for NFs, since massive packets have to access remote storage before they can be further processed. The third limitation relates to costly packet delivery. FaaS functions are non-addressable and they utilize a proxy module (e.g., Kafka [15]) to communicate with each other. This design is suitable for short-lived functions which are frequently created and destroyed but can add extra latency for SFCs, since numerous packets need to be delivered by the proxy between every two NFs.

To address above problems, we propose Serpens, a high performance FaaS platform for NFs. Serpens has four novel designs: First, we propose a reusable NF runtime design that advocates decoupling instance into NF logic and general runtime, and it utilizes memory mapping to achieve fast instance startup.

Manuscript received 14 March 2022; revised 8 March 2023; accepted 15 March 2023. Date of publication 30 March 2023; date of current version 6 July 2023. This work was supported in part by the National Key Research and Development Program of China under Grant 2020YFE0200500, and in part by National Natural Science Foundation of China under Grant 62002009. Recommended for acceptance by K. Gopalan. *(Corresponding author: Han Zhang.)*

Heng Yu and Han Zhang are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Zhongguancun Laboratory, Beijing 100190, China (e-mail: yuheng@mail.zgclab.edu.cn; zhhan@tsinghua.edu.cn).

Junxian Shen and Yantao Geng are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China (e-mail: shenjx22@mails.tsinghua.edu.cn; gyt22@mails.tsinghua.edu.cn).

Jilong Wang and Mingwei Xu are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, also with the Zhongguancun Laboratory, Beijing, P. R. China, and also with the Quan Cheng Laboratory, Jinan 250103, China (e-mail: wjl@cernet.edu.cn; xumw@tsinghua.edu.cn).

Congcong Miao is with the Tencent, Beijing 100193, China (e-mail: mccmiao@163.com).

Digital Object Identifier 10.1109/TPDS.2023.3263272

Second, we design a novel state management mechanism that persists state in a long-running daemon to support local state access with shared memory for all NFs. Third, we propose an advanced service chaining approach that decouples resource unit and NF execution, and it utilizes function call to avoid extra packet delivery in an SFC. Fourth, we design an elastic NF scaling mechanism to minimize performance fluctuation and provide a programming abstraction to mitigate NF development as well as SFC chaining efforts. Compared with existing FaaS platforms [10], [20], [21], Serpens has two advantages. On the one hand, Serpens is specifically designed for NFs, while most platforms (e.g., Fission [20], SAND [21]) are designed for general applications and could incur high performance overhead when directly supporting NFs. On the other hand, Serpens considers instance startup, state management, packet delivery and NF scaling, which is more comprehensive and practical than existing works (e.g., SNF [10]). Thus, this paper mainly makes the following *contributions*:

- We present the motivation of providing FaaS platform for NFs and elaborate on the performance problem of existing FaaS platforms to support NFs (Section II).
- We design Serpens, a high performance FaaS platform for NFs, including reusable NF runtime to slash instance startup overhead, a novel state management mechanism to support local state access and an advanced service chaining approach to avoid extra packet delivery. Besides, we design an elastic NF scaling mechanism to minimize performance fluctuation (Section III).
- We have implemented a prototype of Serpens, which provides a programming abstraction to mitigate NF and SFC development (Section IV). Compared with the NFs and SFCs that run on existing FaaS platforms (e.g., Fission [20]), Serpens can improve the throughput by more than 10× and reduce the latency by more than 90% (Section V).

The article is organized as follows. Section II shows the background and motivation. Section III introduces the designs of Serpens. Section IV describes Serpens prototype. Section V shows the evaluations. Section VI shows the discussions. The related works are shown in Sections VII. Finally, Section VIII concludes the article.

II. Background and Motivation

In this section, we first introduce the primary benefits of supporting NFs with FaaS platform. Then, we thoroughly analyze the performance problem of existing FaaS platforms when supporting NFs.

A. The Primary Benefits of Supporting NFs With FaaS Platform

To illustrate the necessity of supporting NFs with FaaS platform, we first describe recent tendency of running NFs with NFV. Then, we introduce the evolution of cloud computing. Finally, we sketch the main advantages of supporting NFs with FaaS platform.

1) Running NFs With NFV: The Internet is growing exponentially with ever increasing number of devices and users being connected to it, along with an exploding demand for various resource and performance intensive network services [22], [23]. The last few years have seen an explosive increase in interest of adopting the NFV technology in both enterprise and telecommunication environments to address the limitations of traditional proprietary middleboxes [5]. The NFV decouples NFs from various dedicated hardware middleboxes by realizing them in software and running on cloud platforms [1]. The software-based approach is able to reduce the expenses of network devices as well as improve the flexibility of network management [2], [3], [24]. For example, AT&T announces an ambitious plan of offloading the whole 5 G network into Microsoft's cloud, which includes deploying massive NFs on the cloud platform [4].

2) Evolution of Cloud Computing: Existing cloud platforms can be divided into four categories [6]: (1) Bare machine, such as SNS [16]; (2) Virtual machine, such as OpenStack [17]; (3) Container, such as Kubernetes [18]; (4) FaaS, such as Knative [19]. Table I shows the general comparison of existing platforms. Benefiting from the virtualization technology, more and more applications are deployed on the VM and container platforms. To run applications on such platforms, users need to exclusively monopolize a fixed amount of hardware resources (CPUs and memory) at a coarse granularity (minutes or hours) into VMs or containers [25]. However, this model can lead to two problems: (1) resource inefficiency. The request rate in real-world applications keeps drastically fluctuating from near zero to O(1 M) requests per second [26], [27]. Application owners usually over-provision resources to meet the peak workload demands. However, this method can waste a large amount of resources, since the request rate is far less than the peak load most of the time [28]. A better method is re-provisioning resources according to workload demands at regular intervals. However, due to the sharing level and virtualization granularity in VM and container platforms [6], resources are usually allocated at a minimum of minutes. Re-provisioning need to meet the peak workload of the next interval, which still wastes resources [29]. (2) management challenge. The platform often offers dozens of VM or container types with subtly different hardware configurations. Meanwhile, the performance of different applications is sensitive to different types of hardware resources [30]. Thus, it is challenging to find the most suitable VM or container types to deploy and scale various applications. Besides, application owners need to manage these VMs and containers themselves and handle various failures.

To multiplex resources and simplify management for users, a recently thriving computing model named *FaaS*, also known as *serverless computing*, is emerging as an essential evolution in cloud platform [6], [7]. Nowadays, many FaaS platforms are already available, both from commercial enterprises [31], [32], [33], [34] and open-source projects [19], [20], [35], [36]. First, this model shifts the responsibility of resources management to FaaS platform and provides the abstraction of *function*. Benefiting from the runtime-level sharing and fine-grained virtualization [6], functions can run anywhere with enough resources and accommodate efficient resource provision. Second, users only

TABLE I
A GENERAL COMPARISON BETWEEN FAAS AND OTHER CLOUD PLATFORMS

Category of Platform	Typical Platform	Sharing Level	Virtualization Granularity	Resource Efficiency	Application Deployment	Application Scalability	Application Management	Monetary Cost
Bare machine	SNS [16]	None	Coarsest	Lowest	Hardest	Worst	Hardest	Highest
Virtual machine	OpenStack [17]	Host	Coarse	Low	Hard	Bad	Hard	High
Container	Kubernetes [18]	OS	Fine	High	Easy	Good	Easy	Low
FaaS	Knative [19]	Runtime	Finest	Highest	Easiest	Best	Easiest	Lowest

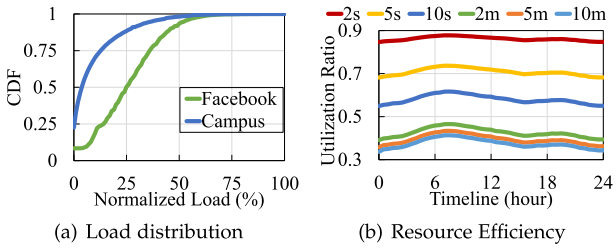


Fig. 1. Traffic patterns of two large-scale networks.

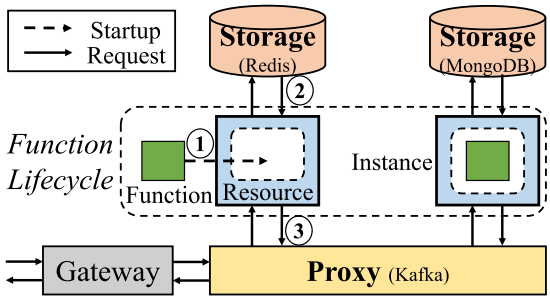


Fig. 2. Framework of typical FaaS platforms.

need to provide their applications as a set of functions, and the FaaS platform automatically deploys and scales these functions with an ephemeral execution environment according to dynamic workloads [7], [8]. Thus, users can be completely released from the application management. Besides, users only pay for what they use at very fine time and resource granularity (e.g., 100 ms) and can therefore save more monetary cost [6], [7], [8].

