Serpens: A High-Performance Serverless Platform for NFV

Junxian Shen[1,3], Heng Yu[1,2,3], Zhilong Zheng[2,3], Chen Sun[1,4], Mingwei Xu[1,2,3], Jilong Wang[1,2,3]
[1]Institute for Network Science and Cyberspace, BNRist, Tsinghua University
[2]Department of Computer Science and Technology, Tsinghua University, [3]Peng Cheng Laboratory (PCL), [4]Alibaba Group
{shenjx19, h-yu17, zhengzl15}@mails.tsinghua.edu.cn, qichen.sc@alibaba-inc.com, {xumw, wjl}@tsinghua.edu.cn

***Abstract*—Many enterprises run Network Function Virtualization (NFV) services on public clouds to relieve management burdens and reduce costs. However, NFV operators still face the burden of choosing the right types of virtual machines (VMs) for various network functions (NFs), as well as the cost of renting VMs at a granularity of months or years while many VMs remain idle during valley hours. A recent computing model named *serverless computing* automatically executes user-defined functions on requests arrival, and charges users based on the number of processed requests. For NFV operators, serverless computing has the potential of completely relieving NF management burden and significantly reducing costs. Nevertheless, naively exploring existing serverless platforms for NFV introduces significant performance overheads in three aspects, including *high remote state access latency*, *long NF launching time*, and *high packet delivery latency between NFs*. To address these problems, we propose Serpens, a high-performance serverless platform for NFV. Firstly, Serpens designs a novel state management mechanism to support local state access. Secondly, Serpens proposes an efficient NF execution model to provide fast NF launching and avoid extra packet delivery. We have implemented a prototype of Serpens. Evaluation results demonstrate that Serpens could significantly improve performance for NFs and service function chains (SFCs) comparing to existing serverless platforms.**

I. Introduction

Network Function Virtualization (NFV) was recently introduced to address the limitations of traditional proprietary middleboxes. NFV runs Network Functions (NFs) on commodity servers rather than various dedicated hardware to simplify management and reduce capital and operation expenses [1]. Currently, many enterprises deploy NFs on public clouds to completely relieve the management burden of machine maintenance and leverage the economies of scale to reduce costs [2].

However, running NFV on public clouds still faces significant management burdens and unnecessary costs. First, public clouds offer dozens of VM types with subtly different hardware configurations. Meanwhile, the performance of different NFs is sensitive to different types of hardware resource [3]. Thus, it is challenging to find the most suitable VM types for various NFs. Second, the deployment and performance tuning of service function chains (SFCs) are difficult. To achieve high performance, merely choosing the right CPU cores to run NFs in an SFC is non-trivial [4], let alone other hardware details and traffic characteristics. Finally, NFV operators have to rent sufficient VMs to handle peak workload [5]. Besides, launched VMs are billed at a granularity of hours to months, while many VMs are idle for most of its processing time [6].

Recently, a new computing model named *serverless computing*, also known as *function-as-a service (FaaS)*, is introduced, which has the potential of eliminating problems mentioned above like management burden and further reducing costs. Users only need to provide *functions* to the serverless platform. The platform automatically executes the functions within newly deployed instances (e.g. containers) to process requests and destructs these instances when no more requests arrive for processing. Users no longer need to consider function deployment, management, and scaling issues. Moreover, serverless platforms charge users according to the number of requests, i.e. packets in the NFV context, so that users only pay for what they use at a very fine granularity. Above benefits have been highlighted in many researches [7]–[9], and serverless computing is already available in commercial platforms [10], [11] and open-source platforms [12], [13].

However, naively exploring existing serverless platforms to support NFV could suffer from significant performance overhead. As a recent research effort reported [7], the average latency of a serverless-enabled Firewall is 3.39 ms, which is higher than a container-based [14] or process-based Firewall [15] by one to two orders of magnitude. Moreover, we have implemented a Firewall on an advanced serverless platform [16] and found that the total latency between a batch of packets entering and exiting the Firewall is 482.54 μs, while the actual packet processing time is merely 9.11 μs, which reveals severe performance overhead in serverless enabled NFV. To fully understand the performance problem, we profiled this NF and identified the following three major latency sources. We present more details in §II.

- **State access latency.** To maintain function state (e.g variants) after instance destruction, existing serverless platforms use remote storage to store state, which could incur high state access latency during packet processing.
- **NF launching time.** Existing platforms launch a new process that executes NFs for each batch of packets, which introduces unavoidable NF launching time.
- **Packet delivery latency.** Existing platforms forward packets between NFs in an SFC in a centralized or distributed manner, which suffers from high packet delivery latency.

To address the above problems, we propose Serpens, a

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

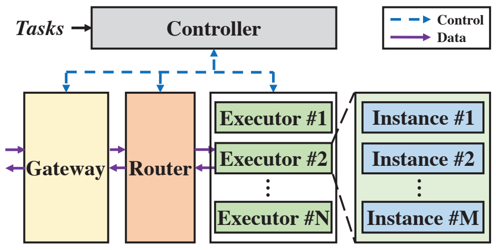


Fig. 1: Framework of typical serverless platforms

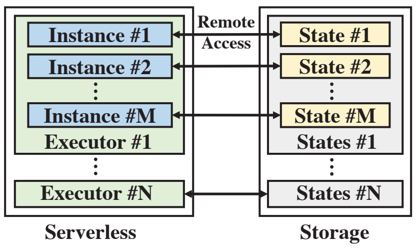


Fig. 2: State access in serverless

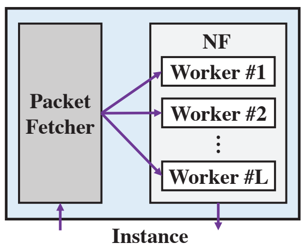


Fig. 3: NF execution model

high-performance serverless platform for NFV. To the best of our knowledge, Serpens is the first serverless platform specially designed for NFV. Serpens first designs a novel state management mechanism that persists state in shared memory and supports local state access for all NFs. Second, Serpens proposes an efficient NF execution model by decoupling resource unit and NF execution, and utilizes function call to enable fast NF launching and avoid extra packet delivery. Moreover, Serpens provide a programming abstraction to mitigate NF development efforts based on our serverless platform.