3) Supporting NFs With FaaS Platform: Attracted by the benefits of FaaS, many applications have been migrated to FaaS platforms, such as distributed computing [37], video processing [38], and software compilation [39]. Here, we argue the necessity of supporting NFs with FaaS platforms rather than conventional NFV platforms from the resource efficiency and NF management aspects. First, the heavy-tail phenomenon in network traffic has been observed in various studies and is of great importance for both traffic characteristics analysis and network capacity planning [40]. Our analysis of the traffic from Facebook [41] and a campus network shown in Fig. 1(a) also proves this, where the load is under 40% at 80% of the time. The staggered peaks of traffic patterns demand fine-grained resource multiplexing and we also analyze how the granularity of hardware resources affects their efficiency. Fig. 1(b) shows the resource utilization of Facebook over a day with different resource granularity to meet the peak resource demand, and it illustrates that more fine-grained resource multiplexing can effectively improve resource efficiency. For example, the resource utilization of NFs can be improved by an average of 30% from the minute-level resource granularity (e.g., VM and container) to the second-level resource granularity (e.g., FaaS). Second, the performance tuning of NFs and SFCs is difficult and time-consuming. To achieve high performance, merely choosing the right CPU cores to run NFs in an SFC is non-trivial [42], let alone other hardware details and traffic characteristics. However, realizing NFs over FaaS platform can completely release the management burden, since NF owners are never responsible for deploying and scaling NF instances according to traffic variation. In fact, outsourcing NFs to FaaS platforms is recently becoming a new tendency and some efforts are progressing in the context of 5 G networks [43].

B. The Performance Problem of Running NFs on FaaS Platforms

1) Running NFs on FaaS Platforms: To enable fine-grained resource multiplexing for applications, all functions are *short-lived* and only occupy resources on demand. If no request arrives, functions need to free occupied resources and be destroyed. However, short-lived functions lead to three key features that are completely different from conventional applications: (1) Startup. Conventional applications are started on the deployment, while FaaS platforms start function instances when receiving requests. (2) Stateless. FaaS functions are stateless since the memory of the function instance needs to be freed on destroy. (3) Non-addressable. FaaS functions are non-addressable since the address of the function instance needs to be recycled on destroy.

Compared with traditional cloud platforms, the three features bring essential evolution for FaaS platforms. Fig. 2 shows a typical framework of FaaS platforms, which reflects key characteristics of both open-source FaaS platforms(e.g., [20], [21], [35]) and commercial platforms(e.g., [31], [32], [33]). In addition to the traditional components in cloud platforms (e.g., Gateway), the FaaS platform mainly introduces three modules: (1) Startup module. According to workload variation, the FaaS platform dynamically allocates resources and starts instances. (2) Storage module. To maintain state across instance destroy, FaaS function stores all persistent states into a storage module (e.g., Redis [14]). (3) Proxy module. To provide address even without instance, FaaS function utilizes a proxy module

TABLE II
LATENCY COMPARISON WHEN RUNNING FIREWALL ON STATE-OF-THE-ART FAAS PLATFORMS. ALL DATA COME FROM OUR RE-IMPLEMENTATION ON A LOCAL SERVER CLUSTER

Packet Processing	Instance Startup	State Access	Packet Delivery
2.95μs	56.98ms	90.55μs	23.01μs

(e.g., Kafka [15]) to accommodate suddenly arrived requests. Essentially, the FaaS platform relies on long-running modules to overcome the shortcoming of short-lived functions.

When running NFs on FaaS platforms, the lifecycle of NFs is as follows: On the deployment of NFs, users need to provide their NFs as a set of functions and register corresponding packets to trigger these NFs. The FaaS platform records these packet-NF pairs in Gateway for future mapping. On receiving packets, the FaaS platforms start function instances on allocated resources. These instances get packets from the proxy and access states from the storage. After handling packets, these instances store states in the storage and return packets to the proxy. To improve resource efficiency, the FaaS platform usually treats a batch of packets (e.g., 32) as a request and dynamically starts instances to provide varying processing capacities. Besides, the platform could destroy all instances of an NF if no packet arrives for a period of time (e.g., 30 seconds).

Although many FaaS platforms are available today, they are designed for general applications and could incur high performance overhead when supporting NFs [8], [9]. For example, we run a common SFC (Firewall, Monitor, NAT) on a popular platform (e.g., Fission [20]), whose setup is shown in Section V. The throughput of the SFC is only 82Kpps while the latency can be up to 3.2 ms. However, the throughput of conventional NFV platforms can often reach Mpps and latency is usually hundreds of microseconds [2], [11], [12], which is much better than existing FaaS platforms. To understand the performance problem, we analyze the main mechanisms of FaaS platforms [20], [35] and rewrite them to support packet processing. For convenience and generality, we take Firewall as a typical NF example to illustrate the performance problem. However, our basic conclusions remain the same for other NFs. Through exhaustively measuring the latency of packet processing workflow, as shown in Table II, we identify three main performance overheads.

2) Slow Instance Startup: Each instance usually runs NF as a process inside a sandbox. To handle time-varying packet rates, FaaS platform needs to dynamically start instances. This process is known as *cold start* and many research works have explored the optimization of instance startup. According to the technique route, existing research works can be classified into three categories: (1) Optimization from specific sandbox [13], [44], [45], [46]. For example, Catalyzer [13] utilizes the snapshot technique, which is provided by the sandbox, to accelerate memory recovery. However, they only focus on one type of sandbox and have low compatibility. (2) Optimization with specialized environment [29], [47], [48], [49]. For example, SEUSS [47] utilizes a library kernel [50], which can be customized for NF, to mitigate most startup overhead. However, they rely on specialized execution environments and are hardly deployed at a large scale. (3) Optimization with proactive startup [21], [51]. For example, SAND [21] proactively starts an instance for each function. However, they are hard to scale to thousands of functions in one server due to the high resource overhead. Besides, even the state-of-the-art research work (e.g., Catalyzer [13]) still has a significant performance overhead. As shown in Table II, the instance startup could consume 56.98 ms and is 19315$\times$ of the packet processing time.

3) Remote State Access: Many NFs are stateful and need to maintain persistent states for correct packet processing [24], [52], [53]. To support stateful NFs, existing FaaS platforms decouple the states that NFs need to maintain from NF logic, and store these states in a remote storage [54]. In this way, even if all instances of an NF are destroyed, the state can be persisted for subsequent packet processing. However, remote state access could introduce significant overhead, especially for NFs that require state access for every packet. As shown in Table II, the state access latency could reach 90.55 μs and is 31$\times$ of the packet processing time. Although some recent research works (e.g., SNF [10]) propose to provide local state access for per-flow state, they still cannot reduce the access latency for global state.

4) Costly Packet Delivery: In NFV, multiple NFs are usually chained together to form an SFC [2], [3], [12]. To chain multiple NFs, existing FaaS platforms rely on a proxy to identify the next NF and deliver packets according to SFC requirements. In this way, even if all instances of an NF are destroyed, the packet can be temporarily accommodated on the proxy. However, introducing a proxy between every two NFs can incur significant overhead, especially for global proxy that aims to provide unified load balance. Some recent research works (e.g., SAND [21]) propose to introduce a local proxy in each server to avoid the packet delivery overhead between servers. As shown in Table II, even chaining two Firewalls with a local proxy, the packet delivery latency could be 23.01 μs and is 8$\times$ of the packet processing time.

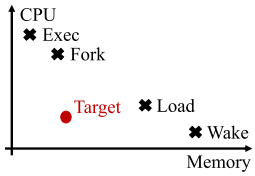
Summary and Root Cause Analysis. All of above mechanisms are designed for general computing and could efficiently support lots of applications, such as video processing [38] and distributed computing [37]. However, as shown in Table II, they incur high performance overhead when directly supporting NFs. This is because most NFs are typical *narrow tasks* that usually finish in a short time. Thus, the actual processing time for a batch of packets only occupies a very small portion of the total running time. Based on this observation, we propose a new FaaS platform specially designed for NFs. This platform keeps the feature of short-lived functions and therefore inherits the benefit of FaaS on resource efficiency and NF management, but adopts completely different designs on above three aspects to address the performance problem.

III. Serpens Design

In this section, we first introduce the designs of Serpens on instance startup, state management and service chaining to address the performance problem. Then, we describe the design on NF scaling to minimize performance fluctuation.

	Non-load Binary File	*Pre-load Binary File*
Non-initialize Process Runtime	Exec (Cold Start)	Fork (SAND)
Pre-initialize Process Runtime	Load (SOCK)	Wake (Warm Start)

(a) Different initial states and remaining operations for process startup



(b) Design space on CPU and memory trade off

Fig. 3. Comparison of different startup approaches.

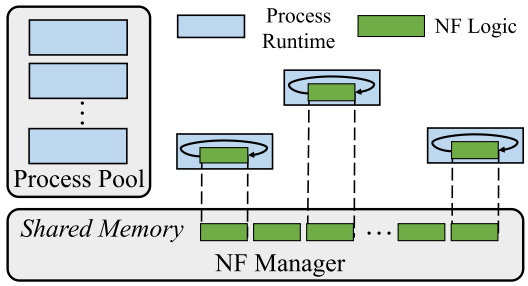


Fig. 4. Design for reusable NF runtime.

A. Fast Instance Startup

Although instance startup only happens when the packet arrival rate changes, it could severely jeopardize user experience [21], [51]. As mentioned in Section II-B2, existing research works can be incompatible with sandbox [13], [44], [45], [46], or rely on specialized environment [29], [47], [48], [49], while we adopt the technical route of category 3, which focuses on the design of general software stack. Since recent instance startup mechanisms [13], [45] pre-boot a sandbox pool to completely eliminate sandbox startup overhead, we mainly focus on the optimization of process startup, which still incurs unacceptable startup latency.

Reusable NF Runtime. Existing process startup methods are trade-offs between CPU and memory resources as cold start wastes more CPU resources on process startup for more efficient memory multiplexing [6], [55]. In fact, process startup can be further divided into two steps: (1) loading binary file into memory. (2) initializing process runtime for execution. These two steps can be performed independently, which forms four startup methods in current system. Fig. 3(a) shows the four methods with their initial states and remaining operations. In addition to the frequently used cold start and warm start [55], we can pre-load the binary file of NF and *fork* a new process to execute it on demand (e.g., SAND [21]) or pre-initialize the process runtime for execution and *load* the binary file of NF on demand (e.g., SOCK [51]). Fig. 3(b) shows the design space on the trade-off between CPU and memory for process startup. All above startup methods are simply trading memory for CPU to reduce cold start overhead. However, our target is approaching the bottom left corner of the design space and enabling fast startup with little memory overhead.

The key observation to achieve our target is that *not all NFs inside a server become active at the same time*. First, most NFs in FaaS platform experience sporadic packets and each server can hold thousands of NFs [13], [56]. Second, bursty packet arrival pattern determines only partial NFs could become active simultaneously [40]. Third, even massive NFs become active, FaaS platform could distribute them to different servers to avoid hardware resource overload [57]. Based on this observation, our key idea is designing a *reusable NF runtime*. We *decouple process into NF logic and general runtime, and enable statistical multiplexing among NF runtimes*. As shown in Fig. 4, we pre-load binary files of all NFs and manage these individual NF logics with an *NF Manager*. Besides, we maintain a *Process Pool* with sufficient pre-launched processes, each of which provides a general NF runtime. These processes pre-initialize the execution environment (e.g., delivering packets) and wait for wake-up signal (e.g., calling sigwait()). Once an NF instance needs to start, we can quickly map the memory of NF logic to a runtime and wake up the process.

However, these NFs are loaded to the address space of NF Manager and can not be accessed by the process in Process Pool. To enable statistical multiplexing among NF runtimes, we *move these NFs to shared memory* and any process can access these NFs for on-demand execution. For the NFs running in containers or VMs, we can leverage existing technologies [2], [58] to provide shared memory between them. Since the NF logic can run in any process runtime, the number of processes launched in Process Pool can be much smaller (e.g., $0.1\times$) than the number of NFs loaded in NF Manager, and only need to catch the maximum concurrently running processes (e.g., 100). As a result, deploying an NF only occupies several KB (several MB in warm start) and starting an instance only consumes tens of microseconds (tens of milliseconds in cold start), which could significantly accelerate cold start with little overhead.

NF Startup With Reusable Runtime is Practical. Since most NFs are designed for high performance and written in C/C++ languages [2], [11], [12], [24], [52], [53], [59], [60], compared with existing startup methods [13], [45], [46] that designed for general applications, we focus on the process startup in C/C++ environments. Nevertheless, supporting quick NF startup within any process runtime poses lots of challenges. To support incremental deployment, we compile each NF into a dynamic library and load them with default system loading tool (e.g., dlopen()). However, this tool cannot directly load NFs into shared memory [61], [62], [63]. A straightforward method is loading NFs into the default memory region, then copying them to the shared memory. Unfortunately, this method can not work because of two aspects: (1) library dependency; (2) absolute address. First, NFs may have dependencies (e.g., system libraries) and we need to copy their dependencies into shared memory. However, these dependencies are recursively loaded and shared across NFs, we can not get their accurate address range. Second, partial symbols in binary files are accessed through absolute addresses after being loaded into memory (e.g., NF name). However, if we copy NFs into shared memory, the addresses of these symbols would change and can not be accessed correctly with previous addresses.

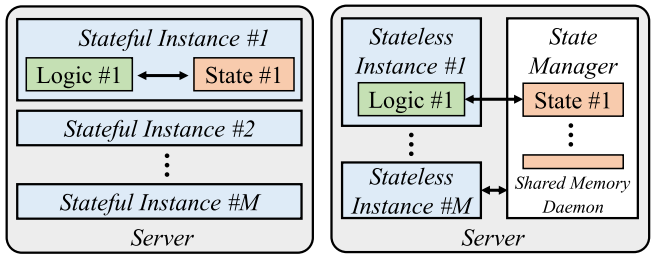
To address above challenges, we design a new loading tool to directly load NFs in shared memory. This tool inherits most

techniques (e.g., symbol identification) from the default loading tool (e.g., dlopen()) and does not require any modification to the Linux environment. It maintains a separate memory region for each loaded NF and mainly includes two steps: (1) copying binary files of the NF and its dependencies into shared memory. (2) relocating the addresses of partial symbols in binary files for correct access. Then, NFs can run correctly in shared memory within NF Manager. Unfortunately, the memory of these NFs and their dependencies need to be further mapped into process runtimes in Process Pool for execution. To provide NF isolation for memory mapping, we separate the memory regions of different NFs and load their dependencies independently. However, this method can repeatedly load some system libraries that have been loaded into the process by default (e.g., glibc) and waste lots of memory resources. Meanwhile, after the NF memory is mapped into another process, the address range of this NF changes and partial symbols that are accessed through absolute addresses would not be accessed again. However, we can not simply relocate these symbols with a correct absolute address since the NF may be mapped into multiple processes for parallel execution with different address ranges.

To utilize the system libraries that have been loaded into process runtime, we collect the locations of these system libraries when launching process and directly jump to these locations when calling libraries. However, if the Address Space Layout Randomization (ASLR) [64] is enabled, the locations of these system libraries change in different processes. To be compatible with ASLR, we adopt an indirect jump mechanism. First, we collect these locations in a table and allocate a fixed address to this table for simple access. Then, any library call can access this table to get the correct location of the library and jump to different locations across processes. To enable parallel execution for an NF in multiple processes, we reserve a rarely used address range for shared memory. Since the address space of a 64-bit system is almost infinite, we can reserve a huge address range for thousands of NFs. Then we map the memory of NFs to the same address range across all processes and these NFs could run correctly in any process runtime.

B. Novel State Management

As mentioned in Section II-B3, stateful NFs could suffer from significant performance overhead due to remote state access in FaaS platforms. A straightforward solution for this problem is *state localization*, i.e., co-locating NF states and logics within the same instances [24]. Most NFV platforms (e.g., OpenNF [24] and S6 [53]) adopt this method to design their state management mechanisms. As shown in Fig. 5(a), this method stores states inside instances for local and efficient access. Unfortunately, to improve resources efficiency, FaaS instances are designed to be short-lived [8], which implies that *all* NF instances would be destroyed if no packet arrives for a period of time (e.g., 30 seconds [35]). When an instance is destroyed, existing NFV platforms (e.g., OpenNF [24] and S6 [53]) can migrate their states to other alive instances. However, they cannot handle the situation that *all* instances are destroyed (scale to zero), which *loses* related states and happens often in FaaS platform since most NFs experience sporadic packets [56]. Thus, this method cannot be directly adopted in FaaS platforms to support stateful NFs.



(a) State management in NFV (b) State management in Serpens

Fig. 5. State management for FaaS platform.

To address this problem, we propose to *decouple the lifecycle of instances and states*. Our observation is that the loss of state is caused by the shared lifecycle of instances and states. When all instances are destroyed, the loss of state is inevitable. Instead of storing states within instances, we move the states of NFs into a specially designed *State Manager*, which is a long-running daemon. However, different from existing NFV platforms (e.g., StatelessNF [52] and CHC [65]), which introduce an external store for states and related state access is done through the *store*, Serpens runs a state manager in each server and NF instances can directly access these states through *shared memory*. Therefore, we can simultaneously achieve state persistence and localization. Fig. 5(b) shows the state management mechanism in Serpens. Since the states of NFs are stored in the shared memory of server daemon, each stateless instance can directly and efficiently access these local states. Besides, newly launched instances could correctly process subsequent packets, even if all previous instances have been destroyed.

Above state localization can effectively accelerate per-flow state access. However, NFs with global states, such as the traffic volume counter that should be maintained by all Monitor instances [24], still suffer from high performance overhead due to cross-instance state synchronization. The essence of the global state problem is that these states require frequent remote access. However, existing state management mechanisms in NFV platforms (e.g., OpenNF [24], StatelessNF [52], S6 [53] and CHC [65]) have not completely solved this problem. Although some recent research works (e.g., S6 and CHC) have made great process in global state handling, i.e., caching for non-partitioned (cross-flow) read-heavy states in S6 (CHC), they still have some states, i.e., non-partitioned (cross-flow) read/write states in S6 (CHC), need to be accessed remotely. In response, we propose to localize all states, including global state (non-partitioned states in S6 or cross-flow states in CHC), by slightly relaxing synchronization strictness.

Localizing all States. To understand the states in NFs, we thoroughly study common NFs and classify their states into four categories according to state scope (e.g., per-flow or global) and access pattern (e.g., read or write): (i) per-flow/read, (ii) per-flow/write, (iii) global/read, and (iv) global/write. Table III shows the states of some frequently used NFs (derived from [53],

TABLE III
STATE SCOPE AND ACCESS PATTERN OF COMMON NFS

NF Name	State Description	State Scope	Access Pattern
Firewall	Connection black/white lists	Per-flow	R
Traffic Monitor	Connection statistics	Per-flow	W
	Host statistics	Global	W
Flow Compressor	Compression rules	Per-flow	R
Load Balancer	Flow-server mappings	Per-flow	W
	Server usage statistics	Global	W
IPSec	Security associations	Per-flow	R
NAT	Flow-IP/Port mappings	Per-flow	W
	Pool of IPs	Global	W
VPN	Key/tunnel configures	Per-flow	R
IDS/IPS	Connection analysis objects	Per-flow	W
	Match patterns	Global	R

[66]). For states in categories (i) and (ii), it can be easily localized since packets in a flow are always sent to the same instance and access the same shared memory. State in category (iii) almost never changes after instance startup. Therefore, we can maintain a copy of this state in each instance, which localizes the global/read state. For category (iv), multiple instances need to update the global/write state simultaneously, which requires consistency guarantee. A straightforward method is adopting synchronization primitives (e.g., locks). However, it could compromise the NF performance. To address this challenge, we propose to trade state consistency for higher performance and provide different consistency guarantees according to NF requirements.