Overall, this paper makes the following contributions:

- We present the motivation of enabling serverless computing for NFV and elaborate on the performance problem of existing serverless platforms to support NFs (§II).
- We design Serpens, a high-performance serverless platform for NFV, including a novel state management mechanism to support local state access and an efficient NF execution model with resource-NF decoupling to enable fast NF launching and avoid extra packet delivery(§III).
- We have implemented a prototype of Serpens. Evaluation results demonstrate that compared with existing serverless platforms, Serpens could improve performance for NFs and SFCs by up to two or three orders of magnitude (§IV).

II. Motivation and Problem Statement

In this section, we first introduce the concept of serverless computing and the reason why serverless could benefit NFV. Next, we thoroughly analyze the performance problem of existing serverless platforms when supporting NFV.

A. Serverless Computing

Serverless computing is emerging as an essential evolution in cloud computing. This model shifts the responsibilities of application deployment to cloud providers and significantly reduces the burden on users. In serverless computing, users no longer have to launch VMs and deploy applications by themselves. Instead, they only need to provide their applications as a set of functions and register corresponding requests to trigger these functions, the serverless platform takes the responsibility of deploying applications and running associated functions with an ephemeral execution environment when receiving trigger requests. Besides, users do not need to manage cloud resource and scale applications according to workload variation. The serverless platform allocates resources for function execution on demand and users only pay for what they use at a very fine granularity. For example in AWS Lambda [10], the costs of users are computed either based on the time their functions running at the granularity of millisecond or based on the number of requests their functions processing.

B. When NFV Meets Serverless

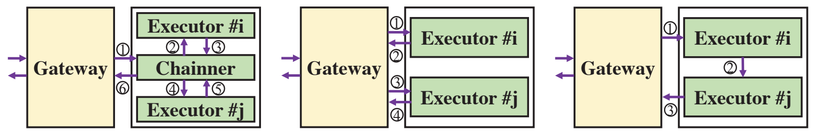
As mentioned above, running NFV on public clouds still suffers from management burdens and unnecessary costs. Serverless computing has the potential of overcoming these shortcomings. We present a typical framework of serverless platforms in Fig. 1. This framework is derived from open-source serverless platforms [12], [13], [16], which could also reflect key characteristics of commercial platforms. Next, we introduce how NFV could benefit from serverless platforms, mainly focusing on the NF deployment and NF scaling.

1) NF deployment: To deploy NFs on serverless platforms, users need to provide their NFs as a set of functions and register corresponding packets to trigger these NFs. As shown in Fig. 1, on receiving deployment tasks, the *Controller* first initializes an *Executor* for each deployed NF. These executors maintain the required execution environments for NFs. Then, it registers corresponding trigger packets for NFs to the *Gateway* and the gateway records these packet-NF pairs for future mapping. Finally, it updates the routing information in the *Router* to correctly deliver packets to executors. On receiving packets, the gateway first maps these packets to their triggered NFs and tags these NFs into packets. Then, the router delivers packets to executors according to the tags in packets. Finally, executors run deployed NFs to process these packets and return execution results.

2) NF scaling: To handle time-varying workloads, serverless platforms need to provide on-demand resource management and scaling. As shown in Fig. 1, each executor represents a specific NF and consists of multiple *Instances*. Usually, each instance in an executor represents a container and runs the same NF as other instances. Based on NF's current workloads, the controller dynamically launches and destroys instances to provide corresponding packet processing capacity. Note that to save resources, the controller could destroy all instances of an executor if no packet arrives for a period of time.

C. Performance Problem

Although many serverless platforms are available today, they are all designed for general-purpose computing and could incur high performance overhead when directly supporting NFV. To understand the performance problem, we



(a) Executor based chaining (b) Gateway based chaining (c) Direct chaining

Fig. 4: Service chaining approaches of existing serverless platforms

Packet Processing	State Access	NF Launching	Packet Delivery
9.11	223.71	249.72	70.97

Fig. 5: Latency (μs) profiling when running Firewall on a state-of-the-art serverless platform [16]

thoroughly analyze the key mechanisms of existing serverless platforms [12], [13], [16] and rewrite them to support packet processing. For convenience and without loss of generality, we take Firewall as a typical NF example when analyzing performance problem. Our basic conclusions remain the same for other NFs. Through exhaustive profiling (evaluation setup shown in §IV)) the latency of the entire packet processing pipeline, we identify three major performance overheads.

1) Remote state access: Many NFs are stateful and need to maintain persistent states for correct packet processing [17]–[19]. To support stateful applications, existing serverless platforms store the state through storage services [8], [20]. Fig. 2 shows the current state management mechanism of serverless platforms. This approach decouples the states that NFs need to maintain from NF processing, and store the state in a remote storage [21]. In this way, even all instances of an executor are destroyed, the NF's state still persists for subsequent packet processing. However, such remote state access could bring significant latency overhead, especially for those NFs requires state access for every packet. As shown in Fig 5, the actual time for a Firewall to process a batch of packets is only 9.11 μs. However, the state access time reaches 223.71 μs, which is 25$\times$ of the packet processing time.

2) Slow NF launching: Each instance is usually a container and runs the deployed NF. When the NF needs to access state, it sends a request to the storage service and waits for response, which would block processing and waste CPU cycles. To avoid this waiting, a common approach is launching multiple processes to hide remote access and keeps CPU busy with concurrency. Fig. 3 shows the current NF execution model in an instance and each *Worker* represents a process. During initialization, the instance launches a *Packet Fetcher* process to wait for packets. Upon receiving a batch of packets, the Packet Fetcher launches a worker to process these packets. In most open-source and commercial platforms, such as OpenFaaS [12] and AWS Lambda [10], the Packet Fetcher launches a worker by loading the binary file of NF. However, this approach suffers from long NF launching time [22]. To reduce the launching time, SAND [16], a state-of-the-art serverless platform, proposes to load the binary file in advance and fork a worker to execute it when needed. However, this approach still needs to launch a new process (with forking) for each batch of packets. As shown in Fig. 5, the launching time of a Firewall could be 249.72 μs and is 27$\times$ of the packet processing time.