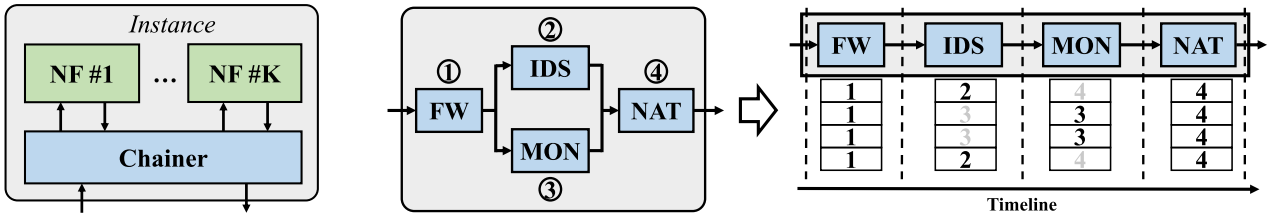
Specifically, we further divide global/write state into two types, including *single-writer state* and *multi-writer state*. For the single-writer state, it only allows one writer to update the state at any time. For example, the pool of IPs/Ports in NAT is a typical single-writer state, which does not allow multiple instances to update the state simultaneously in order to avoid different flows being mapped to the same IP/Port. For this type of state, providing *release consistency* [67] is sufficient. We separate such state into multiple *sub-states* and allocate each sub-state to one instance. Taking NAT as an example. For a pool of 100 IPs/Ports, we separate it into two sub-pools and each pool has 50 IPs/Ports. Each NAT instance monopolizes one sub-pool, which completely eliminates the necessity for two instances to compete for shared state. For the multi-writer state, it allows multiple instances to update the state at the same time under the condition of copying. The host statistics (e.g., the traffic volume of a unique IP address) in Monitor is a typical multiple-writer state, since multiple instances could collect their own host statistics simultaneously. For this type of state, we propose to guarantee *eventual consistency* [68]. Specifically, we copy the state into multiple *replications* and assign each replication to an instance. For example, to maintain statistics of 100 hosts in two Monitor instances, each instance maintains local host statistics, and we can periodically merge these replications to obtain final statistics.

However, completely localizing the global/write state may violate the correctness of packet processing. For example, an instance of NAT could exhaust its local pool of IPs/Ports and refuses new connections while other instances have enough IPs/Ports, which raises the requirement of dynamical state update across instances. We provide two ways to update the local state of global/write state. The first one is periodical update. For example, we could periodically reallocate the remaining IPs/Ports across multiple NAT instances. The second one is active update. For example, a NAT instance could actively trigger the reallocation of remaining IPs/Ports across multiple NAT instances after exhausting its local IPs/Ports.

Compared with recent state management mechanisms in NFV platforms (e.g., S6 [53] and CHC [65]), Serpens differs mainly in two aspects: (1) Serpens introduces a state manager in each server to support local state access as well as store related states in case of all instances of an NF are destroyed. (2) Serpens proposes a practical localization solution for global/write state (e.g., state split and update method). Although S6 and CHC carefully classify existing states, they still have to access global/write state remotely. For example, S6 relies on the microthread to reduce the cost of remote access. CHC can only cache global/write state when there is no more than one instance. Although our solution trades state consistency for higher performance, we think it is reasonable considering the following three aspects: (1) It can satisfy the specific NF scenario without violating the correctness of packet processing; (2) It provides rich ways to manage state and users can deal with consistency issues by themselves; (3) It provides more choices for state handling and platforms can have more flexibility to optimize performance. Therefore, Serpens works for NFs where strong global consistency is not necessary. If it is in the worst case, then Serpens can default to approaches similar to S6 and CHC, i.e., remote access would be required. However, we believe that our attempt is meaningful since Serpens is the first to localize all states and can provide some inspiration for subsequent work.

C. Advanced Service Chaining

As mentioned in Section II-B4, current service chaining approach in FaaS platforms depends on a proxy to deliver packets among separate NF instances. However, packet delivery incurs much higher overhead than actual packet processing and severely degrades the performance of FaaS platform. First, all packets need to wait in queues between every two NFs, which could increase latency due to potentially long waiting time. Second, if multiple NFs in an SFC run on the same CPU core, they suffer from context switch overhead. Otherwise, the overhead of inter-core communication and cache threshing is unavoidable. The essential cause of performance overhead is the *coupling between resource unit and NF execution*, which starts a new instance for each NF. Based on this observation, we propose a *resource-NF decoupling* mechanism and perform NF execution of the entire SFC within one instance. Although such a mechanism can lead to weaker isolation, our reasoning is that: (1) all NFs of an SFC belong to the same user and do not require such strong isolation as between SFCs [21]. (2) users are not aware of this variation and still benefit from modular design by providing their NFs as functions. (3) the length of SFC is usually



(a) SFC support in Serpens (b) Dynamic SFC support by transforming it to a prime sequential SFC with label array

Fig. 6. Service chaining approach in Serpens.

short (e.g., 4 [69], [70]), and this mechanism has little influence on resource management and flexibility.

Specifically, we consolidate multiple NFs of an SFC into a single instance and chain them through sequential function calls. All packets for an SFC are delivered to one instance and no packet delivery between instances is required. Fig. 6(a) shows the service chaining approach in Serpens. We introduce a *Chainer* into each instance and it sequentially calls NFs according to the SFC requirement. Since all NFs are uploaded with source codes, similar to [60], we develop an automatic tool with well-designed templates to compile them into a binary file. However, in some SFCs, the subsequent NF of a packet is determined by the processing result of its previous NF, which we call dynamic SFC. The left part of Fig. 6(b) shows an example of the dynamic SFC. All packets for this SFC are first sent to the Firewall (FW). If a packet is detected as suspicious, it needs to be sent to the IDS for further inspection. Otherwise, it is sent to the Monitor (MON) for statistics.

To support such SFC, a naive approach is attaching buffers to each branch and scheduling packets to these buffers [3]. However, it introduces complex scheduling logic, which may burden the Chainer and incur performance degradation. To design a lightweight Chainer, our key idea for dynamic SFC is *cutting off all execution branches and assembling NFs as a prime sequential SFC*. To achieve this goal, we introduce a *label array* for each batch of packets. Each label records the next NF for a packet, determined by the processing result of its previous NF. When the Chainer calls an NF, it first checks the label array to determine which packets in the batch should be processed by this NF. After execution, the NF needs to update these labels according to processing results. The right part of Fig. 6(b) shows an example of the label array. After the packets are processed by FW (①), the label array is updated to "2–3–3–2," which indicates that the first and fourth packets should be processed by IDS (②) while others should be processed by MON (③). In this way, we can simply call all NFs sequentially to achieve the same effect of dynamic SFC. Although the approach of consolidating multiple NFs of an SFC into a single instance has been proposed in NFV (e.g., NetBricks [3]), Serpens is different mainly in two aspects. First, we do not require NFs to be written in Rust [71]. Second, we introduce a label array to accelerate dynamic SFC.

D. Elastic NF Scaling

Since FaaS platform relies on dynamical resource provision to handle varying packet rates, the mechanism of NF scaling is at the core of FaaS platforms. However, overdue NF scaling can lead to performance fluctuation [53], especially during the period of resource variation. Next, we introduce the scaling mechanism of Serpens for instances and states. Since we consolidate multiple NFs of an SFC and treat entire SFC as an NF, the scaling mechanism between SFC and NF has no difference. Therefore, we scale entire SFC across CPU cores instead of single NF to avoid overload. For the NF failure, similar to [52], we can easily start a new instance to reload previous states and handle remaining packets. Considering more complex situations, we can refer to the design described in [72] and leave it as our future work.

TABLE IV
AVAILABLE INSTANCE SCALING MECHANISMS

Scaling Mechanism	Scaling Method	Related Work	Overhead (μs)
Scaling Out	Instance Snapshot	Catalyzer [13]	56980
Scaling Up	Process Fork	SAND [21]	250
Scaling Up	Thread Create	Photons [73]	24
Scaling Out	Memory Map	**Serpens**	34

1) Instance Scaling: Existing instance scaling mechanisms can be classified into two categories: (1) Scaling out (horizontal scaling). This mechanism starts new instances to occupy more resources and the resources of each instance are equal. (2) Scaling up (vertical scaling). This mechanism employs more resources into existing instances and the resources of instances are elastic. Recent research works [21], [73] adopt scaling up rather than scaling out mainly due to the scaling overhead. As shown in Table IV, SAND [21] utilizes Process Fork to scale up instance resources. Compared with the Instance Snapshot mechanism discussed in Section II-B2, Process Fork can lazily initialize most resources and reduce the scaling overhead to 250 μs. Moreover, Photons [73] utilize Thread Create to improve processing capacity. Compared with Process Fork, Thread Create can share the same address space and slash the scaling overhead to 24 μs. However, as shown in the last row of Table IV, benefit from the startup mechanism designed in Serpens, which only requires a Memory Map, the scaling out overhead can be reduced to 34 μs, which is comparable to the overhead of fastest scaling up.

In Serpens, we adopt the scaling out mechanism for instances based on the consideration of three aspects: (1) the scaling out overhead is reduced to tens of microseconds, which significantly slashes the advantage of scaling up. (2) the resources of scaling

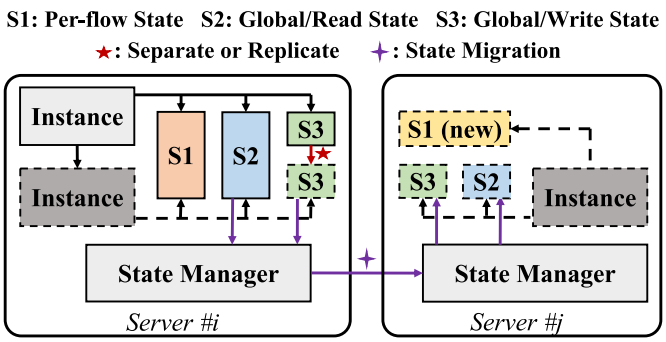


Fig. 7. NF scaling workflow inside one server and across multiple servers.

up are limited in a single server while scaling out can overcome this shortcoming and utilize the resources of multiple servers. (3) the resources of each instance are different in scaling up, which increases the complexity of resource management.

2) State Scaling: To handle suddenly increased packet rates, we dynamically start new instances for NF scaling. Besides, we need to migrate some flows as well as their states to these newly started instances. Due to the limited resources of single server, newly started instances could be placed on other servers. Thus, state migration between servers must be done to ensure correct packet processing. However, during the migration of states, corresponding flows need to be temporarily terminated for processing, which could incur significant performance fluctuation (e.g., OpenNF [24], S6 [53], CHC [65] and SNF [10]). To address this challenge, we propose the *performance fluctuation minimization strategy*. First, we prefer starting new instances within the same server of previous instances, which could benefit from the shared memory design to avoid state migration. Second, if scaling instances across servers is required, similar to E2 [74], we can avoid the migration of per-flow state by only steering new flows to new instances, while reusing existing instances to process old flows. When state migration across servers is unavoidable, we can utilize the mechanism proposed in SNF [10] to guarantee the correctness of state migration. Third, if the NF contains global state, we need first separate or replicate the global/write state, and then migrate the global state to corresponding server. However, no matter scaling NF inside the same server or across multiple servers, we only terminate packet processing while separating or replicating the global/write state, which usually can be done in a very short time (e.g., 10 μs).

In Serpens, we rely on the *State Manager* in each server to handle state scaling. When instance *scaling out* starts, the state manager separates or replicates global/write state and migrates them to other servers if required. Fig. 7 shows the intra-server and inter-server NF scaling workflow. When scaling NF with per-flow state (i.e., S1), we use the same state inside the same server or a new state in other servers for newly started instances. For NF with global/read state (i.e., S2), we use the same state in the same server or a replication of it in other servers. For NF with global/write state (i.e., S3), we first separate or replicate the state. If the newly started instance locates on other servers, we need further migrate the state to that server. When instance *scaling in* starts, the state manager merges states of recycled instances into other running instances of the same NF. If no running instance exists at that time, the state manager can store the state to persist storage for future access.

Compared with the state migration mechanism in NFV platforms (e.g., OpenNF [24], S6 [53] and CHC [65]), Serpens differs mainly in two aspects: 1) they mainly solve the workflow and correctness of state migration, while Serpens further analyzes the state migration scenario to optimize the migration strategy for state scaling. For example, Serpens utilizes the shared memory design in state manager to avoid state migration by starting new instance within the same server. 2) They utilize the arrival of packets to trigger the migration of corresponding states, which can lead to long scaling in time if no packets happen to arrive. Serpens utilizes the state manager to proactively migrate states, and can quickly finish the migration process when instance scaling in is required. Compared with the proactive state migration in recent research works (e.g., SNF [10]), Serpens differs mainly in three aspects: 1) they migrate the state between different flowlets of the same flow, while Serpens handles the requirement of state migration during NF scaling. 2) They focus on the design of when and where to migrate state, while Serpens targets at how to migrate the state to minimize performance fluctuation. 3) They only migrate the per-flow state, while Serpens carefully classifies existing states and provides a migration solution for all of them.

IV. Abstraction and Implementation

In this section, we first introduce the programming abstraction of Serpens to ease NF development and SFC chaining. Then, we present the implementation of Serpens, especially on the differences with existing platforms.

A. Programming Abstraction

Serpens proposes new state management and service chaining mechanisms to improve the performance of FaaS NFs, which requires developers adapting to Serpens. To ease the development effort, Serpens provides a programming abstraction in the format of APIs to hide the complexity of state management and service chaining. Fig. 8 presents these APIs. Next, we introduce them in detail.

State Management. We design four APIs for NF state storage and handling. The $create\_table$ function creates a new table in shared memory for state storage. Besides, the developer needs to indicate the category of state stored in the table by setting *flags*. We provide four flags according to state classification, including *per-flow, global, read and write*. Besides, if the flag is `global | write`, the developer needs to indicate guaranteed consistency, including *release* and *eventual*. The get_state and put_state functions provide state access and update, and are able to directly operate the shared memory. The $delete\_state$ function allows the developer to delete an existing per-flow state in a *table* against a flow *key* (e.g., five-tuple). If the table stores global state, the function throws an exception since global state is not allowed to be deleted when an NF is running.

```
//STATE STORAGE AND HANDLING
create_table(table, flags);
get_state(table, key);
put_state(table, key, value);
delete_state(table, key);
flags = per-flow | global | read | write |
        [release | eventual]

//EXTRA HANDLING FOR GLOBAL/WRITE STATE
split(table, key);
merge(table, key);
update(table, key);
timer(table, key, timeout);
trigger(table, key);

//SERVICE CHAINING FOR DYNAMIC SFC
chain(nf1, nf2, ...);
distribute(nf1, nf2, ..., selector);
aggregate(nf1, nf2, ...);
```

Fig. 8. Programming abstraction in Serpens.

```
1  void nat_init() {
2      create_table(pool_table, global | write | release);
3      create_table(mapping_table, per-flow | write);
4      init_pool(cluster_id, IPs);
5      put_state(pool_table, cluster_id, IPs);
6  }
7
8  int nat_handler(pkt_t pkt) {
9      extract 5-tuple from pkt;
10     (IP, Port) = get_state(mapping_table, 5-tuple);
11     if ((IP, Port) == NULL) {
12         IPs = get_state(pool_table, cluster_id);
13         (IP, Port) = select(IPs);
14         if ((IP, Port) == NULL) {
15             trigger(pool_table, cluster_id);
16             IPs = get_state(pool_table, cluster_id);
17             (IP, Port) = select(IPs);
18             if ((IP, Port) == NULL)
19                 return DROP;
20         }
21         update_pool(IPs, (IP, Port));
22         put_state(pool_table, cluster_id, IPs);
23         put_state(mapping_table, 5-tuple, (IP, Port));
24         extract reverse-5-tuple from pkt and new IP/Port;
25         put_state(reverse-5-tuple, (pkt.ip, pkt.port));
26     }
27     update_header(pkt, (IP, Port));
28     return PASS;
29 }
```

Fig. 9. Simplified NAT programming in Serpens.

For global/write states, extra APIs including *split*, *merge* and *update* functions, are called when NF scaling out/in or state update is required. To relieve NF development burden, we provide default operations. For release consistency, *split* function separates the state into two sub-states and *update* function equally splits available resources by default. For eventual consistency, *split* function copies the state into two replications and *update* function simply adds state value by default. Specifically, *update* function could be called either on time via a *timer* function (e.g., Monitor periodically sums up host statistics) or on demand via a *trigger* function (e.g., NAT adjusts available IPs/Ports when the IPs/Ports of an instance are exhausted).

Fig. 9 presents a simplified NAT programming to illustrate how to use the state management APIs to construct NFs. For stateful NFs, the developer needs to provide a function for

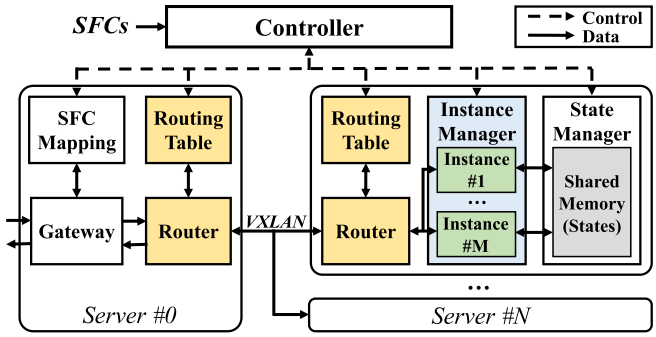


Fig. 10. Implementation of Serpens.

memory allocation and state initialization, and Serpens calls this function on deployment (line 1). For NAT, it creates two tables for IPs/Ports pool and Flow-IP/Port mappings and then initializes the IPs/Ports pool (lines 2-5). Note that the IPs/Ports pool is global/write state and the guaranteed consistency need to be set on creation (e.g., release consistency). Then, the developer needs to provide a function for packet processing (line 8). For each packet, it first looks up existing Flow-IP/Port mappings against 5-tuple (lines 9-10). If no mapping exists, it selects a new IP/Port from the IPs/Ports pool (lines 11-13). If IP/Port resources have been exhausted, the NAT actively triggers IPs/Ports reallocation among instances and tries to acquire a new IP/Port again (lines 15-17). If there is still no available IP/Port, it drops this packet (lines 18-19). Otherwise, it updates the IPs/Ports pool and Flow-IP/Port mappings with 5-tuple and reverse-5-tuple of the packet (lines 21-25). Finally, it updates the packet header with corresponding IP/Port and sends the packet out (lines 27-28).