3) Costly packet delivery: In NFV, multiple NFs are usually chained together to form an SFC. As shown in Fig. 4, existing service chaining approaches in open-source serverless platforms can be classified into three categories. Fig. 4(a) shows the executor based chaining approach and it introduces a special executor, which we call *Chainner*, for each SFC. The chainner is responsible for identifying the next NF and sending packets to NFs according to SFC requirements. However, since packets are delivered through the router (shown in Fig. 1, skipped here for simplicity), this approach needs the router to deliver packets 6 times for an SFC with 2 NFs. Fig. 4(b) shows the gateway based chaining approach and it identifies the next NF through gateway. However, this approach still needs to send packets to the gateway between every two NFs and are delivered 4 times. Fig. 4(c) shows the state-of-art chaining approach and it directly sends packets to the next NFs. With this approach, packets are only delivered 3 times and could achieve the lowest latency so far. However, all these approaches need the router to deliver packets between NFs and could incur high delivery latency. As shown in Fig. 5, even directly chaining two Firewalls to form an SFC, the packet delivery latency between the two Firewalls could be 70.97 μs and is 8$\times$ of the packet processing time.

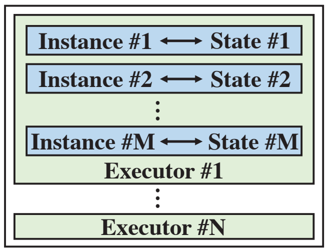
Summary and root cause analysis. All of the above mechanisms are designed for general-purpose computing and could efficiently support lots of applications such as video processing [9] and distributed computing [8]. However, as shown in Fig 5, they incur high performance overhead when supporting NFV. This is because NFs are typical *narrow tasks* that usually finish in a short time, and the actual processing time on a batch of packets only occupies a small portion of the total running time. Based on this observation, we propose a new serverless platform specially designed for NFV. This platform inherits the framework in Fig. 1 to maintain the benefits of serverless computing, but adopts completely different designs on the above three aspects to improve performance.

III. Serpens Design

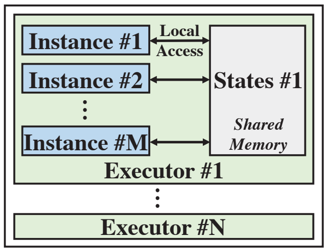
In this section, we first present the design of `Serpens` on state management, decoupling between resource unit and NF execution to address the performance problem. Then, we introduce the programming abstraction of `Serpens` for NF development.

A. State Management

As mentioned in § II-C, stateful NFs could suffer from significant performance overhead due to remote state access in serverless platforms. A straightforward solution for this problem is *state localization*, i.e. co-locating state and NFs within VMs or containers [17]. Several existing NFV platforms [17], [19] adopt this approach to design their state management mechanisms. As shown in Fig. 6(a), this approach stores states



(a) State management in NFV



(b) State management in `Serpens`

Fig. 6: State management for serverless platform

inside instances for local and efficient state access. Unfortunately, to save resources and improve efficiency, serverless instances are designed to be short-lived and stateless [7], which implies that NFs and states would be destroyed if no packet arrives for some time (e.g., 3 minutes [12]). Thus, this approach cannot guarantee correct packet processing for stateful NFs in serverless platforms.

To address this problem, we propose to *decouple the lifecycle of each instance and its state*. This design is based on our observation that the state loss is caused by the shared lifecycle of instances and their states. Instead of storing state within instances, we store the state of NFs *close to the instances*, i.e. inside the executors, which ensures both state localization and state persistence after instance destruction. Moreover, a major benefit of remote shared state access [18] is avoiding state migration among instances during NF scaling. Similarly, we also choose to store the state of the same NF in *shared memory* inside the executor, which significantly ease state management. Fig. 6(b) shows the state management mechanism in `Serpens`. Each executor has its own shared memory, which is separate from others for safety. Since the state of NFs can be *persisted* in shared memory, newly launched instances could simply read/write it to correctly process subsequent packets, even if all previous instances have been destroyed.

However, state localization could only accelerate per-flow state access. NFs with global state, such as the traffic volume counter that should be maintained by all Monitor instances [17], still suffer from high performance overhead due to cross-instance state synchronization. Furthermore, due to the limited resource in each server, when scaling NFs in another server, all states, either per-flow or global, need to be migrated between servers to ensure correct packet processing, which could introduce unstable performance [19]. In response, we propose two key design choices. First, we try to localize all state, even global state, by slightly relaxing synchronization strictness. Second, we propose a performance fluctuation avoidance strategy for stable NF scaling.

1) Localizing all state: To understand state in NFs, we thoroughly study common NFs and classify NFs' state into four categories according to their state scope (i.e., per-flow or global) and state access patterns (i.e., read or write): (i) per-flow/read, (ii) per-flow/write, (iii) global/read, and (iv) global/write. TABLE I shows the state operations of some commonly used NFs (derived from [19], [23]). For the state in category (i) and (ii), it is localized since packets in a flow

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

TABLE I: State scope and access pattern of common NFs

are always sent to the same instance and access the same shared memory. State in category (iii) never changes after NF initialization. Therefore, we can just maintain a copy of this state for each instance or for each executor, which then makes global/read state local. For category (iv), multiple instances need to update the global/write state simultaneously, which requires consistency guarantees. A naive approach is using synchronization primitives (e.g., locks). However, it could compromise the performance. To address this challenge, we propose to slightly trade the 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 to avoid different flows being mapped to the same IP/Port. For this type of state, ensuring *release consistency* [24] is sufficient. We separate such state into multiple sub-states and bind each sub-state with 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 operate a 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* [25]. Specifically, we copy the state into multiple *replications* and assign each replication to an instance. For two Monitor instances, to maintain statistics of 100 hosts, we enable each instance to maintain local host statistics, and 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 own local pool of IPs/-Ports and refuses new connections while other instances have