Service Chaining. We provide three APIs to support dynamic service chaining in Serpens. The *chain* function is used to describe the desired NFs in an SFC and their execution order in a sequence. The *distribute* function is used to describe the branch. This function takes the downstream NFs as input, as well as a user-defined selector function to describe which NF the packet should be distributed according to the processing result of the upstream NF. The *aggregate* function merges packets from multiple NFs and takes the merged NFs as input.

B. System Implementation

We have implemented a prototype of Serpens and utilize DPDK [75] for networking I/O. Fig. 10 shows the detailed implementation upon a cluster of servers. The implementation is basically consistent with existing FaaS platforms. Here, we mainly introduce the *routing system* and *instance management mechanism* in Serpens, which may be different from existing platforms [10].

1) Routing System: We introduce a *Router* to each server for packet delivery, which dynamically forwards packets to corresponding instances according to its *Routing Table*. On receiving packets, the gateway *tags* their triggered SFC on each packet (DPDK provides a 64-bit metadata for tagging). Then, the router forwards packets against the SFC tag. If multiple instances are started for an SFC, other fields (e.g., 5-tuple) could

be matched to distribute packets among instances. Thus, when starting or destroying instances on demand, the controller needs to update corresponding Routing Tables to ensure consistent packet delivery. To maintain compatibility with current network, we utilize the VXLAN [76] to forward packets between servers and tag the triggered SFC on the reserved field in the VXLAN header.

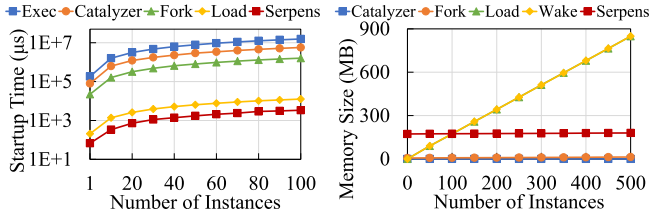
2) Instance Management Mechanism: We introduce an *Instance Manager* in each server to start or destroy instances on demand. Considering the processing capacity of a single instance is limited, we dynamically start more instances to handle increasing packet rates. To avoid frequent instance startup, we adopt a common practice to reuse started instances by keeping them "warm" for a period of time (e.g., 30 seconds [35]). To save resources and share CPU cycles, the instance actively sleeps during idle time and is woken up by the router if new packets arrive. However, sending a wake-up signal for every packet or massive accumulated packets could either lead to significant resource overhead or a long waiting time. In our current implementation, we choose a moderate number of packets (e.g., 128). However, it can be dynamically adjusted according to the requirement of NFs.

V. Evaluation

To evaluate the performance of Serpens, we implement five NFs including *FW* (Firewall, performing 5-tuple hash match on black/white lists), *NAT* (allocating IPs/Ports for new flows and modifying packet header for existing flows), *MON* (Monitor, maintaining exhaustive flow and host statistics), *IDS* (performing signature matching on public patterns) and *VPN* (encrypting packets based on AES-128).

Experimental Setup. Currently, we deploy Serpens on a testbed of four servers. Two servers run the deployed NFs and each is equipped with two Intel Xeon E5-2650 v4 CPUs (2.2 GHz, 12 physical cores), 128 GB total memory (DDR4, 2400 MHz, 16 GB x8) and one dual-port 10 G NIC (Intel X520-DA2). The other two servers run the gateway and packet generator, each is equipped with two Intel Xeon E5-2620 v3 CPUs (2.4 GHz, 12 physical cores), 128 GB total memory (DDR4, 2133 MHz, 16 GB x8) and one dual-port 10 G NIC (Intel X520-DA2). The packet generator is based on DPDK and directly connected to the gateway. It sends and receives packets to measure the end-to-end throughput and latency. The servers running gateway and deployed NFs are connected through an ethernet switch (10 Gbps per port, 24 ports). All servers run Ubuntu 14.04 (kernel 4.4.0) and use DPDK 18.11 for networking I/O.

To illustrate the benefits of Serpens, we compare it with Fission [20], SNF [10], SAND [21], SOCK [51], Catalyzer [13], OpenNF [24], S6 [53] and CHC [65]. Some platforms are developed with Go (e.g., Fission) or Java (e.g., SAND), which is inefficient for packet processing. Thus, we rewrite their networking I/O with DPDK and utilize C language to develop NFs. Besides, we re-implement the designs described in SNF, SOCK, Catalyzer, OpenNF, S6 and CHC on FaaS platform. Finally, we utilize the same network topology and server cluster of Serpens to evaluate their performance.



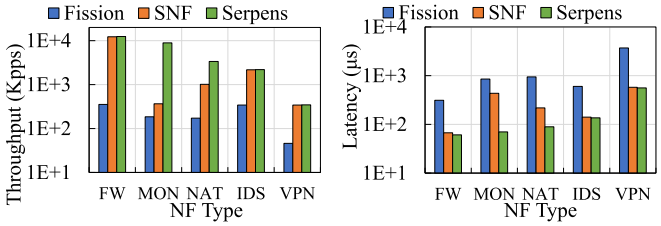
(a) CPU consumption of different startup methods (b) Memory occupation of different startup methods

Fig. 11. Resource overhead of different startup methods.

Evaluation Goals. We evaluate Serpens with the following goals: (1) the performance comparison of different instance startup mechanisms (Section V-A); (2) the performance improvement of Serpens state localization (Section V-B); (3) the benefit from Serpens service chaining approach (Section V-C); (4) the scaling effect of performance fluctuation minimization strategy (Section V-D); (5) the end-to-end performance improvement for SFCs over existing FaaS platforms (Section V-E).

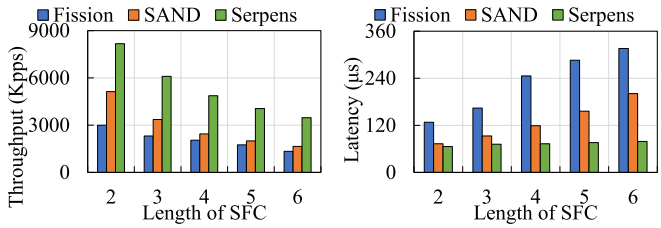
A. Instance Startup

CPU and Memory Overhead With Different Startup Methods. Existing instance startup methods are trade-offs between CPU and memory resources, Serpens achieves fast startup with marginal overhead through reusable NF runtime. To demonstrate its effect, we compare the startup time and occupied memory when adopting different startup methods with increasing numbers of instances. We choose Firewall as the target NF and evaluate its overhead with repeated deployment and startup. Fig. 11(a) shows the CPU consumption of different startup methods with increasing numbers of instances. We can observe that Serpens achieves the fastest startup in all methods. For example, Serpens spends 34 μs on instance startup, which has great advantage than `Exec` (160 ms, cold start [55]), `Fork` (16 ms, SAND [21]) and `Load` (138 μs, SOCK [51]). Note that the result of `Exec` and `Fork` is worse than reported startup time in [21], [29]. This is because the adoption of DPDK, which could increase the startup time due to more initialization for system environment. However, it has no impacts on `Load` and Serpens since they pre-initialize NF runtime to eliminate this overhead. Compared with the `Load`, Serpens can further reduce the startup time by about 75%. Besides, compared with the state-of-the-art cold start method (Catalyzer [13]), Serpens can reduce the startup latency from 56.98 ms to 34 μs. That is because even optimized with the snapshot technique, Catalyzer still needs to start the sandbox and process on demand, which could consume tens of milliseconds, while Serpens can completely skip their overhead with the process pool. Fig. 11(b) shows the memory occupation of different startup methods with increasing number of instances. We can observe that Serpens significantly reduces the memory occupation of NF deployment. For example, deploying an NF in Serpens just loads a binary file into shared memory and only occupies about 20 KB, so Serpens can easily support thousands



(a) Throughput with different FaaS platforms (b) Average latency with different FaaS platforms

Fig. 12. Performance of NFs.



(a) Throughput with different service chaining approaches (b) Average Latency with different service chaining approaches

Fig. 13. Performance of SFCs.

of NFs within one server. Meanwhile, Serpens needs to maintain a process pool in advance (e.g., 100), it could cost non-negligible memory at system initialization. However, with the number of NFs increasing, this extra memory overhead can be amortized by massive NFs. For example, to support 500 NFs, Serpens and `Load` occupies ∼210 MB and ∼1000 MB memory, respectively. Note that above memory is consumed at the deployment of NFs and belongs to extra memory overhead, users do not pay anything for it. Besides, memory is the bottleneck resource in data center [77], let alone thousands of NFs would run in each server [56] and their memory overhead on running (include NF state) is much larger than that on deployment. Therefore, the memory is very scarce and our memory saving efforts are worthwhile.

B. State Management

Performance Improvement of Serpens *State Localization.* To demonstrate the performance improvement of state localization in Serpens, we compare it with different state management mechanisms in existing FaaS platforms. The Fission platform [20] stores all states in remote storage. Here, we utilize the Redis [54], a popular key-value store, to store NF state and locate it at the same server of NF instances, to avoid extra latency of physical network. Besides, we aggregate multiple state access requests into a single one to improve throughput. The SNF platform [10] localizes the per-flow states but still needs to store the global states in remote storage. Fig. 12(a) shows the throughput of the three platforms with different NFs. We can observe significant throughput improvement from the remote to local state access, an average of 35.1×, 48.1×, 19.5×, 6.4× and 7.5× for FW, MON, NAT, IDS and VPN, respectively. We can see that MON improves the most than other NFs, this is because MON has frequent state access and occupies more time on remote state access. Compared with the SNF platform, Serpens can further improve the throughput for NFs with global states. For example, it can improve the throughput of MON by about 24.3×. Fig. 12(b) shows the average latency of the three platforms with different NFs under 80% of the maximum throughput. We also observed significant latency reduction, an average of 80.4%, 91.8%, 90.6%, 77.4% and 84.9% for FW, MON, NAT, IDS and VPN, respectively. Compared with the SNF platform, it can still reduce the latency of MON by about 83.9%.