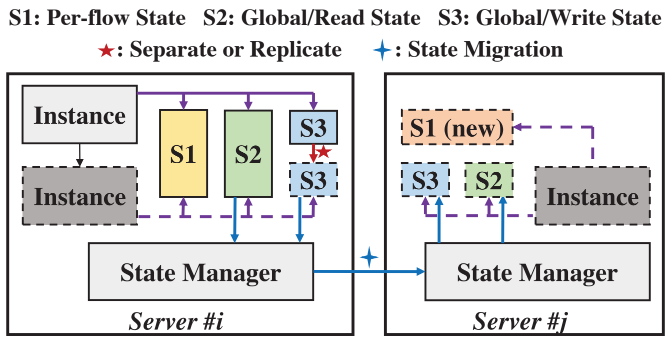


Fig. 7: NF scaling workflow inside one server and across multiple servers

enough IPs/Ports, which raises the requirement of dynamical state update across instances. We provide two ways to update the localized state of the global/write state. The first one is periodical updating. For example, we could periodically reallocate the remaining IPs/Ports across multiple NAT instances. The second one is actively updating. For example, a NAT instance could actively trigger the reallocation of remaining IPs/Ports across multiple NAT instances after exhausting local IPs/Ports.

2) NF scaling without performance fluctuation: When an NF instance is overloaded, the existing approach for NF scaling out is launching a new instance and migrating some flows with their states from the overloaded instance to the new one. Nevertheless, due to the limited server capacity, the newly launched NF instance could be placed on other servers. Thus, state migration between servers *must* be done to ensure packet processing correctness. Especially, the per-flow state has to be migrated while these flows are terminated for processing, which could incur significant performance fluctuation [17]. To address this challenge, we propose the *performance fluctuation avoidance strategy*. First, we prefer launching new NF instances within the same server as previous instances and benefit from the shared memory design to avoid state migration. Second, when having to scale NF across servers, we adopt a migration avoidance strategy, which only steers new flows to new instances, while using existing instances to process old flows. Finally, to correctly handle global/write state, we separate or copy it during scaling. If the newly launched instances locate at other servers, we need further migrate the global state to that server. However, no matter scaling NF inside the same server or across multiple servers, we only terminate packet processing while separating or copying the global/write state, which could be done very quickly (e.g., several microseconds).

3) State Manager: To localize all state and support scaling strategy, we introduce a *State Manager* to manage states during packet processing according to the consistency requirement. When NF scaling out starts, the state manager separates or copies 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 and new state in other servers for newly launched instances. For NF with global/read state (i.e., S2), we use the same state in the same server and a copy in other servers (with migration operation). For NF with global/write state (i.e., S3), we first separate or copy the state. If the newly launched instance locates at other servers, we need further migrate the state to that server.

For NF scaling in process, the state manager merges states of recycled instance to other running instances of the same NF. If no running instance exists, the state manager stores state to persist storage for future access. For instance failure, similar to [18], NF can easily recover by launching a new process and reloading state from shared memory. Due to space limitation, we omit their detailed workflows.

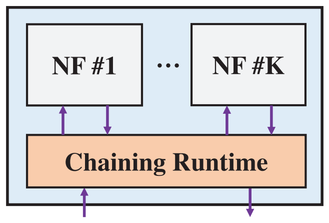
Improvement Mechanism	**Launch Approach**	**Overhead (μs)**
+ No forking	kThread Creat	23.66
+ No creating	kThread Pool	5.57
+ Reduced context switch	uThread Pool	0.46
+ No context switch	**Function Call**	**0.11**

TABLE II: Available NF execution models. kThread represents kernel-level thread, uThread represents user-level thread. ”+” represents added mechanism upon the previous one.

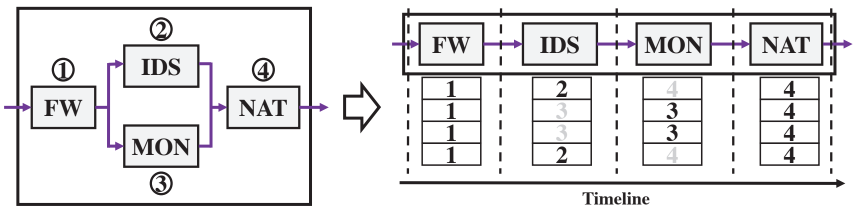
B. NF Execution

As mentioned in § II-C, existing NF execution model shows significant performance overhead. First, common NF launching approach starts a new process for every batch of packets. Second, current service chaining approach delivers packets between every two NFs. However, both process starting and packet delivery incur much higher overhead than actual packet processing and severely degrade the performance of serverless platform. The essential cause of performance overhead is *the coupling between resource unit and NF execution*, which starts a new process for every NF with every batch of packets. Based on this analysis, we propose a resource-NF decoupling mechanism and perform NFs execution of the entire SFC for all packets within one process. Although such a mechanism can lead to weaker isolation, our reasoning is that packets and NFs of an SFC belong to the same user and not requires such strong isolation as between SFCs [16]. Besides, users still benefit from modularization by providing their NFs as functions and not aware of this variation.

Execution model for NF launching. When coupling resource unit with NF execution, Packet Fetcher has to start new processes for every batch of packets. TABLE II lists the possible NF launching approaches and their overhead. Compared with forking overhead of 249.72 μs mentioned in §II-C, kernel-level threading (first row of TABLE II) replaces process forking with thread creating, reducing the launching time to 23.66 μs. Kernel-level thread pool (second row of TABLE II) also helps alleviate the launching burden to 5.57 μs as it launches NFs by simply selecting active idle threads. User-level thread pool (third row of TABLE II) can further reduce the launching overhead to 0.46μs by avoiding context switches between kernel and user space.