C. Service Chaining

Benefit From Serpens *Service Chaining Approach.* To demonstrate the benefit of Serpens, we compare it with available service chaining approaches in existing FaaS platforms. The Fission platform [20] introduces a global proxy between every two NFs. The SAND platform [21] designs a local proxy in each server but still cannot avoid the queuing delay. We place all NFs of an SFC at the same CPU core to ensure equal resource consumption. Besides, we chain multiple Firewalls to form an SFC and vary its length to evaluate the performance. Fig. 13(a) shows the throughput of the three platforms with increasing length. We can see that Serpens achieves the highest throughput in all cases. It can outperform the Fission by 2.3× ∼ 2.7×, and outperform the SAND by 1.6× ∼ 2.1×. Fig. 13(b) shows the average latency of the three platforms with varying lengths under 80% of the maximum throughput. We also observe that the average latency can be reduced from 48.4% to 75.0% for Fission, and from 9.6% to 60.7% for SAND.

D. NF Scaling

Scaling Effect of Performance Fluctuation Minimization Strategy. To demonstrate the advantage of performance fluctuation minimization strategy in Serpens, we compare it with state migration-based approach (e.g., OpenNF [24], S6 [53] and SNF [10]). Here, we do not compare Serpens with approach that depends on remote state access to avoid state migration (e.g., StatelessNF [52]), due to the high performance overhead as evaluated above. However, we compare Serpens with CHC [65] since it relies on local cache to accelerate the access of per-flow state, which needs to be migrated during NF scaling. We evaluate the performance with a synthetic NF, which is similar to FW but can configure the size of state. We first start an instance for this NF and send 1Mpps traffic with 1 k concurrent flows to it. Then, we start a new instance at the same server and migrate half flows to the new one. Since all approaches could migrate half flows between instances within hundreds of milliseconds and have no obvious throughput variation, we only show the latency comparison of different state migration approaches during NF scaling in Fig. 14. As we can see, the latency of OpenNF, S6, CHC and SNF can be 138 ms ∼ 151 ms, 1.63 ms ∼ 1.82 ms, 1.16 ms ∼ 1.39 ms, and 0.46 ms ∼ 0.55 ms, which is much higher than the normal latency. However, the latency of Serpens can be reduced to 60 μs ∼ 63 μs, and almost incurs no latency

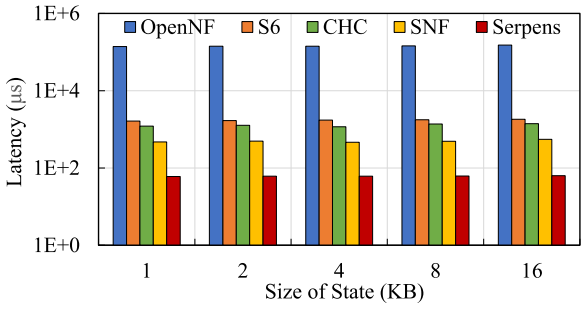
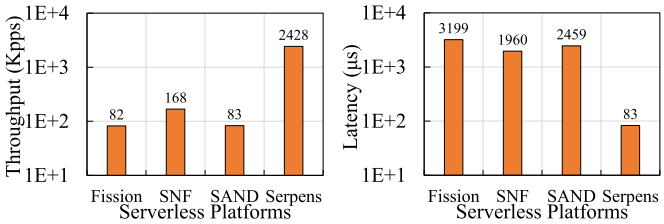


Fig. 14. Latency comparison of different state migration approaches during NF scaling.



(a) Throughput with different FaaS platforms (b) Average Latency with different FaaS platforms

Fig. 15. End-to-end performance of SFC.

variation during NF scaling. Besides, compared with OpenNF, S6, CHC and SNF, Serpens can localize all states while they still have to remotely access the global/write states and therefore lead to more performance overhead.

E. Performance Improvement

Performance Improvement for SFCs. To demonstrate the performance improvement of Serpens over existing FaaS platforms, we compare it with three popular platforms, including Fission [20], SNF [10] and SAND [21]. Besides, we chain the Firewall, Monitor and NAT to form an SFC. Fig. 15(a) shows the throughput of the SFC on different FaaS platforms. We can observe significant throughput improvement for Serpens, an average of 29.6×, 14.5× and 29.3× over the Fission, SNF and SAND, respectively. Fig. 15(b) shows the average latency of the SFC on different FaaS platforms. We also observe significant latency reduction for Serpens, an average of 97.4%, 95.8% and 96.6% over the Fission, SNF and SAND, respectively. Furthermore, we can observe that the throughput of the SFC can reach 2.4Mpps and the latency is only 83 μs, which is comparable with conventional NFV platforms and can satisfy the requirement of most scenarios [2], [11], [12].

VI. Discussion

Security. Since users usually provide source codes to FaaS platform, we cannot protect users from malicious providers. User NFs may be malicious and the design of Serpens could introduce security risks to FaaS platform. However, we considered the security in our design and argue that the risk is acceptable. First, Serpens runs NFs from different users into separate instances, and we can directly destroy the instance if the NF is found to be malicious. Second, Serpens utilizes read-only memory mapping to start NFs, any modification and data are stored in temporary memory under copy-on-write (COW) mechanism. Third, Serpens only maps the memory of required NF for startup, other NFs are isolated by the page table of operating system (OS). Fourth, Serpens can audit NF codes in advance and run them with software isolation [78], [79]. We leave the comprehensive analysis and design for security as our future work.

Elasticity. To improve the performance of NFs, we introduce a local state manager into each server and consolidate multiple NFs into one instance, which could hurt the elasticity and flexibility of FaaS platforms. However, we argue that our design choice is reasonable and acceptable. Different from general applications (e.g., microservice [80]), which could have diversified logics and consist of massive functions (e.g., 100), the state of NFs is relatively simple and the length of SFCs is usually short (e.g., 4). Therefore, the influence on elasticity under NF scenario is limited. Besides, to support various services (e.g., management and measurement), FaaS platform usually need to deploy lots of system components on each server, and we can also treat the components introduced by Serpens (e.g., state manager) as part of the system components.

Billing. One of the major advantages is users can pay for what they use at very fine time and resource granularity. Specifically, only the resources that are occupied to process packets are billed. If no packet arrives, the users do not need to pay anything even their NFs are loaded into memory or other services are still running. Besides, the resource granularity for billing is very fine, such as occupying hardware resources (e.g., memory) for 100 ms or processing specified packets for 10 [6], [7], [8]. Therefore, the platform should avoid any extra resource consumption (e.g., memory) if there are no packets to process. In contrast, once users have rented hardware resources on VM or container platforms, even if these resources are idle, users still need to pay for them and therefore leads to more monetary cost. For states stored in the state manager, similar to the hardware resources in FaaS platform, they can be billed by time and volume. Users are agnostic of the state manager and pay for these states as they are stored in the Amazon S3 [81].

VII. Related Work

FaaS Platform. Existing research works on FaaS platform can be roughly classified into two categories. The first category focuses on leveraging FaaS platform to support various applications [8], [9], [37], [38]. For example, Jonas et al. [37] propose supporting distributed computing on FaaS platform. Fouladi et al. [38] propose a mechanism to run video processing tasks on AWS Lambda. Singhvi et al. [8] propose a framework to execute NFs on AWS Lambda, but its preliminary evaluation concluded that current AWS Lambda is not suitable for NFs. The second category targets at improving FaaS platforms for various purposes [13], [21], [29], [45], [51], [82]. Pocket [82] is designed with an elastic ephemeral storage for analytics. SOCK [51]

includes an optimized container technology to reduce initialization time. Boucher et al. [29] adopt language-based isolation to reduce launching time. However, orthogonal to above research works, Serpens addresses the performance problem and designs a special-purpose FaaS platform for NFs.

NFV Platforms. Many NFV platforms have been proposed to provide high performance packet processing [3], [8], [12], [60], [83], [84]. For example, NetVM [83] consolidates multiple NFs into one server and accelerates packet delivery among NFs through shared memory. NetBricks [3] adopts language-level virtualization technology and runs multiple NFs in a single process to improve performance. NFP [12] analyzes the state operation of various NFs and utilizes network function parallelism to reduce latency of an SFC. Recently, many research works propose to run NFs on cloud platforms to relieve the management burden of infrastructure and leverage the economies of scale [5], [8]. However, different from existing research works, Serpens introduces FaaS platform to support NFs and design a high performance platform to run NFs and SFCs.

VIII. Conclusion

We have presented Serpens, a high performance FaaS platform for NFs. Starting from the necessity of supporting NFs with FaaS platform as well as identifying three performance problems in existing FaaS platforms, Serpens proposes three designs to improve performance, including fast instance startup, novel state management and advanced service chaining. Moreover, Serpens provides an NF scaling mechanism to minimize performance fluctuation, and a programming abstraction to ease NF and SFC development. Our evaluations have demonstrated that compared with the NFs and SFCs that run on existing FaaS platforms, Serpens can improve the throughput by more than 10× and reduce the latency by more than 90%.

References

[1] V. Sekar et al., "Design and implementation of a consolidated middlebox architecture," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 323–336.

[2] J. Hwang et al., "NetVM: High performance and flexible networking using virtualization on commodity platforms," *IEEE Trans. Netw. Service Manag.*, vol. 12, no. 1, pp. 34–47, Mar. 2015.

[3] A. Panda et al., "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 203–216.