(a) SFC support with `Serpens` model



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

Fig. 8: Service chaining approach in `Serpens`

Despite that user-level thread pool could significantly reduce NF launching overhead, it trades memory consumption for launching time and still requires context switch between user threads. Actually, so long as the coupling between resource unit (e.g., processes or threads) and NF execution exists, overhead introduced by resource initialization and context switches cannot be fully avoided. To address this problem, we eliminate this coupling and utilize *function call* for NF launching (last row of TABLE II) to reduce the overhead to only 0.11 μs. As mentioned in § II-C, instance launches multiple workers to hide remote state access latency while we merge all resources into one worker. However, benefiting from our state management mechanism (i.e., state localization), state access latency is significantly slashed. Besides, running multiple workers simultaneously has the potential to necessitate synchronization primitives to ensure correct packet processing (e.g., increasing the counter in Monitor), which could degrade the performance. Therefore, NF execution with a single worker is better for `Serpens`, which implies that launching NFs with function call is practical and beneficial.

Execution model for service chaining. The current service chaining approach in serverless platforms depends on a centralized or distributed routers to delivery packets among separate NF instances (§ II). However, due to the coupling of resource unit and NF execution, this approach can lead to two problems. 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-server/inter-core communication and cache threshing is unavoidable.

To address the above problems, we also eliminate this coupling by consolidating multiple NFs in 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. 8(a) shows the service chaining approach in `Serpens`. We design a *Chaining Runtime* in each instance for service chaining. It sequentially calls NFs according to the SFC requirement.

NF executor

As mentioned in Fig 3, the Packet Fetcher collects packet batches and delivers them to the corresponding workers (NF logic). Different from running NF logic as processes, for a new batch of packets, the Packet Fetcher directly invokes the corresponding NF through function call. Such an approach completely eliminates the worker concept inside NF instances. The only process running in the instance is the Packet Fetcher itself. Therefore, there is only one process (i.e., the Packet Fetcher process) inside every instance.

When it comes to SFC, NFs are sequentially called by the Chaining Runtime (the same as Packet Fetcher). This mechanism remains the same when considering the situation where 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. 8(b) shows an example of the dynamic SFC. First sent to the Firewall (FW), all packets for this SFC will be sent to the IDS for further inspection if detected suspicious; otherwise 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 them [15]. However, it introduces complex scheduling logic to the runtime, which may burden the runtime and induce performance degradation. To design a light-weight runtime, our key idea to support 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 runtime 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. 8(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", indicating that the first and fourth packet should be processed by IDS (②) while others processed by MON (③). In this way, we simply call all NFs sequentially to achieve the same effect as the dynamic SFC.

C. Programming Abstraction for NFs

`Serpens` proposes new state management and service chaining approaches to improve the performance of serverless NFV, which requires NF developers to adapt `Serpens` mechanisms. In order to ease NF development, `Serpens` provides a programming abstraction in the format of APIs to hide the complexity of state management and service chaining. Fig. 9 presents the 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. Developer needs to indicate the category of state stored in the table by setting

```
//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. 9: Programming abstraction in Serpens

flags, including *per-flow, global, read and write*. Further, if the flag is global | write, guaranteed consistency for the state, including *release* and *eventual*, should be indicated. The *get_state* , *put_state* and *delete_state* functions provide operations of state access, update and deletion.

For global/write state, extra APIs, including *split*, *merge* and *update* functions, are called to split, merge or update them 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 resource by default. And 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 summing up host statistics) or on demand via a *trigger* function (e.g., NAT adjusting available IPs/Ports when the IPs/Ports of an instance are exhausted).

Service chaining. We provide three APIs to support 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 branches. 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.

IV. Implementation and Evaluation

A. Implementation

We have developed a prototype of Serpens based on DPDK [26], and Fig. 10 shows the detailed implementation upon a cluster of servers. The implementation is basically consistent with existing serverless platforms. Here, we mainly describe the *routing system* and *instance startup strategy* in Serpens, which are different from existing platforms.

1) Routing system: We introduce a *Router* in each server to deliver packets and achieve dynamic packet forwarding to

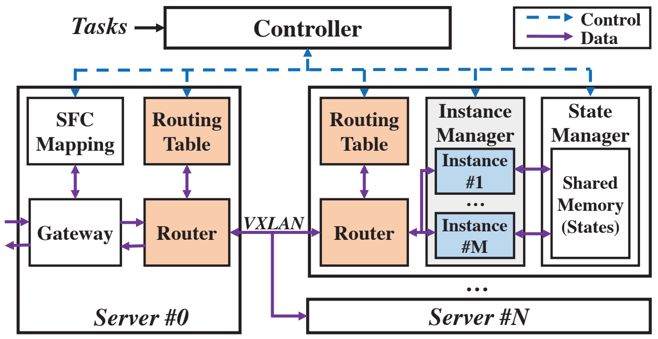


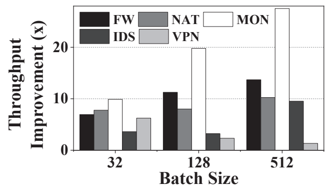
Fig. 10: Implementation of Serpens

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 launched for an SFC, other fields (e.g., 5-tuple) could be matched to distribute packets among instances. Thus, when launching or destroying instance on demand, the controller needs to update the corresponding Routing Tables to ensure consistent packet delivery. To maintain compatibility with current network, we leverage the *VXLAN* [27] to forward packets among different servers and tag the triggered SFC on reserved field in VXLAN header.

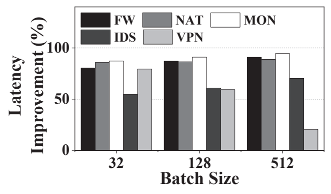
2) Instance startup strategy: We introduce an *Instance Manager* in each server to launch or destroy instances on demand. Considering the high startup latency of ”cold” start launching approach, we maintain a ”warm” instance pool in advance. To reduce the unnecessarily resource occupation with idle instances, we adopt a common practice by reusing launched instances by keeping them ”warm” for a period of time. Fig. 8(a) shows the layout of instances. To save resources and share CPU cycles, the Chaining Runtime actively sleeps during idle time and is woken up by router if new packets arrive. However, router sending signals for every packet or massive accumulated packets could either lead to significant resource overhead or long waiting time. In our current implementation, we choose a moderate number size of packets (128) and it can be adjusted at runtime according to the requirement of users.

To evaluate the performance of Serpens, we implement five NFs including *FW* (Firewall, performing 5-tuple hash matching 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 (2.2GHz, 12 physical cores), 128GB total memory (DDR4, 2400MHz, 16GB x8) and one dual-port 10G NICs (Intel X520-DA2). The other two servers run the gateway and packet generator, each is equipped with two Intel Xeon E5-2620 v3 CPUs (2.4GHz, 12 physical cores), 128GB total memory (DDR4, 2133MHz, 16GB x8) and one dual-port 10G NICs



(a) Throughput improvement of different NFs



(b) Average latency improvement of different NFs

Fig. 11: Performance improvement of state localization

(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 latency and throughput. The servers running gateway and deployed NFs are connected through an ethernet switch (10Gbps per port, 24 ports). All servers run Ubuntu 14.04 (kernel 4.4.0) and use DPDK 18.11 for networking I/O.

Evaluation goals. We evaluate `Serpens` with the following goals: (1) the performance improvement of state localization and scaling effect of performance fluctuation avoidance strategy (§IV-B); (2) the performance of different NF execution models for both local and remote state access (§IV-C); (3) the benefit from new service chaining approach (§IV-D); and (4) the end-to-end performance improvement for NFs and SFCs over existing serverless platforms (§IV-E).

B. State Management

Benefits from state localization To demonstrate the performance improvement of state localization in `Serpens`, we compare it with remote state access in existing serverless platforms. For the remote state access, we use Redis [21], a popular key-value store, to store NF state and locate it at the same server with NF instances to avoid additional latency on physical network. Besides, we aggregate multiple state access requests into a single one to optimize throughput. For NF execution, we adopt the kThread pool model to reduce launching time. We vary the batch size of packets and measure the performance of different NFs.

Fig. 11(a) shows the throughput improvement of local state access over remote state access. From this figure, we can observe significant throughput improvement, an average of 10.6×, 8.7×, 19.1×, 5.5× and 3.3× for FW, NAT, MON, IDS and VPN, respectively. Besides, we can see that MON improves the most than other NFs, this is because MON has frequent state access, thus occupies more time on remote state access. Fig. 11(b) shows the average latency reduction under 30% of the maximum throughput. We also observed significant latency reduction, an average of 86.1%, 87%, 90.9%, 62% and 53.1% for FW, NAT, MON, IDS and VPN, respectively. Here, we use Linux kernel based TCP stack for Redis communication, which could become the performance bottleneck under high-speed packet rate and magnify the overhead of remote state access. A common optimization is leveraging user space TCP stack [28] or RDMA [29] (crossing servers). However, as reported in [18], even with them, the additional remote access latency is inevitable and not ignorable.

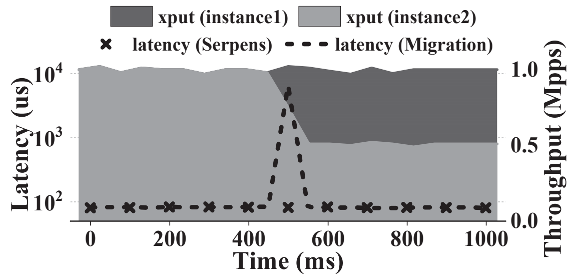
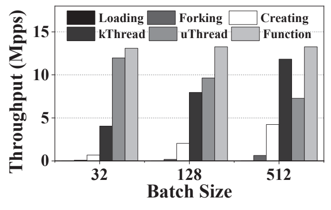
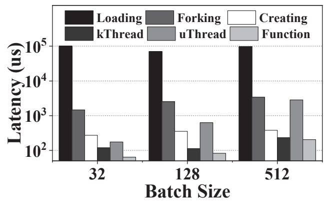


Fig. 12: Performance fluctuation of NAT during scaling



(a) Throughput with different NF execution models



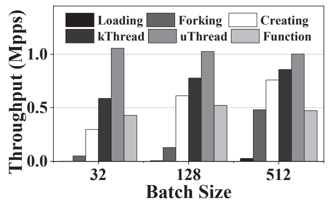