[4] AT&T, "AT&T to offload 5G into Microsoft's cloud," 2021. [Online]. Available: https://www.lightreading.com/the-core/atandt-to-offload-5g-into-microsofts-cloud/d/d-id/770600

[5] J. Sherry et al., "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, 2012.

[6] S. Hendrickson et al., "Serverless computation with openLambda," in *Proc. 8th USENIX Conf. Hot Top. Cloud Comput.*, 2016, pp. 33–39.

[7] C. Paul et al., "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[8] A. Singhvi et al., "Granular computing and network intensive applications: Friends or foes?," in *Proc. 16th ACM Workshop Hot Topics Netw.*, 2017, pp. 157–163.

[9] P. Aditya et al., "Will serverless computing revolutionize NFV?," *Proc. IEEE*, vol. 107, no. 4, pp. 667–678, Apr. 2019.

[10] A. Singhvi, J. Khalid, A. Akella, and S. Banerjee, "SNF: Serverless network functions," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 296–310.

[11] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, "Microboxes: High performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 504–517.

[12] C. Sun et al., "NFP: Enabling network function parallelism in NFV," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 43–56.

[13] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 467–481.

[14] J. Carlson, *Redis in Action*. New York, NY, USA: Simon and Schuster, 2013.

[15] J. Kreps et al., "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.

[16] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-based scalable network services," in *Proc. 16th ACM Symp. Operating Syst. Princ.*, 1997, pp. 78–91.

[17] openstack, "Open source cloud computing infrastructure - openstack," 2022. https://www.openstack.org/

[18] kubernetes, "Production-grade container orchestration," 2022. https://kubernetes.io/

[19] knative, "Kubernetes-based platform to deploy and manage modern serverless workloads," 2020. https://knative.dev/

[20] Fission, "Open source, kubernetes-native serverless framework," 2019. [Online]. Available: https://fission.io/

[21] I. E. Akkus et al., "SAND: Towards high-performance serverless computing," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 923–935.

[22] A. Greenberg et al., "Vl2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 51–62.

[23] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 63–74.

[24] A. Gember-Jacobson et al., "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 163–174, 2014.

[25] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synth. Lectures Comput. Architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[26] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.

[27] D. Xie et al., "The only constant is change: Incorporating time-varying network reservations in data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 199–210, 2012.

[28] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 44–48, 2012.

[29] S. Boucher et al., "Putting the "micro," back in microservice," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 645–650.

[30] M. Dobrescu et al., "Toward predictable performance in software packet-processing platforms," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 141–154.

[31] Amazon, "AWS Lambda serverless compute," 2019. [Online]. Available: https://aws.amazon.com/lambda/

[32] Google, "Serverless environment to build and connect cloud services," 2019. [Online]. Available: https://cloud.google.com/functions/

[33] Microsoft, "Azure functions serverless architecture," 2019. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[34] IBM, "IBM cloud functions," 2019. [Online]. Available: https://www.ibm.com/cloud/functions

[35] OpenFaas, "Serverless functions made simple," 2019. [Online]. Available: https://github.com/openfaas/faas

[36] OpenWhisk, "Apache openwhisk is a serverless, open source cloud platform," 2019. [Online]. Available: https://openwhisk.apache.org/

[37] E. Jonas et al., "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, 2017, pp. 445–451.

[38] S. Fouladi et al., "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 363–376.

[39] S. Fouladi et al., "From Laptop to Lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 475–488.

[40] Y. Li, "Queuing theory with heavy tails and network traffic modeling," 2018, ffhal-01891760.

[41] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 123–137, 2015.

[42] H. Yu et al., "Octans: Optimal placement of service function chains in many-core systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2202–2215, Sep. 2021.
[43] D. Breitgand et al., "Towards serverless NFV for 5G media applications," in *Proc. 11th ACM Int. Syst. Storage Conf.*, 2018, pp. 118–118.
[44] A. Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in *Proc. 17th Usenix Conf. Netw. Syst. Des. Implementation*, 2020, pp. 419–434.
[45] K.-T. A. Wang et al., "Replayable execution optimized for page sharing for a managed runtime environment," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.
[46] D. Ustiugov et al., "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 559–572.
[47] 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.
[48] H. Fingler, A. Akshintala, and C. J. Rossbach, "USETL: Unikernels for serverless extract transform and load why should you settle for less?," in *Proc. 10th ACM SIGOPS Asia-Pacific Workshop Syst.*, 2019, pp. 23–30.
[49] S. Shillaker and P. Pietzuch, "FAASM: Lightweight isolation for efficient stateful serverless computing," 2020, *arXiv 2002.09344*.
[50] 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. Operating Syst. Des. Implementation*, 2016, pp. 671–688.
[51] E. Oakes et al., "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 57–70.
[52] M. Kablan et al., "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX Conf. Netw. Syst. Des. Implementation*, 2017, pp. 97–112.
[53] S. Woo et al., "Elastic scaling of stateful network functions," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 299–312.
[54] S. Sanfilippo, "Redis: High performance in-memory data structure store," 2018. [Online]. Available: https://redis.io/
[55] L. Wang et al., "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 133–146.
[56] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. {USENIX} Annu. Tech. Conf.*, 2020, pp. 205–218.
[57] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 152–166.
[58] datawookie, "Shared memory & docker," 2021. [Online]. Available: https://datawookie.dev/blog/2021/11/shared-memory-docker/
[59] S. G. Kulkarni et al., "NFVnice: Dynamic backpressure and scheduling for NFV service chains," *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 639–652, Apr. 2020.
[60] S. Han et al., "Softnic: A software NIC to augment hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, 2015.
[61] W. Frings et al., "Massively parallel loading," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput.*, 2013, pp. 389–398.
[62] M. T. Jones, "GNU/Linux application programming," *Charles River Media*, 2008.
[63] M. K. Johnson and E. W. Troan, *Linux Application Development*. Reading, MA, USA: Addison-Wesley, 2004.
[64] H. Shacham et al., "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Comput. Commun. Secur.*, 2004, pp. 298–307.
[65] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 501–516.
[66] H. Sadok et al., "A case for spraying packets in software middleboxes," in *Proc. 17th ACM Workshop Hot Topics Netw.*, 2018, pp. 127–133.
[67] P. Keleher et al., "Lazy release consistency for software distributed shared memory," *ACM SIGARCH Comput. Architecture News*, vol. 20, no. 2, pp. 13–21, 1992.
[68] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, 2013.
[69] M. T. Surendra, S. Majee, C. Captari, and S. Homma, "Service function chaining use cases in data centers," *Working Draft, IETF Secretariat, Internet-Draft draft-ietf-sfc-dc-use-cases-06*, 2017.
[70] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 51–62.
[71] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Lett.*, vol. 34, no. 3, pp. 103–104, 2014.
[72] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu, "Reinforce: Achieving efficient failure resiliency for network function virtualization based services," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, 2018, pp. 41–53.
[73] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a diet," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 45–59.
[74] S. Palkar et al., "E2: A framework for NFV applications," in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 121–136.
[75] Intel, "Data plane development kit (DPDK)," 2019. [Online]. Available: http://dpdk.org
[76] M. Mahalingam et al., "Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks," Tech. Rep. rfc7348, 2014.
[77] J. Guo et al., "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. IEEE/ACM 27th Int. Symp. Qual. Serv.*, 2019, pp. 1–10.
[78] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symp. Operating Syst. Princ.*, 1993, pp. 203–216.
[79] B. Yee et al., "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. 30th IEEE Symp. Secur. Privacy*, 2009, pp. 79–93.
[80] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 3–18.
[81] A. W. Service, "Cloud object storage-amazon S3," 2022. [Online]. Available: https://aws.amazon.com/s3/?nc1=h_ls
[82] A. Klimovic et al., "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 427–444.
[83] W. Zhang et al., "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Funct. Virtualization*, 2016, pp. 26–31.
[84] H. Yu et al., "Buffet: Enabling multi-tenant network functions," in *Proc. Glob. Commun. Conf.*, 2019, pp. 1–6.

Heng Yu received the BS degree from the School of Communication and Information Engineering, University of Electronic Science and Technology, China, in 2017, and the PhD degree from the Institute for Network Sciences and Cyberspace, Tsinghua University, China, in 2023. He is currently working in the Zhongguancun Laboratory. His research interests include cloud computing, serverless computing, and network function virtualization.

Han Zhang (Member, IEEE) received the BS degree in computer science and technology from JiLin University and the Ph.D. degree in Tsinghua University. He is currently working in the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include computer network systems and network security.

Junxian Shen received the BS degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2019. He is currently working toward the PhD degree in the Institute for Network Science and Cyberspace, Tsinghua University. His research interests include network function virtualization, cloud computing, and serverless computing.

Yantao Geng received the BS degree from the Department of Automation, Tsinghua University, China, in 2022. He is currently working toward the PhD degree in the Institute for Network Science and Cyberspace, Tsinghua University. His research interests include congestion control, serverless computing, and software defined network.

Jilong Wang (Member, IEEE) received the PhD degree in computer science from Tsinghua University, Beijing, China, in 1996. He is currently a professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include network architecture, network measurement, cyberspace mapping, and cyberspace governance.

Congcong Miao received the BS degree in the Beijing University of Posts and Telecommunications, China, in 2015, and the PhD degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2020. He is currently a senior engineer in Tencent. His research interests include network measurement and management, machine learning in networking, and cloud computing.

Mingwei Xu (Senior Member, IEEE) received the BS and PhD degrees from Tsinghua University, China. He is currently a professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include computer network architecture, high-speed router architecture, and network security.