(b) Average latency with different NF execution models

Fig. 13: Performance of local state access (”Creating” represents ”kThread create” model, ”kThread” represents ”kThread pool” model.)

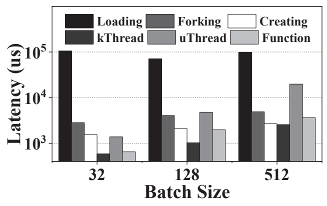
Performance fluctuation avoidance strategy. To demonstrate the benefit of performance fluctuation avoidance strategy in `Serpens`, we compare it with state migration based solution (e.g., OpenNF [17]). Here, we do not compare `Serpens` with solution that depends on remote state access to avoid state migration, e.g., StatelessNF [18], due to their low performance (as evaluated above). We choose NAT as our evaluated NF since it has global/write state which needs to be separated during scaling. We first launch an instance for this NF and send 1 Mpps traffic with 1k concurrent flows. Then we launch a new instance at the same server and migrate half flows to it. Fig. 12 shows the performance changes during scaling. Since both approaches could migrate flows between instances in a short time, we only show the throughput changes of the two instances in `Serpens` to demonstrate its effective at balancing load. However, `Serpens` almost incurs no latency variation during scaling while the migration based approach leads to a high latency spike.

C. NF execution

NF execution models for local state access. We evaluate the end-to-end performance of different NF execution models (discussed in § III-B). We choose Firewall as our evaluated NF and vary execution models. Fig 13(a) shows the throughput of these approaches with different batch sizes. We can see that the function call approach achieves the highest throughput and outperforms than others by 1.4× $\sim$ 3442.3×. Fig 13(b) shows the average latency with varying batch size, which is also measured with 30% of the maximum throughput. We can see a similar result as throughput in latency, with an average reduction of 99.8%, 95.4%, 66.5%, 28.7% and 80.1%, respectively. We can also observe that with batch size increasing, the throughput of other models increases accordingly, and

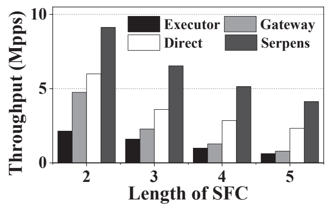


(a) Throughput with different NF execution models

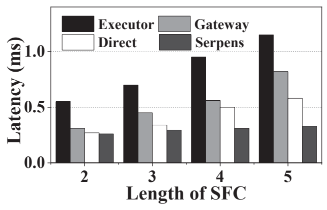


(b) Average latency with different NF execution models

Fig. 14: Performance of remote state access



(a) Throughput with different service chaining approaches



(b) Average Latency with different service chaining approaches

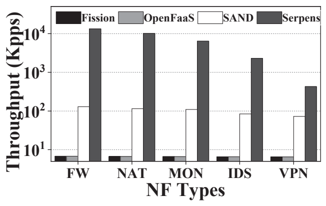
Fig. 15: Performance of SFC support

even some approaches could be very close to our approach (e.g., kThread pool). This is because their launching overhead can be amortized by batching more packets with trading off latency. However, our approach could achieve high throughput and low latency simultaneously. Besides, the performance of uThread Pool becomes worse with batch size increasing, that is because its scheduling policy (i.e., Round Robin) is not suitable for packet processing [30]. However, even with a perfect scheduling policy, it still cannot avoid the overhead caused by synchronization and context switch between threads.

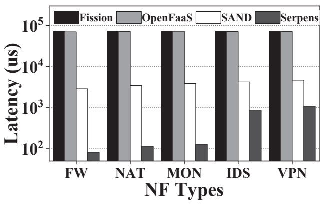
NF execution models for remote state access. However, our execution model may be not suitable for remote state access because of long blocking time. To evaluate this, we use the same evaluating method as the above. Fig 14(a) and Fig 14(b) show the throughput and average latency with varying batch size. We can see, the function call approach performs poorly at throughput as well as latency, and thus cannot be applied to remote state access. In contrast, the thread pool is more suitable for remote state access because of its high concurrency and relatively low launching overhead.

D. Service chaining

`Serpens` model for service chaining. `Serpens` model could improve performance by avoiding queuing delay and context switch. To prove this demonstration, we compare it with the available service chaining approaches in existing serverless platforms (shown in Fig. 4). To ensure equal resource consumption, we place all NFs of an SFC at the same CPU core. We chain multiple Firewalls to form an SFC and vary its length to measure performance. Fig 15(a) shows their throughput with varied length. We can see that our approach (based on `Serpens` model) achieves the best throughput in all cases and outperforms others by $1.7\times \sim 5.1\times$. Fig 15(b) shows their average latency. We can also observe that the average latency can be reduced from 24.5% to 62.3%.

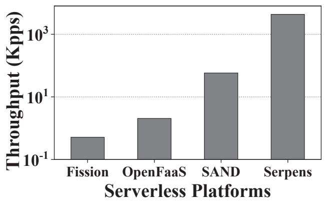


(a) Throughput with different serverless platforms

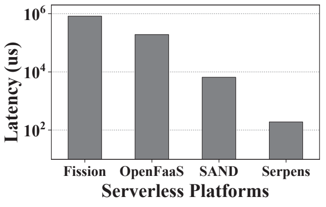


(b) Average latency with different serverless platforms

Fig. 16: End-to-end performance of NFs



(a) Throughput with different serverless platforms



(b) Average Latency with different serverless platforms

Fig. 17: End-to-end performance of SFC

E. End-to-end performance improvement

To demonstrate the performance improvement of `Serpens` over existing serverless platforms, we compare it with three popular open-source platforms, including Fission [13], OpenFaaS [12] and SAND [16]. These platforms are developed with Java or Go, which is not efficient for packet processing. Therefore, we rewrite their packet I/O with DPDK and use C to develop NFs. Specifically, all these platforms leverage a storage service (local or remote) to store state. For NF execution model, Fission and OpenFaaS launch NFs by loading binary files, SAND launches NFs by forking processes. For SFC support, Fission adopts the executor based chaining approach (Fig. 4(a)), OpenFaaS adopts the gateway based chaining approach (Fig. 4(b)), and SAND adopts the direct chaining approach (Fig. 4(c)).

Performance improvement for NFs. Fig 16(a) shows the throughput of different platforms with various NFs. We can see `Serpens` improves throughput for all NFs. For example, comparing with SAND that achieves the best throughput in existing serverless platforms, `Serpens` can outperform it by $103.3\times$, $88.5\times$, $58.8\times$, $24.8\times$ and $5.9\times$ for measured NFs, respectively. Fig 16(b) shows their average latency. We can observe a similar result, which is 97.1%, 96.6%, 96.1%, 79.4% and 76.7% latency reduction compared with SAND (also the best existing platform). We can also observe that if an NF becomes complex, the performance improvement decreases accordingly. That is because the packet processing time occupies more portion of the total time and our optimization becomes less critical. However, even with the most complex NF (e.g., VPN), `Serpens` still improves throughput by $5.9\times$ and reduces latency by 76.7% over SAND.

Performance improvement for SFCs. We chain Firewall, Monitor and NAT to form an SFC. Fig 17(a) and Fig 17(b) shows the throughput and average latency of this SFC on different platforms. We can see that even compared with the highest performance of existing platform (the SAND platform), `Serpens` can still improve throughput by 74.2$\times$ and reduce latency by 97.1% .

V. Related Work

High-performance NFV platforms. Many NFV platforms have been proposed to enable high-performance packet processing. For example, NetVM [14] consolidates multiple NFs into one server and accelerates packet delivery among NFs through shared memory. SoftNIC [31], NetBricks [15] abandons virtualization technology and run NFs in a single process to improve performance. Recently, many researches propose to run NFs on public cloud to relieve the management burdens and leverage the economies of elastic scaling [7]. However, none of them focus on designing a special-purpose cloud aware platform to support NFV. `Serpens` is the first to introduce serverless computing to NFV and design a high-performance platform to run NFs and SFCs.

Serverless computing. Existing serverless computing researches can be divided into two categories. This first category aims to support various applications by utilizing serverless computing [7]–[9]. For example, Jonas et al. [8] target at supporting distributed computing on serverless platform. Fouladi et al. [9] manage to support video processing on AWS Lambda. Singhvi et al. [7] evaluated that NFV is not suitable to run on the current AWS Lambda and design a framework for NF executions on AWS Lambda. The second category attempts to improve current serverless platforms for further purposes. Boucher et al. [6] reduces launching time by language-based isolation. Pocket [20] promotes serverless analytics by designing an elastic ephemeral storage. SOCK [22] reduces process initialization time in serverless platforms by its optimized container technology. SAND [16] distinguishes itself by its serverless platform designed for complex applications, but our evaluation has shown that it still incurs high performance overhead for NFV. Overall, none of them are specifically designed for NFV. `Serpens` address the performance problem and design a special-purpose serverless platform for NFV.

VI. Conclusion

We have presented `Serpens`, a high-performance serverless platform for NFV. Starting from identifying three performance problems in existing serverless platforms, `Serpens` proposes two novel designs to improve performance, including state localization and resource-NF decoupling. Moreover, `Serpens` provides a programming abstraction to ease NF development and SFC chaining. Our evaluations have demonstrated that `Serpens` can significantly improve the performance of NFs and SFCs over the state-of-the-art serverless platform.

Acknowledgement

We thank anonymous IWQoS reviewers for their valuable comments. This work is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (61625203, 61832013, 61872426). Mingwei Xu is the corresponding author.

References

[1] V. Sekar, N. Egi *et al.*, “Design and implementation of a consolidated middlebox architecture,” in *NSDI 12*, 2012, pp. 323–336.

[2] J. Sherry, S. Hasan *et al.*, “Making middleboxes someone else’s problem: network processing as a cloud service,” *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[3] M. Dobrescu, K. Argyraki *et al.*, “Toward predictable performance in software packet-processing platforms,” in *NSDI 12*, 2012, pp. 141–154.

[4] Z. Zheng, J. Bi, H. Yu *et al.*, “Octans: Optimal placement of service function chains in many-core systems,” in *INFOCOM 19*. IEEE, 2019.

[5] D. Xie, N. Ding *et al.*, “The only constant is change: Incorporating time-varying network reservations in data centers,” *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 199–210, 2012.

[6] S. Boucher, A. Kalia *et al.*, “Putting the” micro” back in microservice,” in *USENIX ATC 18*, 2018, pp. 645–650.

[7] A. Singhvi, S. Banerjee *et al.*, “Granular computing and network intensive applications: Friends or foes?” in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 2017, pp. 157–163.

[8] E. Jonas, Q. Pu *et al.*, “Occupy the cloud: Distributed computing for the 99%,” in *ACM SoCC 17*. ACM, 2017, pp. 445–451.

[9] S. Fouladi *et al.*, “Encoding, fast and slow: Low-latency video processing using thousands of tiny threads,” in *NSDI 17*, 2017, pp. 363–376.

[10] AMAZON. (2019) Aws lambda serverless compute. [Online]. Available: https://aws.amazon.com/lambda/

[11] Microsoft. (2019) Azure functions serverless architecture. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[12] OpenFaas. (2019) Serverless functions made simple. [Online]. Available: https://github.com/openfaas/faas

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

[14] W. Zhang *et al.*, “Opennetvm: A platform for high performance network service chains,” in *HotMiddlebox’16*. ACM, 2016, pp. 26–31.

[15] A. Panda, S. Han *et al.*, “Netbricks: Taking the v out of nfv,” in *OSDI’16*, 2016, pp. 203–216.

[16] I. E. Akkus, R. Chen *et al.*, “Sand: Towards high-performance serverless computing,” in *USENIX ATC 18*, 2018, pp. 923–935.

[17] A. Gember-Jacobson, R. Viswanathan *et al.*, “Opennf: Enabling innovation in network function control,” in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 163–174.

[18] M. Kablan *et al.*, “Stateless network functions: Breaking the tight coupling of state and processing,” in *NSDI 17*, 2017, pp. 97–112.

[19] S. Woo, J. Sherry *et al.*, “Elastic scaling of stateful network functions,” in *NSDI 18*, 2018, pp. 299–312.

[20] A. Klimovic *et al.*, “Pocket: Elastic ephemeral storage for serverless analytics,” in *OSDI 18*, 2018, pp. 427–444.

[21] S. Sanfilippo, “Redis: High performance in-memory data structure store,” 2018. [Online]. Available: https://redis.io/

[22] E. Oakes, L. Yang *et al.*, “Sock: Rapid task provisioning with serverless-optimized containers,” in *USENIX ATC 18*, 2018, pp. 57–70.

[23] H. Sadok, M. E. M. Campista *et al.*, “A case for spraying packets in software middleboxes,” in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, 2018, pp. 127–133.

[24] P. Keleher, A. L. Cox *et al.*, *Lazy release consistency for software distributed shared memory*. ACM, 1992, vol. 20, no. 2.

[25] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Queue*, vol. 11, no. 3, p. 20, 2013.

[26] Intel. (2019) Data plane development kit (dpdk). [Online]. Available: http://dpdk.org

[27] M. Mahalingam, D. Dutt *et al.*, “Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks,” Tech. Rep., 2014.

[28] E. Jeong, S. Wood *et al.*, “mtcp: a highly scalable user-level {TCP} stack for multicore systems,” in *NSDI 14*, 2014, pp. 489–502.

[29] Roce. (2019) Rdma over converged ethernet. [Online]. Available: https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet

[30] S. G. Kulkarni *et al.*, “Nfvnice: Dynamic backpressure and scheduling for nfv service chains,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 71–84.

[31] S. Han, K. Jang *et al.*, “Softnic: A software nic to augment hardware,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.